COMPOSITE OBJECTS
DYNAMIC REPRESENTATION AND ENCAPSULATION BY STATIC CLASSIFICATION OF OBJECT REFERENCES

ULF SCHÜNEMANN

Composite Objects

Dynamic Representation and Encapsulation by Static Classification of Object References

by
© Ulf Schünemann

A thesis submitted to the
School of Graduate Studies
in partial fulfilment of the
requirements for the degree of
Doctor of Philosophy

Department of Computer Science
Memorial University of Newfoundland

June 2005

St. John's Newfoundland

Your file Votre référence
ISBN: 978-0-494-15662-9
Our file Notre référence
ISBN: 978-0-494-15662-9

Abstract

The composition of several objects to one higher-level, composite object is a central technique in the construction of object-oriented software systems and for the management of their structural and dynamic complexity. Standard object-oriented programming *languages*, however, focus their support on the elementary objects and on class inheritance (the other central technique). They do not provide for the expression of objects' composition, and do not ensure any kind of encapsulation of composite objects. In particular, there is no guarantee that composite objects control the changes of their own state (*state encapsulation*).

We propose to advance software quality by new program annotations that document the design with respect to object composition and, based on them, new static checks that exclude designs violating the encapsulation of composite objects' state. No significant restrictions are imposed on the composite objects' internal structure and dynamic construction. Common design patterns like Iterators and Abstract Factories are supported.

We extend a subset of the Java language by *mode* annotations at all types of object references, and a user-specified classification of all methods into potentially state-changing *mutators* and read-only *observers*. The modes superimpose composition relationships between objects connected by paths of references at run-time. The proposed mode system limits, orthogonally to the type system, the invocation of mutator methods (depending on the mode of the reference to the receiver object), the permissibility of reference passing (as parameter or result), and the compatibility between references of different modes. These restrictions statically guarantee state encapsulation relative to the mode-expressed object composition structure.

Dedicated in gratitude to my parents.

In memory of my father.

Contents

List of Figures

List of Symbols

$\boxed{c}$	class named c
$\boxed{\underline{o}}$, $\boxed{\underline{o{:}c}}$, $\boxed{\underline{{:}c}}$	object named o, object o of class c, unnamed object of class c
`x`: $\boxed{\;v\;}$	variable `x` with value v
$\longrightarrow$	object reference
$\xrightarrow{\mathsf{m()}\blacktriangleright}$	message `m()` sent via object reference
$\dashrightarrow$	dependency between classes
$\longrightarrow\!\!\triangleright$	generalization, or Java's `extends` relationship between classes
$\dashrightarrow\!\!\triangleright$	realization, or Java's `implements` relationship between classes
$X \doteq Y$	formulæ X and Y are defined and $X = Y$
$X[y/x]$	substitution of y for x in term, sequence, or set X
$\nrightarrow$	partial map
$\uplus$, $\setminus\!\!\setminus$	multiset-union and multiset-subtraction
ϵ	empty path, empty sequence of association roles
X^*	set of sequences of X's (Kleene closure)
$\vec{x}$, $\overline{\tau\ x}$	sequences $x_1, \ldots, x_n$ and $\tau_1\ x_1, \ldots, \tau_n\ x_n$
$\pi_1 \circ \pi_2$, $\vec{\alpha}.\vec{\gamma}$	concatentation of paths, and of association role sequences
$\mu_{\mathbf{r}} \circ \mu$	adaption of mode μ imported through $\mu_{\mathbf{r}}$
$\leq_{\mathrm{m}}$	mode compatibility relation
$\Sigma(c)$, $\Sigma(\mu\ c)$	signature of c-objects, and of μ c-references
$\alpha, \beta, \gamma \in \mathbb{A}$	association roles, and the set of association roles
$c, d \in \mathbb{C}$	class names, and the set of class names
e	expression, runtime term
η, ϱ	method-local and object-local binding environment
Γ	type assignment, typing environment
$\mathfrak{g}$	object graph
h	object reference value (handle)
κ	kind of method
$\mu \in \mathcal{M}$	mode, and the set of valid modes
$o, \omega, q, u, v, w \in \mathbb{O}$	object identifiers, and the set of object identifiers
π	path of object references
$\mathit{Sanc}(o)$, $\mathit{StRep}(o)$	o's sanctuary and state representation
$\mathfrak{s}$	store
t, τ	type

Chapter 1

Introduction

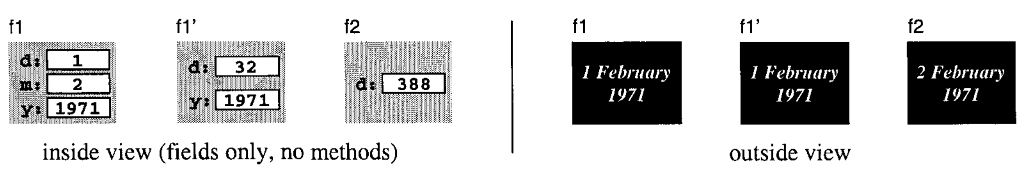
1.1 Summary

This section gives a gentle introduction to the thesis, without formalism, UML diagrams, and program code (mostly).

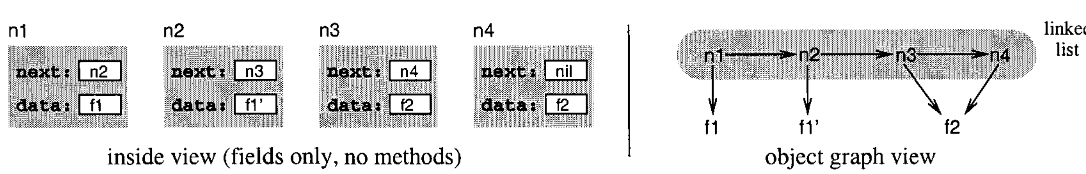
1. THE OBJECT ABSTRACTION is the central concept of object-oriented programming. It neatly integrates ***data*** and ***behavior***, the two foundations of computation, into one unit. Not only is the program partitioned into *class modules* that define types of objects by combining definitions of *data fields* (also called "instance variables," "attributes," "data members" or "slots") on one side, and *methods* (also called "operations" or "member functions") on the other side. Also the runtime system is partitioned into objects that collaboratively, on one side, represent the program's data in their fields and, on the other side, carry out the program's computation by executing their methods. The architecture of an object-oriented system is made of *objects* as the active components, and *references* between them as the collaboration-enabling connectors: Object references transport the requests for method executions (operation invocations) from caller to callee, and return the results back to the caller. This architecture can change dynamically by the creation and destruction of objects and object references.

Three related notions of "object" can occur in the description of software systems, as the following paragraphs shall illustrate: At the base-level, the system is a flat "sea" of elementary ***implementation objects***, i.e., instances of concrete classes that have only the fields and methods defined by their class. This is the perspective of object-oriented programming languages. Above that, structures of collaborating objects rooted in a "representative" object can be seen as one ***composite object*** with the fields and methods of the representative, and additionally component objects (possibly composite). The view of the system as a hierarchy of nested composite objects corresponds to the structure of canonical recursive top-down refinement or bottom-up composition of the system in object-oriented design. Finally, each object, if seen from the outside, is an ***abstract object*** defined solely by its operations' externally visible behavior. Abstract objects are classified by *abstract classes* (also called "interfaces" or "types"), which are specifications of their instances' public operations, but leave it to concrete subclasses to define the fields and methods to implement them by the field-manipulation and cooperation with component objects.

2. DATA REPRESENTATION. With objects, data can be represented at runtime in several forms: First, the object abstraction supports ***data abstraction*** at a basic level by allowing one to use the values of objects' fields as the concrete representation of some data to which the outside has access in an abstract fashion through their methods. Objects (if they have methods to manipulate and return field values) *are* data abstractions in the external view. Since the types of abstract data are defined by the behavior of the operations on them, the classification of data abstractions is supported in form of the classification of objects by abstract classes. For example, calendar dates can be reified in software by objects with operations `year`, `month`, `dayOfMonth` and, maybe, `dayOfYear` (abstract class `Date`). *Implementation classes* then subclassify `Date` according to the used representation scheme for dates. The realization of the representation schemes *year + month + day-of-month*, and *year + day-of-year*, and *days since 1 January 1970* (in unix) by the objects f1, f1′, and f2 is depicted below:

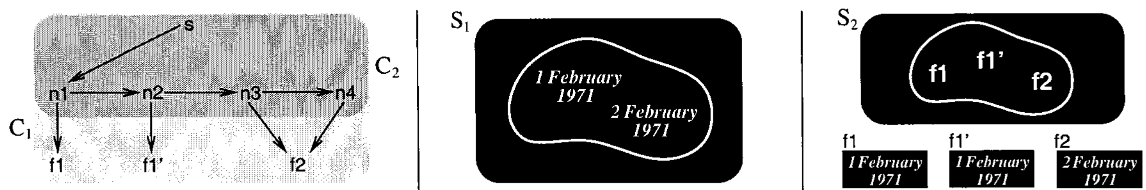


Second, the object abstraction supports ***linked data structures*** like double-linked lists, rings, trees, etc., by allowing objects to capture references to one another in their fields (whether or not they use them for message exchange). For example, instances of a class `Node` can be used as the nodes of a *single-linked list* by using one field for the link to the next node and another field for the value at that node. E.g. we can store the above objects f1, f1′, and f2 in a list of linked `Node` objects (once or repeatedly):



The graph which captures the structure of objects' interconnection by all object references in a particular state (i.e., which object currently has a reference to which object?) is called the ***object graph***. In general the graph includes more than the object references representing data structure links and stored data values—and thus models not only the data structures in the system. It also includes all references through which operation request messages may be sent between objects—and thus models the system's architecture.

Third, the object abstraction supports ***abstract data structures*** (sets, stacks, dictionaries/maps, etc.) as the instances of abstract classes represented by not just a single implementation object but an entire structure of objects, a *composite object.* Its "representative" is the instance of the concrete class implementing the abstract class. The representative implements the abstract object's behavior, i.e., the abstract data structure's behavior in this case, by going beyond being a data abstraction, and interacting (directly and indirectly) with the other objects in the structure, the *subobjects*, to make use of their behavior too. For example, a set can be represented by adding a representative s with a reference to the above list's initial node n1, and with suitably implemented set-operations contains, size, Add, Remove, etc.:



object graph view (a/b) outside view (a) outside view (b)

Depends on the methods' external behavior, this structure can represent two different types of set abstractions. (In C++, these types could be written set<Date> and set<Date*>.) A *set-of-dates* data structure S_1, that reifies the set {*1 February 1971*, *2 February 1971*} of two dates, would be implemented if size() returns two and contains(o) returns true for all Date objects o representing *1 February 1971* or *2 February 1971.* A *set-of-*Date*-objects* data structure S_2, that reifies the set {f1, f1′, f2} of three software objects, would be implemented if size() returns three and contains(o) returns true exactly for $o \in \{\mathsf{f1}, \mathsf{f1}', \mathsf{f2}\}$. Note that in the former case, the data in the Date objects' fields is part of the concrete representation C_1 of the set abstraction S_1, and s will have to send messages to the Date objects to find out what dates they represent. In the latter case, what the Date objects represent is irrelevant for the set, and there is no interaction between s and them. That is, only s and the Node objects constitute the composite object C_2 representing the set abstraction S_2. The Date objects are separate data abstractions in this case.

One can use a set-of-Date-objects composite C_2 to construct an alternative representation C_1' of the set-of-dates abstraction S_1: Simply place a representative s′ in front of s to adapt the methods' behavior: s′'s Remove method removes from C_2 any Date object representing the given date; and instead of adapting size and contains, it is easier to adapt the Add method to filter out Date objects representing dates already represented by C_2's Date objects. (It would not be a good idea to obtain C_1' not by object composition but by subclassing the implementation class of C_2's representative s: Set-of-dates is not a specialization, not a (behavioral) subtype, of set-of-Date-objects.) We will come back to C_1' in paragraph *8.*

3. Notions of State. In the course of the computation, the values of objects' fields can change and, through this, the object graph and the set of a composite's sub-objects. In programming languages, the notion of an (implementation) object's state is defined as the combination of its fields' current values [Bi$^+$80, GR83, GJS00, ISO98]. This is also called the object's *shallow state* and contrasted to its *deep state*, which is the name for the combination of shallow states of all objects reachable from the object via paths of object references captured in fields. The state of a composite object, the ***composite state***, is something in-between these two extremes: In general, only a certain portion of the objects reachable from the composite's representative along field-captured references belong to the composite object as sub-objects that contribute their shallow states to the composite's state. (Objects reachable only via references local to some method invocations cannot contribute to the composite's state since the references are inaccessible to new invocations of the composite's methods wanting to access the objects' states.) Which of the reachable objects are the "state-representing" sub-objects can be specified by the programmer using the mode annotations introduced further below. The set of the composite's state-representing sub-objects is called its ***state representation***. It will be formalized as the set $StRep(o) \subseteq \mathbb{O}$ of their object identifiers. The abstract state, i.e., externally visible state, of the composite as an abstract data structure (paragraph *2*) is the composite's methods' projection of the composite state to external behavior.

Note that the composite object's state (composite state) can change without any change in the corresponding representative's state (shallow state): *Example 1.* Updating the `d` field of `Date` object `f1'` to 33 or 34 makes it represent, respectively, *2 February 1971* or *3 February 1971*. Since `f1'` is a sub-object of composite object C_1, this is also a change of C_1's state. The first change is not visible in the outside view; the represented data structure S_1 is not affected. The second change has a side-effect: S_1 is now reifying a different, extended set {*1 February 1971*, *2 February 1971*, *3 February 1971*}. *Example 2.* Updating the `data` field of `Node` object `n4` to a new `Date` object `f3` representing the date *3 February 1971* is a change to composite objects C_1 and C_2, and thus a change to the representation of set abstractions S_1 and S_2. As a side-effect, it changes the set reified by S_1 to {*1 February 1971*, *2 February 1971*, *3 February 1971*}, and the set reified by S_2 to {`f1`, `f1'`, `f2`, `f3`}.

This dissertation will ensure that such side-effects of the change of `f1'` or `n4` can occur only as the part of `s`'s implementation of a state-changing mutator operation of the abstract set. That is, in the context of the abstract data structure's implementation, these are not unintended side-effects, but desired effects.

4. ENCAPSULATION. The notion of *private fields* means fields of an object that are hidden from outside and accessible only to that object's methods. (In *modular* object-oriented languages like C++, Eiffel, and Java, the meaning of a *private* field is that it is hidden from other class *modules* and accessible only by methods in the class defining that field, irrespective of the field's and the method's object.) Consequently, private fields' values can vary over the object's lifetime only in ways the object's own methods permit. This localization of the access to mutable state is a defining feature of object-oriented languages. By enforcing the hiding of private fields, they improve the modularity of the runtime system and help to predict and control its behavior. For the programmer, hiding fields is not really a severe restriction since whenever needed the object can provide, with minimal overhead, access to the field's value by operations *get-value-of-x* and *set-value-of-x*.

At the composite object level, one would expect a corresponding hiding of (state-representing) component objects. Note that this is not entailed by hiding the fields: If a field is private, this does not mean that the value in it is not shared. Other objects may possess the same value and, if the value is an object reference, use it to access the target object (through its operation interface). It is not uncommon that object references in private fields are shared: For example, in order to provide their clients access to their elements, `Set` objects like S_2 typically return an abstract iterator object which yields one element after another to the client.[1] The typical implementation of iterators would be for `Set` representative `s` to create a concrete iterator which uses a reference into the linked list to extract the data value from each node and return it. While this way the set object avoids making node components accessible to the client, it does make them accessible to the iterator object.

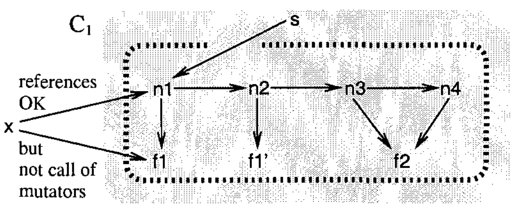
This means that a new mechanism is needed to restrict access to component objects, a mechanism that it less strong than hiding. It should enforce a new property called ***composite state encapsulation***: *A composite object's state can change only through its own operations*, and not by the side-effects described above). Consequently, between executions of the composite's methods the composite state cannot change, so that all invariants over it must remain intact. Hence state encapsulation is a global system property which is strong enough to extend modular reasoning about the representative to modular reasoning about the entire composite object. On the other hand, state encapsulation is weak enough not to exclude structure-sharing iterators and similar common patterns of object-oriented design.

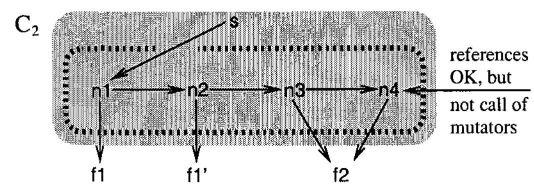
For the component objects, composite state encapsulation means that if they are state-representing then they cannot change state but on the initiative of the corresponding representative. For external objects, composite state encapsulation means that they may obtain references to the state-representing components, but they are read-only.

5. MUTATORS AND SANCTUARIES. The enforcement of state encapsulation by a static type system will be based on the declaration of all operations and methods as either '***mutator***' or '***observer***', and on metaphorically associating each object o with a protection domain, the ***sanctuary*** $Sanc(o)$: An object's fields may only be updated in its own *mutator* methods (*shallow state encapsulation*). And these mutators may be invoked on objects in o's sanctuary only from mutators of o and of objects in o's

[1] Iterators are an example that objects can not only be data abstractions but also *behavioral* or *process abstractions*: Rather than holding data, iterators reify the client's iteration process over the data stored in another object, much like a coroutine.

sanctuary (*mutator control* or "the sanctuary invariant"). Representative o is the only object outside of $Sanc(o)$ that is permitted to send mutators into the sanctuary. This means that all mutator executions in o's sanctuary have to be initiated by a mutator of o. Mutator control plus the shallow state encapsulation property means that field changes in representative o *and* in its sanctuary are possible only through a mutator of o. If o were included in its own sanctuary, $o \in Sanc(o)$, there would be no object to send the first mutator into the sanctuary.





The assignment of objects to a sanctuary will be based on certain paths of object references labeled by the programmer with modes, as explained further below. The mode-labeling and thus the assignment is independent from the references' storage in fields. Hence there may be objects in the sanctuary whose membership lasts just for one method invocation. To get the composite state encapsulated, the programmer has to assign all state-representing sub-objects (save o) to o's sanctuary $Sanc(o)$. That is, $StRep(o) \setminus \{o\} \subseteq Sanc(o)$ ("representation completeness"). (Since state-representing sub-objects' assignment must hold during and between method invocations, it must be established by paths of references that are captured in fields. However, since assignment will be based on the references' mode classification, not their storage place, sanctuaries may temporarily contain some non-state-representing sub-objects.)

By shallow state encapsulation, representation completeness means that any change in a state-representing sub-object requires the execution of a mutator by representative o or by an object in o's sanctuary (and thus also by representative o by mutator control). If then also no object can be added to, or removed from, the state representation $StRep(o)$ without o's mutators (*coherence*), any kind of change to the composite state can be affected only through o's mutators. Since these are, in the composite object view, the mutators of composite O, we have composite state encapsulation.

If one sanctuary includes another one, $Sanc(\omega) \subseteq Sanc(o)$, then the enclosing sanctuary's owner o can send mutators to objects in the nested sanctuary $Sanc(\omega)$ only indirectly via a mutator on ω. Membership in sanctuaries will be defined below so that it is transitive: $\omega \in Sanc(o) \Rightarrow Sanc(\omega) \subseteq Sanc(o)$. (This is consistent with the assumption that state-representing sub-objects of state-representing sub-objects of o also contribute to o's composite state, $\omega \in StRep(o) \Rightarrow StRep(\omega) \subseteq StRep(o)$.)

6. PATHS OF OBJECT REFERENCES. Membership of ω in o's sanctuary as well as the initiation of mutator executions in ω by mutators of o will be based on certain types of paths $o \dashrightarrow \omega$ of object references from o to ω in the current object graph. This dissertation proposes a classification of paths into types called ***modes*** $\mu \in \mathcal{M}$. The basic classification is five-fold:

- A ***rep path*** is a path $o \dashrightarrow \omega$ which means that ω is in o's sanctuary and in all sanctuaries containing o, but in no other. The programmer adds o's state-representing components ω to o's sanctuary by classifying paths $o \dashrightarrow \omega$ from o to ω as `rep`. (Of course, only paths made entirely of references captured in fields can persist between method invocations and thus effectively represent a piece of the composite state.) The proposed type system will ensure that no other object has a `rep` or `free` path to ω, so that o is ω's unique *owner*.
- A ***free paths*** is a path $o \dashrightarrow \omega$ meaning that ω is in no object's sanctuary (excluding `rep` paths to ω), and that all `free` paths to ω must start with the first reference of $o \dashrightarrow \omega$ (so that o is ω's unique *owner*). This meaning will be enforced by the proposed type system. Mode `free` is used for the temporary path to recently created objects that can still be moved to other objects, and that are currently used only locally within a method, like iterators. (Such objects can be understood as non-state-representing, temporary or "behavioral" components.)
- A ***co-path*** is a path $o \dashrightarrow \omega$ which means that ω is in the same (nested) sanctuaries as o, and that the extension $q \dashrightarrow o \dashrightarrow \omega$ of any path $q \dashrightarrow o$ of mode μ by $o \dashrightarrow \omega$ is another path of mode μ. Mode `co` is used for paths with *high cohesion*, like the references linking a data structure or connecting two tightly collaborating objects.
- An ***association path*** also extends other paths, but offers more flexibility in the extension's mode than a co-path. This category is needed for paths representing *semantic relationships* or *data values* like `Set` composite C_2's elements or the iterator's current element. The details will be explained in paragraph *8*.
- A ***read path*** is a path $o \dashrightarrow \omega$ that has no meaning for ω's status and does not extend other paths to a moded path to ω. All paths which are none of the above are classified as `read`.

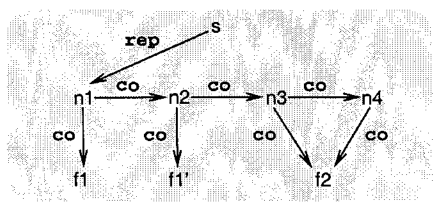
`Rep` and `free` paths $o \dashrightarrow \omega$ are both "ownership paths," guaranteeing that o is the unique owner of ω. We can superimpose an object composition meaning on all of them (state-representing or otherwise), and get a standard object composition hierarchy without shared components.[2] The type system moreover specializes and broadens the above mutator control property for sanctuaries to the ***mutator control path*** property: All mutator requests arriving at ω have (indirectly) been sent from o to ω *along one of its rep or free paths* π. That is, if π is $o = o_0 \rightarrow o_1 \rightarrow \ldots \rightarrow o_n = \omega$ then o invoked an operation on o_1, during whose execution o_1 invoked an operation on o_2, and so on, up to o_{n-1}'s invocation of the mutator on ω.

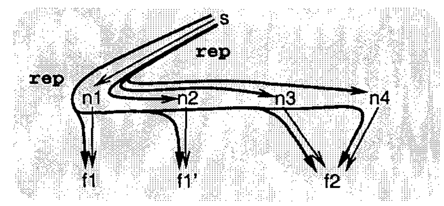
`Co`- and association paths *do not fix* their target's place in sanctuaries or in the object composition hierarchy. They obtain their relevance from *third objects' paths* to the path's initial object, which determine the combined paths' modes. Thus they

[2]Moded paths can be seen as representations of UML links: Ownership paths represent composition links o◆—ω, association paths of role α represent links $o \overset{\alpha}{—} \omega$ of public ordinary association α, `read` paths represent links of private or implicit associations, and `co`-paths represent links modeling integrative relationships, like between lintel and uprights in an arc [Ar+96].

are the basis for reducing the classification of larger paths to that of shorter ones, down to the single object references, whose modes one can actually declare in the program: In the example of date-set S_1 represented by composite object C_1, the first node is assigned to s's sanctuary by classifying the anchor reference s $\longrightarrow$ n1 as a rep reference. The remaining Node and Date objects in C_1 can then be placed into the same sanctuary by classifying all the links between them as co-paths. Then they extend the rep anchor reference to rep paths to each of C_1's components.

moded object graph



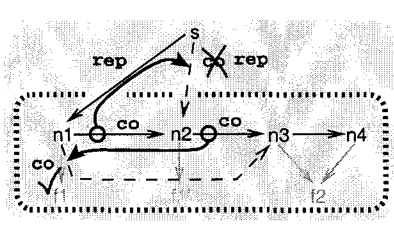


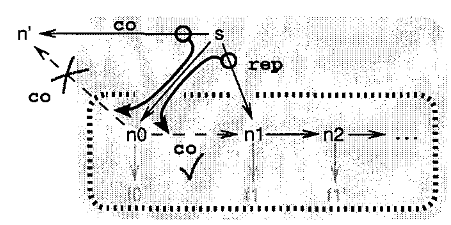
moded reference paths

7. EXCHANGE OF MODED REFERENCES. Since some modes' meaning is relative to the path's source, if references of such modes are exchanged between objects as parameter or result, their mode may have to be adapted to the new source. This is necessary to preserve the consistency of the moding of paths in the object graph and of the objects' assignment to sanctuaries.

For example, if DateSetImp representative s invokes next() on Node n1 which returns the co reference n1 $\rightarrow$ n2, then the reference s $\rightarrow$ n2 which s obtains must not be a co reference, since s cannot be in its own sanctuary *Sanc*(s). The return of the co reference can be better understood as the mode-preserving shortening of two-references path s $\xrightarrow{\text{rep}}$ n1 $\xrightarrow{\text{co}}$ n2 to a one-reference path s $\rightarrow$ n2: The reference which s obtains is a rep reference s $\xrightarrow{\text{rep}}$ n2. Should, on the other hand, one node n1 call next() on its co-object n2, then the returned reference's mode is not adapted, since the return simply shortens co path n1 $\xrightarrow{\text{co}}$ n2 $\xrightarrow{\text{co}}$ n3 to n1 $\xrightarrow{\text{co}}$ n3.

calls of next()





call of SetNext(n)

Analogously, the mode of references passed as parameters has to be adapted: If s has created a new Node object n0 in its sanctuary, then it should supply to n0's SetNext operation (expecting a co reference) one of its rep references, namely s $\xrightarrow{\text{rep}}$ n1, and not a reference s $\xrightarrow{\text{co}}$ n$'$ to a node that is a co-object in the same sanctuary as s (actually, in all the nested sanctuaries in which s resides).

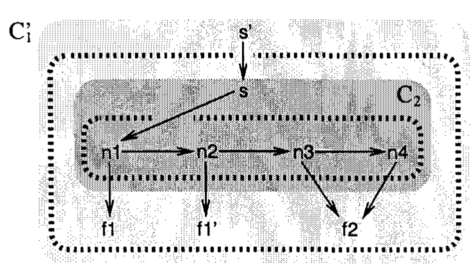
In general, the mode of a result or formal parameter *on the sender's side* of a call-link is an adaption $\mu_r \circ \mu$ calculated relative to the call-link's mode μ_r from the mode μ of the corresponding result or formal parameter of the receiver's operation. Consequently, two notions of interface have to be distinguished:

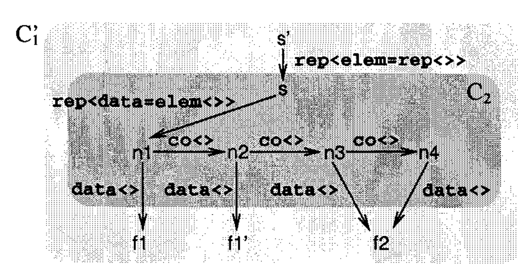
- ***Exported interfaces.*** The interfaces which all instances of a class c export have a (minimum) signature $\Sigma(c)$ defined by the class. Its entries $f : \overline{\mu_i\ d_i} \longrightarrow \mu\ d$ specify the types of the parameter values which implementations of operation f (can) expect to receive, and the type of the result values which they (must) ensure to produce. Against this signature, the operations' implementations in class c and its subclasses are type-checked.

- ***Imported interfaces.*** The interfaces which senders import through $\mu_{\mathbf{r}}$-references to c-objects have the signature $\Sigma(\mu_{\mathbf{r}}\ c)$ with modes from c-objects' signature $\Sigma(c)$ adapted relative to call-link mode $\mu_{\mathbf{r}}$. Its entries $f : \overline{\mu_{\mathbf{r}} \circ \mu_i\ d_i} \longrightarrow \mu_{\mathbf{r}} \circ \mu\ d$ specify the types of the parameter values which the sender must ensure to supply, and the type of the result values which the sender can expect to obtain. Against this signature, the clients of c-objects, who send invocation requests through call-links of type $\mu_{\mathbf{r}}\ c$, are type-checked.

This adaption is comparable in C++ to the signature $\Sigma(\texttt{const}\ c)$ of read-only access to records of type c, which is obtained from the general signature $\Sigma(c)$ of c-records by adapting the type τ of each field to `const` τ.

8. Flexible Extension by Association Paths. A classification with just the modes `rep`, `free`, `co`, and `read` is insufficient for constructing the alternative date-set composite C_1' explained in paragraph *2* from a given set-of-`Date`-objects composite C_2: Classifying the references $\mathtt{n}i \longrightarrow \mathtt{f}j$ stored in the nodes of C_2 as `rep`/`free` or `co` would modify C_2 directly into a date-set composite C_1 by making the `Date` objects components of the respective `Node` object or of `s`. Mode `read`, on the other hand, would leave the `Date` objects outside the set composite C_2, but then provide no basis for their inclusion in the composite C_1' with C_2 as a component. We need a more flexible extension of paths: The `Nodes`' data-references $\mathtt{n}i \longrightarrow \mathtt{f}j$ must be classified as *association* references extending `s`'s paths to the nodes to paths which can extend reference $\mathtt{s}' \longrightarrow \mathtt{s}$ to paths $\mathtt{s}' \dashrightarrow \mathsf{f1}, \mathsf{f1}', \mathsf{f2}$ of mode `rep`.





This requires us to refine the mode-classification of paths:

On one hand, association paths are subdivided according to an unbounded number of ***association roles*** $\alpha \in \mathbb{A}$ in order to distinguish different kinds of (object reference) data in an object, like references in a `Pair` object to its first element *vs.* its second element. This subclassification enables us to define different modes for the extension of a path by association paths of different roles. Syntactically, roles are plain identifiers, similar to labels. For instance, the role of the `Node`'s data-references could be called `data`, and the role of the element references stored in the abstract set-of-`Date`-objects S_2 could be called `elem`.

On the other hand, the classification of all paths by the ***base-modes*** $m \in \mathcal{B} = \{$free, rep, co, read$\} \cup \mathbb{A}$ encountered so far is refined according to the modes of extensions by association paths: A full mode $\mu \in \mathcal{M}$ is a base-mode m parameterized by ***correlations*** α_i=μ_i that specify that the extensions of μ-paths by α_i-paths have mode μ_i. Syntactically, a full mode thus has the general form $m\texttt{<}\alpha_1\texttt{=}\mu_1, \ldots, \alpha_n\texttt{=}\mu_n\texttt{>}$.

In the example, the nodes' data-references could be given association mode data<> and s's anchor reference to the first node the mode rep<data=elem<>>, so that s's reference paths $\mathsf{s} \dashrightarrow \mathsf{f1}, \mathsf{f1}', \mathsf{f2}$ to the objects in the nodes have association mode elem<>. These paths represent the elem references stored in the abstract data structure S_2 and held by S_2 in the external view ("virtual references," similar to virtual attributes). By giving s''s reference to s the mode rep<elem=rep<>>, its extensions $\mathsf{s}' \dashrightarrow \mathsf{f1}, \mathsf{f1}', \mathsf{f2}$ by s's elem paths are given the desired mode rep.

Association paths and correlations are crucial for the structural flexibility of the mode technique. They allow an object class to fix the modes of references in its instances without fixing the reference targets' assignment to a sanctuary. This decision is postponed to each instance's clients. (The type system ensures the consistency of the clients' decisions.) Hence the same class can be reused, in particular as a type of component objects, in many different structural contexts. For example, instances of the same Node class with data references of mode data<> could also be used in the date-set composite C_1 instead of those with co<> data (cf. paragraph *6*): Only change the mode of s's anchor reference to rep<data=rep<>>.

9. JAVA WITH MODE- & MUTATOR-ANNOTATIONS AND -CHECKS. The proposed language JaM is an orthogonal extension of a subset of the Java language by the keywords mut and obs written in front of the return type of *all* operations and methods, by modes $\mu \in \mathcal{M}$ qualifying *all* class names used as types of object references, and by static typing rules that check these annotations w.r.t. composite state encapsulation.[3] Figure 1.1 shows how the set-of-objects data abstraction S_2 and its C_2-realization would be declared in JaM. JaM's mode & mutator checks are orthogonal to Java's type checks since any legal Java program from the Java subset becomes a legal JaM program by annotating, respectively, mut and co<> everywhere: This places all objects into the same sanctuary, so that all mutator calls are legal.

The mode annotations specify a unique mode for all object references at any time during the execution: First, all object references *stored* in a variable (field, local variable, parameter) have their modes fixed to the mode μ which qualifies the class name c in the reference type $\mu\ c$ declared as the variable's range. Second, the temporary reference $o \to \omega$ which the sender o obtains when the receiver q returned reference $q \xrightarrow{\mu} \omega$ has the mode $\mu_{\mathrm{r}} \circ \mu$ that is an adaption of μ relative to the mode μ_{r} of the reference $o \to q$ through o made a call to q. Third, the mode of the

[3]In the formal treatment, a few additional annotations will be used for simplification: They will make explicit the destructive or non-destructive read access to a variable, and allow to assign modes with unique correlations to object creation expressions (new) and to null.

```
// type of set-of-objects data abstraction S2
interface Set {
  obs boolean contains(read<> Object o);
  mut void Add(elem<> Object o);
  mut void Remove(elem<> Object o);
}
// class of node-based realization's representative
class SetImp implements Set {
  rep<data=elem<>> Node anchor;
  ...
}
class Node {
  obs co<> Node next();
  obs data<> Object data();
  ...
}
```

Figure 1.1: Set-of-objects and node-based realization in JaM

initial reference to a newly created object is `free` (with correlations as specified by an additional annotation). From this classification of all paths of length one in the object graph, the classification of longer paths is derived inductively: Paths that are the extension of a μ-path $o \dashrightarrow q$ by a co- or α-path $q \dashrightarrow \omega$ have, respectively, the mode μ or the mode μ' if $\mu = m\texttt{<}\ldots, \alpha\texttt{=}\mu', \ldots\texttt{>}$.

In analogy to this, JaM's typing rules infer, besides the target class c, the modes μ of all object reference-valued expressions based on the modes of variables and results. In particular, the type of an operation call expression with receiver expression of type $\mu_{\mathbf{r}}\ c$ is the result type $\mu_{\mathbf{r}} \circ \mu\ d$ of the corresponding operation in the signature $\Sigma(\mu_{\mathbf{r}}\ c)$ of call-links of type $\mu_{\mathbf{r}}\ c$. Restrictions are imposed by the typing rules on the use of object references as values in order to preserve the properties of `rep` and `free` paths which entailed the safety of permitting mutator calls (as described further below): Object references assigned to μ-variables must have a ***compatible*** mode $\mu' \leq_{\mathrm{m}} \mu$. Object references supplied as actual parameter to operations with formal parameter mode μ in the signature $\Sigma(\mu_{\mathbf{r}}\ c)$ of call-links of type $\mu_{\mathbf{r}}\ c$ must have a compatible mode $\mu' \leq_{\mathrm{m}} \mu$. (Simplified, `free` mode $\texttt{free<}\delta\texttt{>}$ is compatible to any mode $m\texttt{<}\delta\texttt{>}$, any mode $m\texttt{<}\delta\texttt{>}$ is compatible to the `read` mode $\texttt{read<}\delta\texttt{>}$ with the same correlations, and `read` modes are compatible to `read` modes with fewer correlations or correlations to compatible modes.)

In order to enforce composite state encapsulation, additional restrictions are imposed by the typing rules on access to fields and operations through object references: In `mut`-methods, assignments to the fields of `this` and mutator invocations through references of base-modes `rep`, `free`, and `co` are permitted since they either cross into no sanctuary or just into the caller's sanctuary. In `obs`-methods, field assignments and mutator invocations through references of base-modes other than `free` are forbidden since only `free` references guarantee that the target is not in any sanctuary. Assignments to other objects' fields and mutator invocations through `read` and association references are never permitted.

1.2 Contributions

This dissertation is situated at the design-implementation boundary of object-oriented software development, where detailed object-oriented designs get implemented in object-oriented programming languages. The ultimate aim is to improve the modularity of object-oriented runtime system models that are structured by the design abstraction of composite objects. The means is the type-system of object-oriented programming languages extended by a system of type qualifiers called *modes*. Modularity is improved in form of the encapsulation of each composite object's *state*.

The main result is that the presented type system extension for Java guarantees composite state encapsulation as a global system property: Composite objects can change state only through the execution of their own (mutator) methods.

Most other proposals to encapsulate units of the runtime system are works in *alias control* [Hog91, DD95b, Utt96, KM95, Min96, Alm97, GTZ98, NVP98, CPN98, Cla01, ACN02] or *access control* [BC87, Hog91, AW$^+$92, Bos96, Kni96, KT99, GB99, CR00] with the general aim of simplifying controlling, and reasoning about, system behavior. This dissertation focuses, like [DLN98] and [MP99a], on *modularity* that enables the modular verification of object-oriented programs, employing alias and access control only in as far as it works to this end. To the research in modularity, the first description of the property of ***state encapsulation*** is contributed. It can be seen as capturing exactly that global system property needed for modular reasoning about composite objects based only on the code of the representative's class and superclasses, and on ordinary, postcondition specifications of called operations (of external and component objects).

The dissertation provides a flexible system for guaranteeing the encapsulation of every composite object at runtime by pure compile-time type checking. It enables the definition of nested composite objects with a complex internal structure, their observation through external iterator objects, their incremental construction (top-down and bottom-up), and their transfer across abstraction boundaries (one by one, linked to lists, or stored in containers). It supports design patterns like Iterator, Abstract Factory, and Builder [Ga$^+$95]. It is the first purely static system in which container objects and their iterator objects can each be encapsulated individually, i.e., state-protected from one another. (Others need runtime checks [MP99a, ACN02] or encapsulation barriers that are not aligned with object composition [Cla01, ACN02].) Composite objects can link their component objects to data structures or store them in a container object component. Nested container objects o can be built with a given, possibly also composite, container object o' (from an unknown implementation class) as their component. Their iterators i can be structured in parallel as composite objects with the container components' possibly composite iterators i' (from unknown implementation classes) as their components. All this will be demonstrated in the running example of set and map objects with iterators.

To the research on composite object encapsulation by type systems, this dissertation contributes a new technique which is based on a classification of ***paths of object references*** (with single references as a special case). Previous techniques based their type system extensions on aliasing properties or access rights *of object references* [Hog91, Min96, Kni96, Alm97, DLN98, KT99, GB99, ACN02], or on ownership parameters *to objects* [KM95, CPN98, MP99a, Cla01, ACN02]. (Only the informal description of *flexible alias protection* [NVP98] might be understood as using paths, although its official formalization in [CPN98] is based on ownership types.) In the proposed new technique, some types of paths entail aliasing or access restrictions, some have a superimposed object composition meaning (which, with state encapsulation, implies a form of ownership), and some let the path extend other paths.

The system of type qualifiers called ***modes*** is similar to that of *flexible alias protection* [NVP98]. But we provide a formal treatment using standard techniques of formal type systems and formal semantics (small-step with store and environment). The flexibility achieved by parameterizing the types of *objects* in flexible alias protection and other work [KM95, NVP98, CPN98, ACN02] is achieved in the mode system by the first proposal of type qualifiers [FFA99], namely modes, that are parameterized, namely by correlations. This move preserves the complete orthogonality of a reference's mode μ and the class c of its target in the types $\mu\ c$ of object references. Hence the addition of modes does not affect the soundness of Java's subtype polymorphism between object reference types based on subclass relationships between the objects' classes, of class inheritance, of class-parameterized generic classes (and methods), and of dynamic casts w.r.t. a reference's target class.

To alias control a novel weak uniqueness property is contributed, which is based on entire *paths* of object references: ***Free paths*** between two objects have unique *head* references and are not aliased by `rep` *paths*. This property generalizes Hogg's notion of 'free' references [Hog91, NVP98], and of the similar 'unique' [Min96, ACN02] and 'virgin' references [DLN98], which are not aliased by *any* (captured) reference at all. It allows us to rely not exclusively on *destructive* read for accessing the `free` reference in a variable, but to read the value *as a* `read` *reference* without resetting the variable. Due to `free` paths, the proposed mode system is the most flexible one w.r.t. dynamic object creation and composition proposed so far, decoupling object creation from object use (in particular, use as a composite's component).

Finally, to object-oriented software development and programming language design, this dissertation contributes a system of program annotations to document in the code the system's design w.r.t. object composition, and a system of static type checks to exclude designs of poor modularity w.r.t. composite objects. This is important since object composition, i.e., the hierarchical combination of smaller objects to larger composite objects, is a central technique for the construction of object-oriented software systems, and for the management of the system's structural and dynamic complexity. The proposed system keeps the structure of the system (into composite objects) decoupled from the structure of the program (into packages), which are two orthogonal notions [OMG00]. The path-based approach is compatible with object-oriented design's step-wise derivation of high-level object (composition) links from paths of lower-level "manifest" links (i.e., object references).

As a by-product of concretizing the notion of state encapsulation for composite objects, a clarification of the relation between state and object composition is obtained: to object-oriented programming Composite objects have component objects that represent aspects of the composite's state. But they can also have temporary components merely for the implementation of its behavior. For example, an iterator object is a component of the client object that represents (the state of) the client's iteration process—for as long as it lasts. If iterators were considered components of the container object which created them (as in [Cla01]), operations to create and return an iterator would change in the container's composition and thus be mutators.

1.3 Outline

The remainder of the dissertation is structured as follows:

The next three chapters introduce the context of this work regarding object-oriented systems, encapsulation, and other research. Chapter two introduces the reader to the abstraction concepts on which object-oriented programming is based, focusing in particular on the object-oriented view of a running software system, on the dual data- & behavior-nature of objects, references, and object composition, and on the notion of composite objects. Chapter three exlains the importance of the modularity of programs and runtime systems, and its relationship with encapsulation and alias and access control. And it discusses different proposals w.r.t. how encapsulation barriers should be drawn and what encapsulation property should be enforced. Chapter four reviews previous work on systems for composite object encapsulation.

Chapters five and six contain the definition and formal treatment of JaM. As a first step, chapter five considers the addition of a reduced mode system to a Java subset (base-JaM). Its definitions and results are extended in chapter six to a JaM with the full system of modes.

In chapter seven, the relation between modes and types, and the consequences of the mode system for reference and message flow are considered. Some obvious extensions of the formalized JaM language and mode system are discussed, and more examples are provided.

Chapter eight concludes the dissertation with a look back on what was achieved.

The two appendices sum up the formal definition of JaM, and provide the full JaM code of the running example of composite map objects and their iterators.

Chapter 2

Abstraction in Object-Oriented Programming

Does it not require some pains and skill to form the general idea of a triangle, ...
for it must be neither Oblique, nor Rectangle, neither Equilateral, Equicural, nor Scalenon;
but all and none of these at once. In effect it is something imperfect, that cannot exist;
an Idea wherein some parts of several different and inconsistent Ideas are put together.
John Locke (1632-1704)

This chapter sets the background for this dissertation: the composite object abstraction in object-oriented programming. It may be skipped by readers already familiar with the object abstraction in general, with the object-oriented runtime system model, and with dynamic composite objects.

We will review the central abstraction concepts of object-oriented programming (object, class, subclassing), and the object-oriented view of the runtime system as a network of interacting objects. The generalization from elementary objects to composite objects with objects as components will be used for structuring the system into a hierarchy of nested objects. The foundational data/behavior dualism of (composite) objects, object references, and object composition will explain why objects have more object references than those in their fields (namely temporary references in their methods), and how iterators can be components of their clients (temporary behavioral components) although they do not represent their state.

2.1 The Importance of Abstraction

The stuff from which software systems are made is not physical, but abstract (or conceptual). An abstraction (or concept) is "created" by the process of abstraction, i.e., by focusing on certain aspects, the *essentials*, while ignoring others, the *details* [LM88]. The ability to abstract enables us to work with complex domains of interest, like software systems and their application domains. Note that ignoring details does not remove them from the domain but only from our view (or "model") of it.

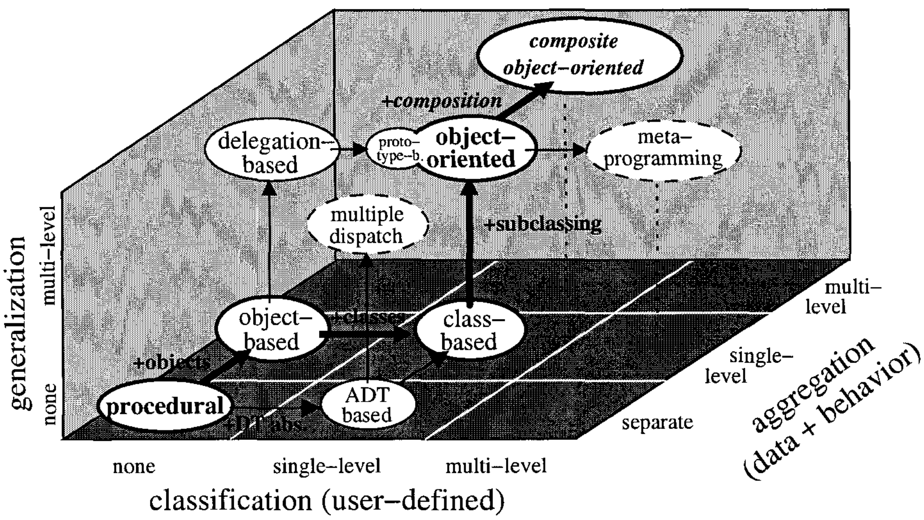


Figure 2.1: Space of programming paradigms

Complexity is intrinsic to software systems and cannot be made to disappear; it can only be managed by structure and abstraction: Industrial-strength software is inherently complex [Boo94]. Software systems like, e.g., SABRE and NORAD are among the most complex artefacts of humankind [Som95]. As Brooks so famously observed [Bro87], this complexity inheres in the problem to be solved (*essential complexity*), so that it cannot be avoided. We have to cope with it, manage it. And abstraction is our best hope for this. (Only the *accidental complexity* of software projects, which results from the technical platform, the development environment, or the organization of the development process, can ever really be removed.)

It should be mentioned that while abstraction is frequently used in programming, the overall process of software development resembles more abstraction's inverse, *concretization*: An initial, unspecific model is refined upon by filling-in what precisely is required (analysis), how to solve the requirements in the abstract (design), and how to make a computer actually carry out that solution for us (implementation).

In programmingm, different so-called "paradigms" can be distinguished by the abstraction concepts which are central to them. The most common kind of abstraction before the identification of the *data abstraction* in the 1970s was the *functional* or *procedural abstraction* [LZ75]. It characterizes traditional, "procedural" programming. The object-oriented paradigm of programming distinguishes itself by the three new abstraction concepts of *object*, *class*, and *subclassing*[1] [Weg90, Sny93, Qui95]. These go one step into each of the three directions of abstraction (cf. fig. 2.1):

1. AGGREGATION. One function of abstraction is to allow us to treat several entities ('parts', 'components', 'constituents') as one by ignoring the distinction between them

[1]Some put the emphasis on subclass polymorphism, others on inheritance (cf. paragraph *3c*).

and subsuming them under one entity ('whole', 'composite'). For example, we can say "the triumvirate ruled Rome from 60 to 49 B.C.," and ignore the distinction between Julius Caesar, Crassus, and Pompeius. In procedural programming, several, more primitive computational steps are combined into one by *procedural abstraction*, and several pieces of data are combined into one compound by *structured datatypes*.

In "object-based" programming, the *object abstraction* overcomes the traditional operation/operand dichotomy of procedures and data in procedural programming by integrating mutable data in form of *fields* (also called "instance variables," "attributes," "data members" or "slots"), and behavior in form of *methods* (also called "operations") into one runtime unit, the object, by ***object abstraction***. Objects are the elementary subsystems of the object-oriented runtime system model described in the next section. They are a universal modeling concept which can reify in the runtime system not only data (with operations on it) but also active agents [Bi+80], control structures [GR83], iteration processes [Ga+95], functions [ISO98], etc.

2. CLASSIFICATION. Another function is to subsume all entities sharing certain selected properties under one 'class' (or 'type', 'kind'), so that they can be treated uniformly: "Types arise informally in any domain to categorize objects according to their usage and behavior" [CW85]. For example, we can investigate the properties of *all* systems with a finite number of states (finite automata) and make laws for *all* people. In programming, the classification of values into types enables us to write algorithms that work with any value of a certain type.

"Class-based" programming extends object-based programming by ***class abstraction***, through which all objects with the same kinds of fields and methods can be collected in an object class [Boo94]. Class abstraction reduces the multitude of objects in the system to a fixed number of classes, the system's *class model.* A class *definition* defines a class of objects by aggregating definitions of their instances' fields and methods; class definitions are the modules of object-oriented programs.

3. GENERALIZATION. Abstraction allows one to subsume all special classes ('subclasses') defined by a common subset of properties under one common, more general class ('superclass'). For example, we can generalize people and corporations to legal entities (and have the same laws for all of them). We can treat as irrelevant the difference. The classical way of *defining* a new subclass, 'species', is to name its superclass, 'genus', and the difference from it [RC00].

Object-oriented programming is only complete with superclass abstraction, better known as ***subclassing***. It allows one to structures the class model as a *class hierarchy* (see paragraph *1a* below), to write reusable client code that works with objects from all subclasses of a class by ignoring objects' precise classes (*subclass polymorphism*, a form of subtyping), and to reuse the definition of one class for the definition of a subclass of it by naming it and then specifying the difference (*class inheritance*).

(Also based on the object abstraction is "delegation-based programming:" It adds inheritance between child and parent *objects* by the mechanism of delegation [US87].

It becomes nearly equivalent to object-oriented programming by the addition of distinguished, class-like '*trait* parent objects' shared among all clones of an object in "prototype-based programming" [Ast96].)

2.2 Abstraction Hierarchies

The recursive application of the abstraction process can lead to higher and higher abstractions in all three directions: Hierarchical aggregation (part-whole hierarchies, partonomies, or *has-a*-relationships) and hierarchical generalization (inclusion hierarchies, taxonomies, or *is-a* relationships) are the classical tools for our understanding and description of the world, in use for at least since Aristotle over two thousand years ago [RC00]. (The idea that classes can also be classified, however, is just over a hundred years old, starting with Peano et al.'s observation that class-membership '$\in$' and class-inclusion '$\subset$' are two distinct relations [LL97], and Frege's insight that classes are abstract objects in their own right and can be classified [Par94]. The unconstrained classification of classes was soon thereafter discovered to lead to Russell's Paradox, a fundamental logical paradox tamed by Russell's *theory of types*, the predecessor of type systems in programming languages.)

In object-oriented programming languages, the characteristic abstractions *class* and *object* are just single-level, while *subclassing* applies recursively. The object composition hierarchy is one of several proposed hierarchies promising still better complexity management. However, different hierarchies seem to co-exist well only if they bring order to orthogonal architectural perspectives (cf. [SNH95]). While the class model is structured through subclassing and the program is structured into packages, object composition brings order to the object-oriented runtime system model. A second hierarchy in any of these perspectives seems to increase the overall complexity more than it helps managing it:

1. The Class Model: Conceptual Architecture. Object-oriented programming is often praised for organizing the system's set of object classes by subclassing into a conceptually clear generalization hierarchy called the ***class hierarchy***. (Procedural programming did not support this for its datatypes.)

Research however showed, first, that over-enthusiastic use of subclassing with class hierarchies deeper than three levels is detrimental for program maintainability [DB$^+$96]. Second, an inheritance-based subclass relationship does not necessarily mean a real specialization because method overriding is not guaranteed to specialize the object's behavior [Ame87, LW94, Tai96]. Third, inheritance-based subclassing is best formalized not by the type-theoretical concept of *subtyping* [Sny86, Lis88, CHC90], but by "F-bounded polymorphism" [Ca$^+$89, CHC90] since at runtime a class is relevant only as a *generator* of objects [SM95].

Most typed object-oriented programming languages restrict inheritance to conform to subtyping. Proposals to work with two separate hierarchies [Bru96, BPF97,

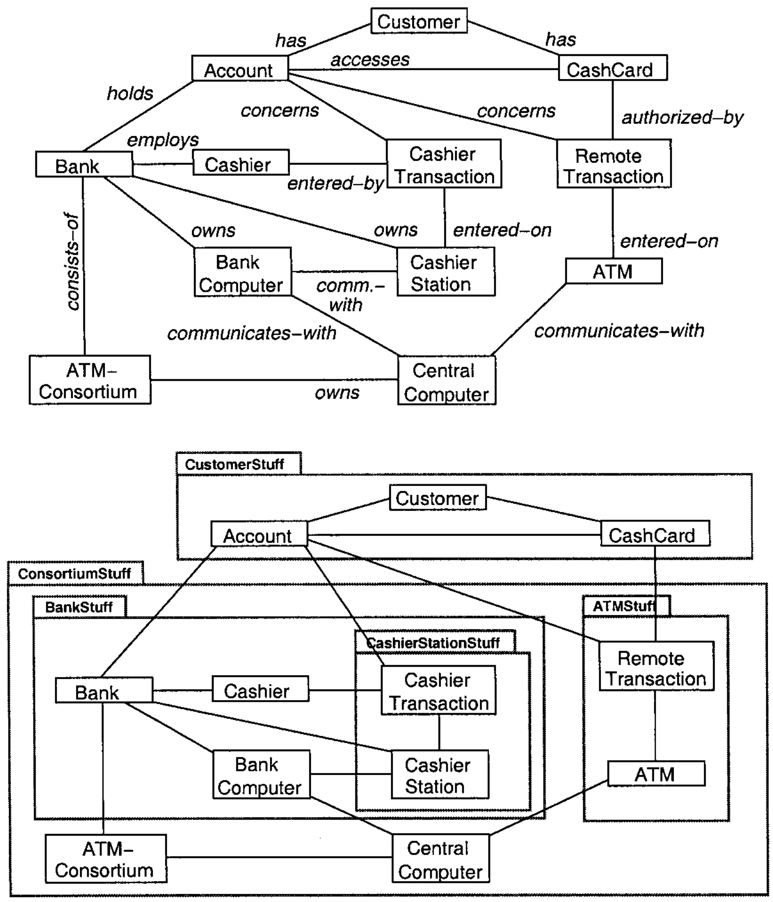


Figure 2.2: Flat, and structured class model of an ATM-banking system

GM97] were not widely accepted. Also the higher-order classification of object classes into *meta-classes* has not found wide use as a programming technique since classes are already sort-of classified by their superclasses [Weg90]. (So-called "meta-classes" in Smalltalk, CLOS, Java, etc. [GR83, Kol99, GJS00] are normal classes of objects that reify a class at runtime for administrative purposes like constructors, `static` members, reflection, ...)

2. THE PROGRAM: MODULE ARCHITECTURE. More helpful is a hierarchy for the *definitions* of the classes in the orthogonal *module architecture* of the program: The aggregation of field and method definitions in class modules is extended to a hierarchical aggregation of smaller class modules into enclosing class modules and of class modules in ***packages***. The introduction of hierachical packaging in Java [GJS00] was so successful because it was already practiced informally by sorting program files into different file system directories and because the notion of a non-class module was known from procedural languages like Euclid, Modula, and Ada [La$^+$77, Wir83, ISO95]. Packages can be used to group classes, e.g., by application domain for retrieval from a library, by vendor for controlling name clashes, as the unit of purchase and revision, and simply to manage the complexity of large programs with hundreds of classes.

For example, the first UML model of a banking system in figure 2.2, with an ATM-consortium, banks, accounts, cashiers, cash-cards, ATM's, and so on (adapted from [Ru$^+$91]), appears "confusing and disorganized" [Kri94]: "The problem is that this kind of description does not reflect the way that we think about and understand such complex systems." The second UML model in figure 2.2 cleans up the class model by dividing classes between those modeling the customers and their property (Customer, CashCards and Accounts) in the CustomerStuff package, and the rest in the ConsortiumStuff package with sub-packages for, respectively, bank-related and ATM-related classes. Complexity management is improved through the possibility of zooming into and out of packages to view the system at different levels of detail.

3. THE RUNTIME MODEL: SYSTEM ARCHITECTURE. Finally, the higher-order extension of the aggregation of fields and methods in objects pervades object-oriented programming—although this is often ignored since it is a matter of object-oriented *design* of the system at runtime, and not explicit in the program text [Ga$^+$95]: The objects in the object-oriented view of the runtime system are aggregated to *linked object structures*, to groups of collaborating objects (*collaborations*), to *composite objects*, etc. In particular, the recursive composition of objects to composite objects produces the system's ***object hierarchy*** (object composition hierarchy).

It is important to get order into the object-oriented runtime system model: Class models of large systems, with hundreds of classes connected by hundreds of relationships, may be complex. More complex still are the corresponding runtime models with an *even larger* and *dynamically changing* number of objects and connections. To cope with the structural and dynamic complexity of the runtime model, object aggregations are naturally used. Providing for their expression in the program would complete the support of object-oriented programming *languages* for the main complexity management techniques of object-oriented *programming.*

All this will be elaborated in this chapter. But first we have to develop an understanding for the object-oriented view of the runtime system.

2.3 Object-Oriented View of the Runtime System

A feature of object-oriented programming (OOP) more fundamental than the static classes (OOP is class-based) are the data and behavior combining units of the runtime system called objects (OOP is object-based). The view of the runtime system as a system of message-exchanging objects distinguishes object-oriented programming from procedural programming more than anything else, and is the common basis of all object-based programming paradigms (class-based object-oriented programming as well as delegation- and prototype-based programming). (The *programs* in object-oriented and procedural programming have the same basic linguistic structure,with modules containing the definitions of related variables and subroutines.) This view possesses a higher degree of uniformity achieved by the dual nature of objects and of object references as providing data as well as behavior. The object-oriented view is considerably more different from how real computers are organized than the procedural view. (A straight-forward execution of object-oriented programs on computers requires one to follow certain constraints on the language design, which have developed into "myths" about object-oriented programming [Rum97].)

1. PROCEDURAL SYSTEMS: DICHOTOMIC ARCHITECTURE. In procedural programming, the runtime system is divided like a virtual computer into active operators in a *program* compartment (the processing unit), and passive operands in a *storage* compartment (the memory unit) [Qui95]. Consequently, program and data are classified and composed separately to procedure types and "procedural abstractions" on one side, and to concrete data types and data structures on the other.

Computation is understood to take place in the procedures (subroutines) within the program's different modules. While some data is in the module's variables, more data can be represented in linked data structures constructed dynamically in the storage compartment.

2. OBJECT SYSTEMS: HOMOGENEOUS ARCHITECTURE. Object-oriented programming overcomes the procedural operator/operand dichotomy by grouping and classifying related data and operations together as objects and object classes [Qui95]. In the small, each object is a tiny procedural system of its own, with its own internal program compartment and storage compartment [Bud95] (which is conceptually concurrent [Rum94c]), while in the large the runtime system is "structured uniformly as a collection of interacting objects" [FM90] connected by object references to a uniform "network architecture" [SG96].

Computation takes place in and between objects, not modules: It is understood to be carried out by the objects internally as the manipulation of their variables and object references (computation in the small), and externally by message exchange along object references and the creation of new objects (computation in the large). In the software architecture, the objects are the architectural components (active computational agents) and the architectural connectors (interaction channels) between them are the object references. This architecture is completely independent from the static structure of the program, but built up incrementally and reconstructed dynamically like a linked data structure by the exchange of object reference values. Since besides this there is no global, static program compartment, in the object-oriented view there is no connection at all any more between the structure of the program in form of modules and packages, and the structure of the runtime system in form of object references. Procedural programming's program/data dichotomy within the runtime system is traded in object-oriented programming for a program/system dichotomy.

3. THE DUAL NATURE OF OBJECTS. The object in the sense of object-oriented programming is an abstraction that combines *data* and *behavior* in one identifiable unit. Since the runtime system in the object-oriented view consists of objects, the system's *state* as well as its *processes* must be partitioned among these objects.

In object-oriented programming, each object owns a chunk of the system's ***global state*** (interprocedurally persisting state), "the" state of the object, to which its methods have shared access and which persists between method executions [Weg90]. Hence objects may be regarded as "functions with memory" [Mez98] that can remember something from previous times they executed a method. Objects support ***data abstraction***, not by data type abstraction as in ADT-based programming, but by representing the abstract data (a calendar date, a tree, a set, ...) in one or more objects' state and providing an operation interface through which the outside accesses it in an abstract faction: They are "procedural data structures" [Rey94]. Data-representing objects are "active data" [Mez98] or "intelligent data objects" [ASS96] to which operations are not applied but that offer to perform these operations on themselves, i.e., on the data: "Ask not what you can do *to* your data structures, but ask what your data structures can do *for* you" [Bud95].

But this is not the complete picture. The behavioral side of objects entails that they have a share in the ***local state*** of the system's processes (transient intraprocedural state), in particular, the values of local variables and already evaluated subexpressions in the methods which the object is currently executing. It may be safe to ignore this as long as an object operates only on its own variables (computation in the small). But not all objects can be data, there must also be the objects communicating with them and each other (computation in the large). There is more to objects than intelligent *data*; they are also communicating *processes*. As such they can reify ***behavioral abstractions*** like Iterators, Commands, Strategies, and Mediators [Ga+95]. For example, an Iterator object represents—with its state and its method executions—the state and the steps of an iteration process (that runs in parallel to the client's method like a coroutine).

Even where it concerns data, communication may have to be used to implement abstract data structures if they contain an *unbounded* amount of information, or to construct linked data structures with an *unbounded* degree of branching: The global state which the programming language's implementation objects have at their disposal is limited to a *fixed* number of variables called fields. Hence the mentioned data abstractions can be implemented only by a collaboration of several implementation objects (in a composite object), i.e., if objects communicate.

4. THE DUAL NATURE OF OBJECT REFERENCES. Each object in the system is identified by a unique *object identifier* $o \in \mathbb{O}$ (assigned to it when it is created). If an object (identified by) o has among the values in its fields or methods the identifier $\omega \in \mathbb{O}$ of another object, then o is said to have, at that moment, an ***object reference*** (link, handle, pointer) to ω, in symbols, $o \rightarrow \omega$. In this reference, o is called the *source* and ω the *target*.

The object references in an object system have a dual function: On the behavioral side, they are the ***architectural connectors*** that enable computation in the large by transporting messages between objects: requests for method executions (operation invocations), and replies of the result. The references an object has at a moment determine to which other objects it can send requests at that moment.

On the data side, object references are ***values*** that can be exchanged between objects as parameters and results, and stored in variables. (These exchanges and the loss of references by variable update are what changes the system architecture dynamically.) The references an object possesses at a moment define the set of object reference values it can avail for passing as parameter and result values (since an object reference cannot be calculated from another value[2]).

Object references can be used as connectors and values irrespective of whether they are stored in any variable: Consider the calls `n.prev().SetNext(n.next())` and `n.next().SetPrev(n.prev())` to unchain the node (identified by) `n` from a double linked list. Here, temporary object references returned from calls `n.prev()` and `n.next()` serve as parameter values and as connectors to the nodes, respectively, in front of `n`, and behind of `n`. But to represent linked data structures and the storage of objects therein, the object references between the objects must be *captured in fields*. References in fields represent *data structure links*, like those between two node objects, and stored *data values*, like a pair object's first and second value.

The notion of ***object graph*** in object-oriented programming is a generalization of the classical notion of *data structure graph* in procedural programming. It is the directed graph made of *all* the objects currently in the system as the nodes, and *all* the object references between them as the edges, whether they are used as connectors or as data, whether they are in fields or in methods. It uniformly captures the structure of all the objects' interconnections at a particular moment, thus integrating both the system's architecture and all the data structures in it. *All objects are connected* in the object graph (of a sequential program), since objects to which there is no path of object references from the initial object are unreachable for the computation and thus can be "garbage collected."

2.4 Complexity in the Large in Object Systems

Object-oriented programming supports well the management of runtime complexity in-the-small by grouping operations and their common data into one object. But its uniform, unstructured network architecture does not help with the complexity in-the-large that results from the many objects around and all the interactions and semantic relationships between them. In analogy to unstructured "spaghetti code," this was dubbed "object spaghetti" [PNC98]. For an impression, look at the banking system in figure 2.3 with a mere two bank objects, two customer objects and three account objects (more on this below). "The traditional 'sea of objects' approach where all objects in the system are visible to each other and exist at the same level is infeasible"

[2]The reference arithmetics of C++ is a much criticized exception.

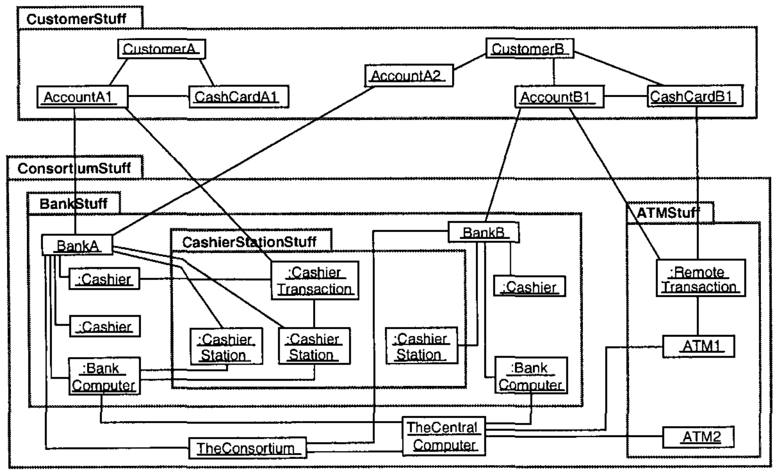


Figure 2.3: Managing the complexity of the object system through packages

[Bos96]. We need a view of the runtime system that is *structured*, that groups objects to larger units so that we can view the system at intermediate levels of detail.

Some degree of structuring is achieved by projecting the packaging of classes (cf. fig. 2.2) onto their current instances, as figure 2.3 shows. But while it reduces the number of constituents in the higher-level view, it is inadequate for managing the *dynamic* complexity of the system. Since objects are grouped together independently from their interaction, the resulting structural units have poor cohesion w.r.t. the system's working. For example, instances from BankStuff classes have to do with other BankStuff objects only if they belong to the same bank, and have more to do with that bank's customers and the central consortium objects than with any BankStuff object of a different bank.

A viable method for coping with large object systems must not squeeze a dynamic number of objects into a static structure, but provide for the genuine aggregation of objects to a dynamic number of larger units. A variety of different kinds of such aggregations have been described: Traditional linked data structures as *object structures*; *collaborations* for the modeling of system dynamics [HHG90, KM96, St$^+$96, OMG00]; *runtime components* made of interface objects and internal objects [MP99a]; *Clarke's aggregates* of all objects with the same "representation context" and the objects allocated in that context [Cla01]; sets of all objects reachable from a particular object by paths of object references in fields (*islands* [Hog91], *balloons* [Alm97]); sets of objects reachable from the object graph's root *only* by paths of references passing through a given object (*umbra*) [PNC98]; and so on.

But the most important object aggregation of all is the composite object.

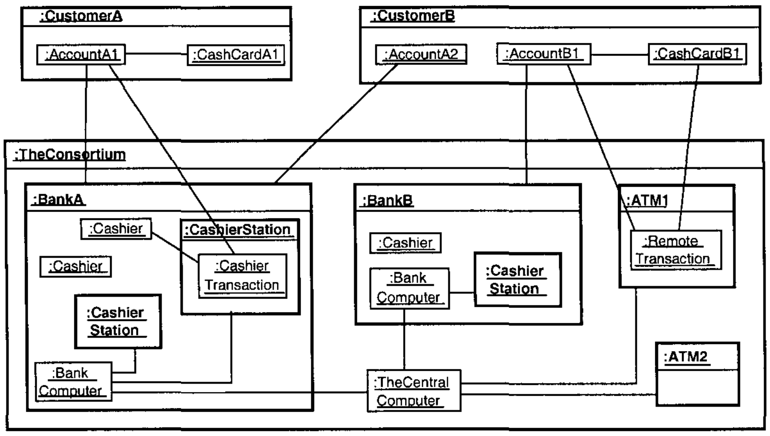


Figure 2.4: Managing runtime complexity through composite objects

2.5 Composite Objects and Structured Systems

1. THE COMPOSITE OBJECT ABSTRACTION generalizes the (elementary) object abstraction, the aggregation of n fields (data primitives) and m methods (behavior primitives) to a data/behavior unit, to the aggregation of n fields, m methods *and* k objects (themselves units of data and behavior) to a more complex data/behavior unit called ***composite object***. The limit case of a composite object is one with zero components (*elementary object*).

In the banking example, composite objects provide not just one structural unit for all the bank (or ATM or customer) stuff, like packages did. As shown in figure 2.4, there is one unit for *each* bank's (and ATM's and customer's) stuff, namely the composite bank object (ATM object, customer object). The composite objects are not additional structural units, like the packages were, but extensions of existing elementary bank (ATM, customer) objects. The additional structure is achieved without additional "boxes" in the diagram. (Also some links between objects do not show up any more because they are now implicit in the nesting of objects [Kri94].)

In a *composite object*-oriented view of the runtime system, composite objects take the place of elementary objects in objects structures, in collaborations, in object references, etc. Object references may connect any top-level or (nested) component object with any other one. Even without an explicit object reference, the composite object from within its methods can directly send invocation messages to its direct component objects. A new type of event possible in object systems structured into a hierarchy of composite objects is the change of this structure: An object can become a particular object's component or cease to be its component [OMG00]; in other words, it can "migrate" from one composite object to another.

2. THE IMPORTANCE OF COMPOSITE OBJECTS. The composite object abstraction is *scalable* from the elementary object up to the entire object system as one allencompassing composite object [Rum94c, Bos96]; it structures the entire object system into one object composition hierarchy without conflict-bearing overlaps. A structuring into composite objects acknowledges that certain groups of objects are tightly coupled and have themselves object-like properties, which is necessary for "a viable method for the characterization of large systems" [Cha91]. Composite objects are the units as which the specification of the object system is "structured naturally," and which guide the reasoning process "in a natural fashion" so that it is local to the composite in many cases [GM93]. Object composition is an important semantic relation that provides a back-bone for *message forwarding* [Cha91, MZ92, GM93, MC94, HG97], *property inheritance* [GL95, OM01] and *refactoring* [JO93].

Structuring an elementary object system into composite objects does not introduce additional structural units, but extends existing ones. The composite object abstraction is not a completely new concept to learn for the programmer, but just a generalization of the elementary object abstraction. Moreover, the notion of composite object is already known from object-oriented design, and implicit in top-down refinement of higher-level objects to lower-level objects and in the *object composition* technique of object-oriented software construction (cf. §2.7). The kind of generalization by object composition, from a shallow notion of object to a nested one, is known from subclassing, which generalizes a shallow notion of class to a transitive one that includes subclass instances. The same way we can resort, where necessary, to the original, shallow notion of class by talking just of its *direct* instances, we can resort to the original, shallow notion of object by talking about the composite's *representative* in paragraph *4* below.

The quality of object aggregation techniques can be judged like the grouping of definitions to program modules: It should produce units of high internal cohesion and with low external coupling to be really useful for the management of complexity. Composite objects have higher cohesion than other object aggregation techniques. First, the constituents collectively represent one higher level abstraction (abstract data structure, behavioral abstraction, etc.), and thus are held together by *conceptual cohesion*. Second, the constituents coordinate their behavior to this end, and thus are held together by *dynamic cohesion*, like in a collaboration. ("A composite object is similar to ... a collaboration, but it is defined completely ... in a static model" [OMG00], namely the class model.) Additionally, a core of state-representing subobjects must be permanently connected in order to implement the representation of the abstraction's state and thus are held together by *structural cohesion* like in an object structure. The external coupling of composite objects can be reduced by techniques of encapsulation discussed in the next chapter.

3. COMPOSITE CLASSES. Composite objects are instances of an object class, which in this case is called a ***composite class***. The definition of a composite class fixes its composite instances' fields and methods, their possible component objects and the possible processes of their dynamic (re)construction. In the example, the structuring of the banking system into composite objects from further above can be distilled by

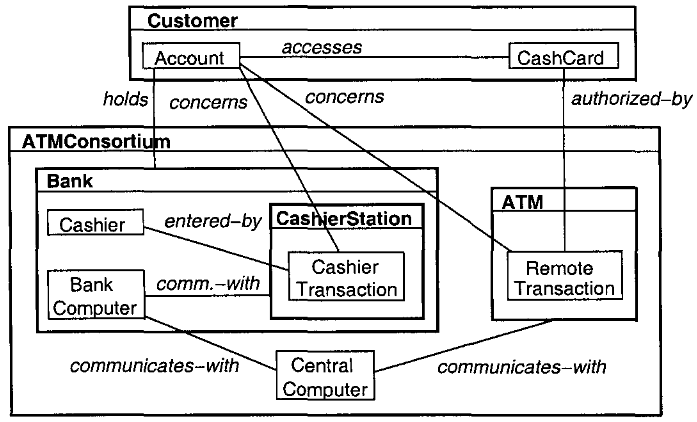


Figure 2.5: Class model of composite object-structured system (adapted from [Kri94])

class abstraction to the class model shown in figure 2.5.

Composite classes have nothing to do with packages [OMG00]: The nesting of (composite) classes in the UML diagram captures the nesting of their instances at runtime, not the nesting of their definition modules in the program (for which package-combinator ⊕— is used in UML). In object-oriented programming, runtime structure is orthogonal to program structure (§2.3). It is natural to let any composite class use any class, no matter the package, as the type of its instances' components. Independently from object composition, we can achieve a cleaner organization of the program into packages in which all classes can be reused.

For example, figure 2.6 shows how the classes of the map example could be sorted into four general packages: At the bottom is the package DSComponents of standard data structure components, like Node and Pair, that have many different uses. Package DSIterators contains the corresponding iterator implementations. The collection implementations that build on these two packages are collected in package DSCollectionImps. At the top is the package of the high-level collection and iterator types, for which the other packages constitute one possible implementation.

4. EXPANSION TO IMPLEMENTATION OBJECTS. Current object-oriented programming languages support only the elementary object view (§2.3). But object composition can be "simulated" [HJS92] by expanding each composite object to an aggregation of elementary implementation objects. The example of a composite Car object car with Engine and Wheel components e and w is shown in figure 2.7. First, the composite's component objects are expanded recursively. Second, the rest of the composite, namely its identity, fields and methods, is combined by elementary object abstraction to a separate implementation object called the ***representative***. Third, the composition relationships between composite and components can be represented by "composition references" between representative and component objects' representatives to explain the messages exchange between them.

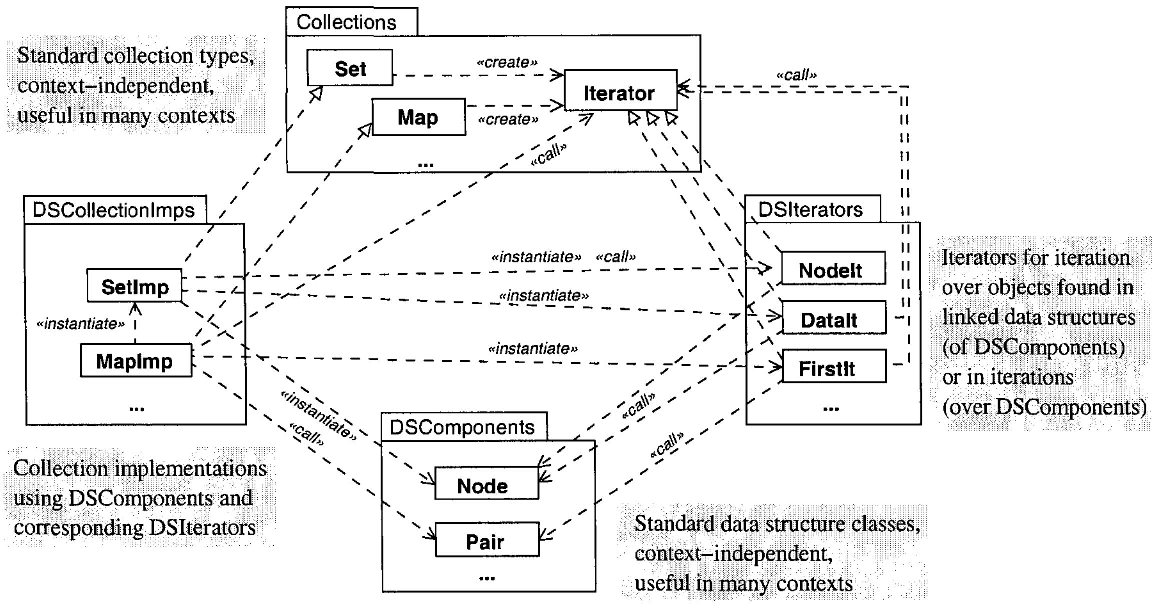


Figure 2.6: Class packaging orthogonal to object composition, with dependencies

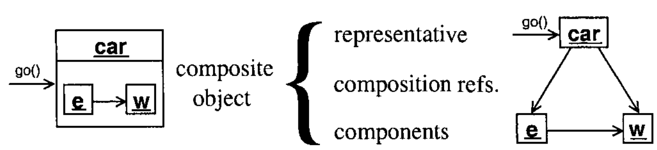


Figure 2.7: Composite object and expansion to elementary objects

In the implementation object view, composite objects are aggregations of implementation objects that function together like one large, complex object. Whereas a composite's components may come and go, its representative remains the same through the composite's existence. Since the representative is unique, the composite can be uniquely identified by identifying the representative. The representative "represents" the composite "as a whole," i.e., modulo the component objects, takes its role as source and target of references and of communication.

The representative is also called the "dominant object" of the composite's expansion [Rum95] or of a corresponding "high-level object (class)" [EKW92], and called the "root instance" of the expansion as a subsystem of the runtime system [BLM97].

5. The Dual Nature of Object Composition. In object composition, more complex, composite objects are constructed from simpler component objects, by giving their union a separate identity with fields and methods independent from the components, in other words, by unifying them under the representative. The data/behavior-dualism of objects in general, and *composite* objects in particular, entails that a component object (with data and behavior aspects) can serve the purpose of implementing the composite's *data aspects* (static properties) as well as implementing the composite's *behavior aspects* (dynamic properties): "Objects obtain their static and dynamic properties by composing, delegating, inheriting, and coordinating those of other objects" [CLF92]. Consequently, among a composite object's component objects one can distinguish the data- or *state-representing* components from the *behavioral* components:

Since objects have state and composite objects o are objects, they must have a state, written $\mathit{CState}(o)$. In this state, o can represent abstract data to implement a data abstraction. To $\mathit{CState}(o)$ belong the states $\mathit{state}(\ell)$ of the composite's fields $\ell \in \mathit{flds}(o)$ as well as the states $\mathit{CState}(\omega)$ of certain components ω of the composite which are accordingly called its ***state-representing components*** $\omega \in \mathit{StCmp}(o)$. In short, the composite object o's current state $\mathit{CState}(o)$ is some kind of union of its fields' and current state-representing components' states:

$$\mathit{CState}(o) \;=\; \bigcup_{\ell \in \mathit{flds}(o)} \mathit{state}(\ell) \;\cup \bigcup_{\omega \in \mathit{StCmp}(o)} \mathit{CState}(\omega)$$

It is crucial for the power of *composite* objects (over elementary objects) and of the object composition technique (over inheritance) that in $\mathit{CState}(o)$ not only each field and component's state $\mathit{state}(\ell)$ and $\mathit{CState}(\omega)$ can change, but that also the set $\mathit{StCmp}(o)$ of state-representing components is able to change dynamically as needed (unlike the set $\mathit{flds}(o)$ of fields). While the former is a "quantitative change" within the state space spanned by the sub-objects' fields, the latter is a "qualitative change" that changes the spanned state space [Bun79]. For example, an implementation `MapImp` of an abstract `Map` data structure, i.e., a variable mapping from key objects to value objects, may represent `Map` states by storing each key:value pair of the map in a variable number of component objects of class `Pair`.

But there is more to object composition. A composite object can also have component objects *not* for representing its state but just for the implementation of a behavioral aspect. The composite state of such ***behavioral components*** does not contribute to the composite's state, so that behavioral components' mutations do not count as changes of the composite. Often a behavioral component exists only while the composite is executing a method.

For example, consider the implementation of the `lookup` operation on the abstract `Map` data structure that will be elaborated in detail in the next section. For `lookup`, a `MapImp` composite d has to iterate over its entry components of class Pair in search for a given (potential) key object. It can chose to represent this iteration by a behavioral Iterator component i, a behavioral abstraction which provides for iteration operations and represents the iteration's state. Iterator i must be viewed as a component of d since the meaning of the `lookup` operation does not allow for sending (state changing) messages to external objects. And i cannot be a *state-representing* component of d since the meaning of the `lookup` operation allows no change of d's state $\mathit{CState}(d)$, whereas Iterator i must change to progress the iteration during `lookup`. Hence i can only be a *behavioral* component of d.

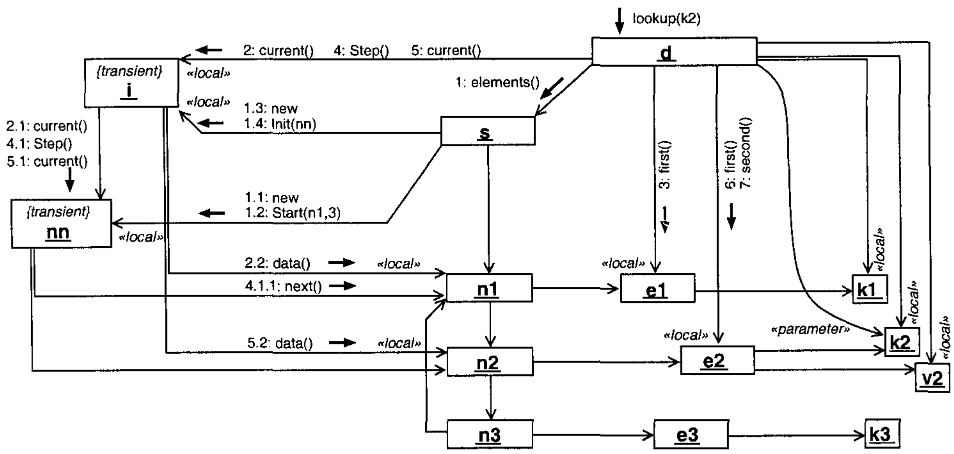


Figure 2.8: Unstructured lookup collaboration

2.6 Managing Dynamic Complexity: The Map Example

The bank example is too large for getting to the bottom of it. A standard example in the field of composite object encapsulation are *container objects* which represent application-independent abstract data structures also called collections. Kent and Maung started the tradition with stacks represented by a linked list of nodes [KM95]. Noble, Vitek, and Potter continued with a hash-table associative containers represented by entries stored in an array object [NVP98]. Both groups pointed out the difference between the objects constituting the container (the stack's or hash-table's representation) and the objects contituting the container's content (the stack's elements or hash-table's arguments).

The example that will accompany us throughout this dissertation is a particular implementation of maps, where the entry pairs are stored in a set represented by linked nodes. A map is an associative container object in which "key objects" and "item" or "value objects" are stored so that each key object is uniquely related with a value object. Even such a relatively simple thing like a map provides us with an example of dynamic complexity if we view it at the lowest object level.

Consider how a request for looking up a key is served: The UML collaboration diagram in figure 2.8 shows the particular interaction betweens eight elementary objects (plus six passive objects) through which a particular lookup in a map with a particular content is implemented. In this unstructured form, it is rather difficult to see how it works. It is natural to parse it first, to start the understanding (or the description) by identifying which objects belong together, and which type of object they are *as a unit*, i.e., as one composite object, as figure 2.9 shows it.

1. THE PARTICIPANTS. On the state side, the Node objects n1, n2, and n3 form a *ring* structure in which the objects e1, e2, and e1 are stored. The nodes belong to s, an object of implementation class SetImp, i.e., are its components. s, in conjunction with its components, i.e., as one composite object, is the software realization of a Set, namely the set $S = \{e1, e2, e3\}$. Objects e1 through e3 are Pair objects representing three map-entries "k1: v1," "k2: v2," and "k3: v3" (v1 and v3 are not shown in fig. 2.9). Composite object s (the entry-set) and objects e1, e2, and e3 (the entry-objects) represent what the map's current content is. Hence they are the *state-representing* components of the composite MapImp object d (not shown as composite in fig. 2.9) which is a software realization of a Map with the aforementioned three entries.

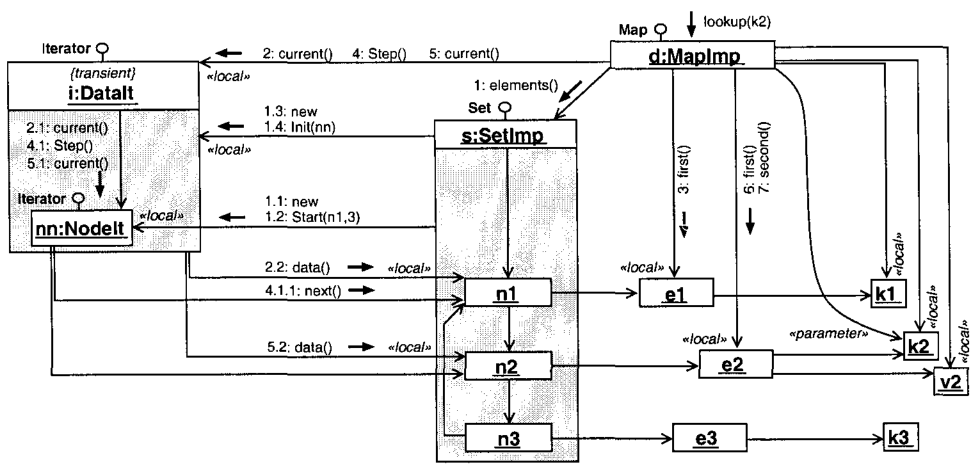
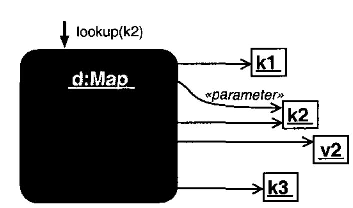


Figure 2.9: lookup collaboration structured with composite objects

On the behavior side, nn is an object of implementation class NodeIt that realizes an Iterator object reifying the iteration n1, n2, n3 over the nodes. It is a component of the DataIt object i. Together, i.e., as one composite object, both realize an Iterator object, i.e., the reification of an iteration e1, e2, e3 over set s's elements. It represents the map's search k1: v1, k2: v2, k3: v3 through the entries for the given key. Composite iterator i is also a component of d, a *behavioral* component. MapImp object d, together with its behavioral component i and state-representing components s, e1, e2, and e3, i.e., as one composite object, realizes a Map that has three entries *and is in the process of looking up a given key.*



2. THE ACTIVITY. Note that all interaction takes place within this composite object. Hence if one takes the map composite as one and abstracts from its parts, i.e., if viewed from outside as black box, then a lookup in the map has minimal complexity: There is only the request message lookup(k2) arriving at object d, and the reply message returning the result v2 (not shown). Nothing else happens. There are no observable intermediate interactions nor states during the lookup.

The complexity of what is going on internally during lookup can be split into two smaller portions along the boundaries of the composite components s and i: At the intermediate level of aggregation, s and i are viewed as black boxes of type Set and Iterator, respectively (see figure 2.10). This view works out the essence of MapImp's implementation of Map's lookup, which is independent from the realization of entry-set s and entry-iterator i. At the level below, we focus on the interactions inside of s and i, and between them, and ignore the context of a MapImp composite performing a map-lookup. This shows us the essence of how the iteration over the map's entries is implemented in the MapImp composite. That is, we see independently from the particulars of a map-lookup how iteration over a set's elements is implemented if that set is realized by a composite of implementation class SetImp.

3. INTERMEDIATE LEVEL: LOOKUP IN A MapImp MAP. When request lookup(k2) arrives at a map realized by MapImp composite d, this leads to the following sequence of events shown in figure 2.10:

1. d sends elements() to abstract Set object s to ask it for an iterator over its elements. s creates the new Iterator i (shown as pseudo-message new sent to i), initializes it in an unspecified way, and returns it to d.
2. d sends current() to its new, behavioral component i to ask it for the initial element in the iteration sequence. i communicates in an unspecified way with s to retrieve a first element e1 and return it to d.
3. s sends first() to entry object e1 to ask it for the key stored in it. e1 returns k1.
4. Since k1 is not the given key k2, s sends Step() to Iterator i to make it move on in the iteration sequence. i implements this by unspecified communication with s.
5. d sends again current() to i to ask it for the new current element in the iteration sequence. i communicates with s and returns e2.
6. s sends first() to entry object e2, which replies by returning k2.
7. Since this is the given key, s now sends second() to the same entry object e2 to ask it for the corresponding map-value stored in it. e1 returns v2, which d returns as the result of the lookup for k2.

4. LOWEST LEVEL: ITERATION OVER SetImp SET. Now consider how composite objects s and i implement steps 1, 2, 4, and 5 of the lookup collaboration by internal communication and communication with each other (see figure 2.9 again). Observe in particular how the references and communication between *abstract* objects s and i, that was not specified in detail in the intermediate-level view, is now implemented by low-level references and communication between different sub-objects of *composite* objects s and i.

When asked for an iterator over its elements (step 1), set s first creates the NodeIt iterator nn (1.1), and sets it up for iteration over its three storage nodes by initializing it with the call Start(n1,3) (1.2). s then wraps nn in a newly created the DataIt iterator i (1.3) by the call Wrap(nn) (1.4).

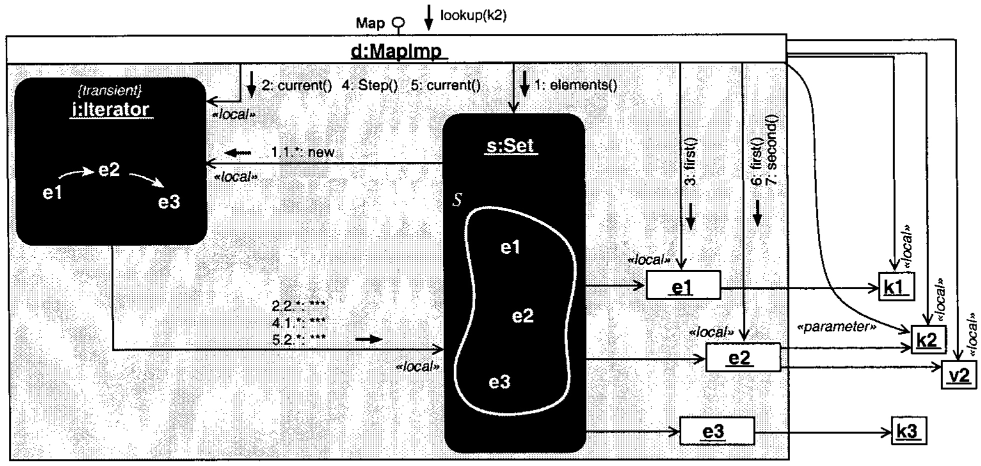


Figure 2.10: lookup collaboration at intermediate level of detail

A composite iterator i that was set up this way will, when asked for the current element, return the data in nn's current node (i.e., an element of abstract Set s), and will, when asked to make an iteration step, advance nn to the next node in the ring (internal to SetImp composite s). When asked the first time for the set-iteration's current element (2), i asks nn for its current element (2.1), which is the node n1 with which it was initialized. The returned n1 is then asked by i for its data (2.2), which is e1. The answer, e1, is returned by i as the first element of the iteration over s. A request to make an iteration step (4) is forwarded by i to nn (4.1). nn asks its current node n1 for the next node (4.1.1). The answer, node n2, becomes nn's new current node. When i is now asked again for the current element of the set-iteration (5), it forwards this request to nn (5.1), which answers with n2. This object is then asked by i for its data (5.2). The answer, e2, is returned by i.

2.7 Origin of the Notion of Composite Object

The notion of a recursive aggregation of (elementary, low-level, concrete) objects to (composite, high-level, abstract) objects, called *object composition* (or *object containment* [Lif93, Kri94, DD95a]), has four sources:

1. Modeling Parthood Relationships in the Domain. All general techniques for modeling real-world domains support *parthood relationships* as a distinguished kind of semantic relationship between two domain objects, the part and the whole (object-oriented models [Ru+91, Boo94, Ja+94, Hen97, OMG00], information models [KR94, Kol99], data models [SS77, Ki+87], semantic networks [JHC84], description logics [Ar+96]). If software object ω reifies the part and o the whole, then the object-oriented programming systems LOOPS [SB85] and ThingLab [BC87] in the knowledge representation field, and the object-oriented database ORION [Ki+87] represented the parthood relationship by a reference to ω in a special "part" field of o. Subsequent research in databases focused on clarifying the issues of shared versus exclusive parts, attribute propagation, existential dependency, constraint propagation, and local referential integrity (e.g., [MSI90, Liu92, HGP92, KS92]). A cognitive science paper [WH87] influenced a string of publications on the characterization of subkinds of the parthood relationship (e.g., integral whole/component, collection/member, mass/portion) in knowledge representation [ILE88, CH88, GP95], information modeling [KR94, Kol99], description logics [Sat95], and object-oriented modeling [Ode94, Hen97, SFL98, HB99b].

2. SOFTWARE CONSTRUCTION TECHNIQUE. Object composition has long been recognized as a central technique of object-oriented programming on a par with class inheritance [CLF92, Lif93, Ga$^+$95, MD95, Pre97]: Each design step can be regarded "as the implementation of some abstract object in terms of a collection of concrete ones that are "assembled" into a configuration that provides the functionality required by the abstract object" [FM90]. Particular cases are the component architecture COM with *inner objects* as components of *outer objects* [MD95], and "delegation-based systems" where a special form of composition between child and parent objects replaces class inheritance [US87]. Favoring object composition over inheritance [Ga$^+$95, Pre97] allows one to avoid excessive class hierarchies (§2.1) and instable base-classes (§3.1).

The external view of the composite object as a communicating process composed from component objects' behavior was formalized as *object embeddings* by Hartmann et al. [HJS92]. Gangopadhyay and Mitra defined objects at an abstract level first and then recursively refined them into composites with a compositional semantics abstracting from internal objects and communication [GM93]. Belkhouche and Wu modeled object composition in CSP by the parallel composition of the components' behaviors and the abstraction of internal communication [BW99].

The abstract state of objects as described in class specifications was formalized by Breu [Bre91] through a mapping from the collection of interconnected objects (*object environment*) representing it. The *refinement* of an *abstract object*, with an unbounded set of data components, to multiple *concrete objects* of the executable program, with a bounded number of fields was considered by Utting [Utt92]. How one object's state is represented in, or dependent upon, the fields of other objects (*components*) was formalized by Wills [Wil92] and, independently, by Leino [Lei95, DLN98, LN00].

3. STRUCTURING THE SYSTEM ARCHITECTURE. De Champeaux's "ensembles" were the first proposal for sub-systems (in object-oriented system analysis) that had object-like features like attributes, message handling, and encapsulation of constituents [Cha91]. Embley et al.'s *high-level object classes* without representative (see below) can be understood as large, complex subsystems [EKW92]. Gangopadhyay and Mitra [GM93], Rumbaugh [Rum94c, Rum95], and Harel and Gery [HG97] used composite objects to structure the system model/specification and as context for local definitions. Moreira and Clark used "aggregation" with hidden components as "a mechanism for structuring large systems" [MC94]. Bosch [Bos96] organized the entire system into a hierarchy of nested objects.

4. HIGHER-LEVEL ABSTRACT VIEWS. Embley et al. collapsed, among others, *low-level objects* with their links and interactions, to one *high-level object* in object-oriented system analysis [EKW92]. And, conversely, they established the meaning of high-level objects in terms of low-level objects. They distinguished high-level views where the high-level object has or has not the same identity as one of the low-level objects, i.e., where there is, or is not a representative (called *dominant object*) in the expansion. Moreira and Clark's "aggregations" with hidden components [MC94] and Rumbaugh's "composite objects" [Rum94c] could be viewed as a single object at a higher level of abstractions. Kristensen [Kri94], and Bock and Odell [BO98] demonstrated complexity management by higher-level views of composite objects and their connections.

SYNTHESIS. Already in 1987, Blake and Cook [BC87] distinguished "additive wholes" (or "collections") from "structured wholes" (like wired-up circuits). They related the latter to classical decompositional analysis, and identified the dilemma between making the part objects accessible to other objects for "a knowledge representation style of programming" [SB85], and protecting the whole's integrity against violations through state changes in part objects. Six years later, Civello [Civ93] distinguished *functional* parthood relationships as making the part "conceptually included" in the whole and deserving encapsulation. He was first to point out the dual use of part hierarchies for modeling part relationships between entities in the domain, and "to control design complexity by encapsulating the parts of composite objects." Moreira and Clark [MC94] similarly distinguished *shared* components from the *hidden* components that permit "the aggregate to be seen as a single object at one level of abstraction, so it can be used as a structuring mechanism."

Rumbaugh established the terminology adopted by the UML modeling standard: Whereas ordinary *aggregation* relates objects at the same semantic level [Rum94a], *composition* produces an aggregation tree that can be abstracted at various levels [Rum94c]. A composite object can be viewed "either in detail or as a single abstract object subsuming relationships to its parts" thus providing "a vehicle for suppressing detail" [Rum94c]. Composite objects can be used to structure the system and as the context for the definition of component objects, their connections, and constraints [Rum95]. Distinguishing components in object-oriented modeling into private and public (*external vs. internal* composition) was proposed in [VMO99].

Chapter 3

Encapsulation in Object-Oriented Programming

The big lie of object-oriented programming is that objects provide encapsulation.
Hogg (1991)

A single object may be encapsulated, but single objects are not interesting.
An object must be part of a system to be useful,
and a system of objects is not necessarily encapsulated.
Hogg et al. (1992)

This chapter zooms in on the purpose of this dissertation: encapsulation for composite objects. It develops the purpose not out of examples of what we want or don't want to happen at runtime, but out of the general software quality of *modularity* that enables divide & conquer development, modular verification, and substitutivity. Encapsulation and information hiding are two complementary aspects of modularity generally agreed to be essential features of object-oriented programming. Their different, competing concretizations will be reviewed.

Encapsulation and hiding limit external (respectively, read or write) access to internal "information" to support, respectively, verification or substitutivity. This may include more than just limiting external references and access to internal *parts* (fields and component objects), since also the information which parts there are has to be protected. Hence modularity for composite objects requires more than to apply alias control or access control to inbound references.

3.1 The Importance of Modularity

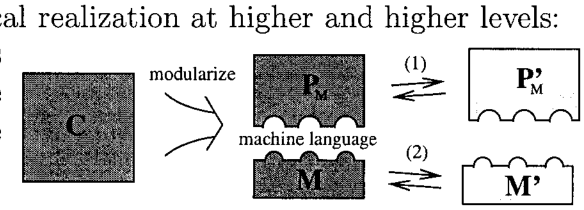
1. Modularity in General. A structuring of the program or system that manages its complexity—which was the subject of the previous chapter—is not automatically a good one. The structuring is of good quality if it is *modular.* ***Modularity*** means the minimalization of couplings, or dependencies, between the structural units [Qui95].

('Coupling' is a dependency in one direction or another, or both.) Effective decoupling is "indispensable for the development of large programs" [Wir83]. Structuring guides the focusing of attention to the limited amounts of complexity within one structural component and one nesting level, and ignoring rest. Modularity is necessary so that complexity ignored in the focused view is, for the most part, *irrelevant* for the structural component in our focus.

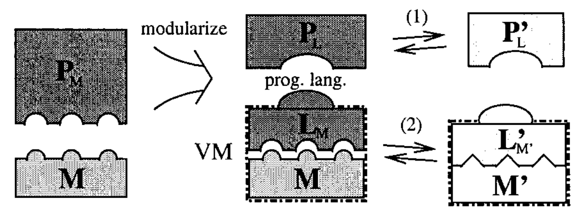
There are three well-known applications of modularity in programming (also found in Wirth's and Wills's analysis of 'hiding' and 'encapsulation' [Wir83, Wil92]):

- ***Divide & conquer.*** The classical divide & conquer problem solving technique presupposes a degree of modularity: Dividing a software development problem produce several smaller subproblems (without reducing overall complexity). Modularity is necessary so that each subproblem can be solved "nearly" independently from the others. The subproblems' solutions (portions of the program code or of the runtime system) combine to a solution for the original problem.
- ***Integrity*** and ***reuse***. Modularity limits the dependencies of a component on the others, its context. The context makes "nearly" no difference to the component (*context independence*, *implementation integrity*). This makes a component more easy to comprehend, and more easy to "unplug" and reuse in a new context [WB+95]. Assuming these limited dependencies (e.g., imported interfaces) are satisfied, it is even possible in principle to verify the component's correct functioning without further regard for its context (modular verification). By thus guaranteeing the correct funtioning of some components, we are "able to limit the area of error search in the case of a malfunctioning program" [Wir83]. A component that works correctly in one context can be "unplugged" and reused in any context satisfying the dependencies (e.g., providing the imported interfaces), and one can rely on it to continue working correctly. No re-verification relative to the new context is necessary.
- ***Transparency*** and ***substitution***. Modularity limits the context's dependencies on the component. Most aspects of the component are irrelevant for the context (*implementation transparency* or *independence*); we can safely ignore them in reasoning about the context. Consequently, the potential for a ripple effect by an error in the component is reduced [WB+95], and changing a component internally or substituting it by a new one is less likely to have an impact on the context [WB+95], should not require any adaptive changes in the context [Wir83].

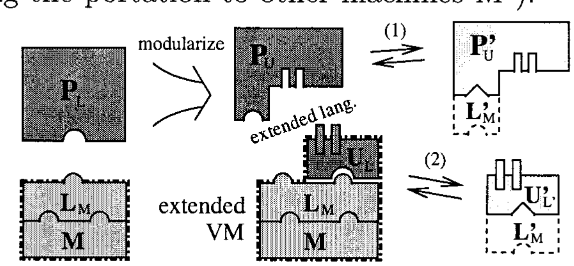
2. FOR EXAMPLE, modularity was applied with great success in computer systems to separate abstract solution from technical realization at higher and higher levels:



First, a defining feature of computers is *programmability*: Computer systems C are abstractly divided into a general purpose machine M (hardware) and a program P_M (software) specifying a particular computation. Computer engineers can focus on constructing computers that execute machine language programs P'_M without malfunctioning. Programmers can focus on expressing computation in machine language; programs P_M can be reused on other machines M' with different hardware but without adaption of P_M if the machine model is the same. (Before this innovation in the 1830's by Babbage and in the 1930's independently by Aiken, Stibitz, and Zuse, the automatization of each computation required to build a different computer, or rewire an old one.)



Second, "high-level programming languages" coming up in the late 1950's separated the high-level program P_L specifying the actual, machine independent computation, from the language's implementation L_M defining the program's translation to the machine. Language implementation L_M (compiler or interpreter, and execution environment) in conjunction with a machine M that can execute it, is a virtual machine that can execute P_L. *Typed* programming languages are designed so that the language implementation's correct functioning cannot be influenced by any program P'_L, and so that the language implementation can be updated or replaced by $L'_{M'}$ under the unchanged program P_L (enabling the portation to other machines M').

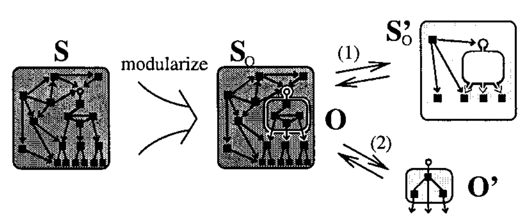


Third, while the original program module was the procedure, the class construct of the first object-oriented language Simula of 1967 [Bi$^+$80] brought the insight of the 1970's that good larger-scale modules result from collecting all the procedures coupled by access to the representation of the same abstract data [Hoa72, Par72, LZ75, GH75, Lis92]: Program P_L is divided into the core program P_U with the high-level program logic, and the implementation of user-defined data types contained in multiprocedure modules U_L. These modules extend the programming language by "a vocabulary of data types" [SG96] (general purpose as well as application-specific). A *modular* programming language ensures the independence of each module U_L's functioning from any context P'_U, and ensures that revisions or reimplementations U'_L of the module have no impact on P_U. (It may even be possible to combine modules and core programs that were translated with different compilers or written in different languages, as in the .NET architecture.)

3. A Step Back? In object-oriented programs, the class definition is the natural multiprocedure program module. However, the central object-oriented software construction technique of *class inheritance*, i.e., the definition of one class by derivation from another, compromizes the program's modularity [Sny86, MD95]. The main thing possible with inheritance but not with object composition is the overriding inherited methods, allowing one to redirect other inherited methods' call of this method (self-calls) to a new implementation [Hau93, SM95]. But it is also the source of a new kind of tight coupling between class modules known as the ***instable*** or ***fragile base-class problem*** [Pre97]. The solution through some form of explicit *specialization interface* between base-class and derived class is still an open issue of research [Lam93, St$^+$96, Sta97, RL00]. The usual advice given to programmers [Ga$^+$95, MD95, Pre97] is to avoid creating new classes by class inheritance with its dependency on the internal method call structure of the base-class ("*white-box reuse*" [Pre97]), and to prefer object composition, where component objects are used through normal exported object interfaces, a clearly defined, well-understood concept ("*black-box reuse*" [Pre97]).

4. MODULARITY IN OBJECT SYSTEMS. In object-oriented programming, the structure of the runtime system model (the object system) does not coincide with the structure of the program (§2.3). Besides the modularity of the program's partitioning into class modules—whose interfaces extend the language in which the program is written,—we can also talk about the modularity of the system's partitioning into objects—whose interfaces specify the language in which objects communicate.

What does modularity mean for a runtime system? It is relevant not for what the programmer can do with program modules, but for what the computation can do with components of the runtime system, in particular, with composite objects:



- A runtime component O's implementation can be verified independently from the runtime context S_O in which it is used. It can be transfered between different parts S'_O of the system, even migrated to other systems S'_O, without starting to malfunction.

- What implementation the runtime component has is transparent to the context. Hence it can be substituted without impact by a component O' with a different implementation.

It is wrong to think that these properties are only interesting for systems with an infrastructure for the dynamic migration of runtime components and for the dynamic switching between implementations. They are crucial for all object-oriented programs since (composite) objects, the runtime components of object systems, are transfered and substituted all the time: Any passing of an object reference to O as parameter into an object's method is like the transfer of O into the context of this object and is like the substitution for a formal parameter object (or for previous parameter objects). Any redirection of an object reference variable from O to O' is like a substitution of target object O' for object O and thus a move of O' into the context S_O around the reference's source. Finally, there are special design patterns which separate the decision about from which implementation to instantiate an object O on one hand, from the object's use in a context S_O on the other hand, so that it can easily be revised or decided dynamically: Factory Methods, which "pervade toolkits and frameworks," Abstract Factories, which are the basis for component systems, and Prototypes, which are the foundation of prototype-based programming [Ga+95].

Observe the distinct advantage of the transparency-aspect of object modularity: Transparency of class modules decouples clients of a class module from the definitions in the module. It makes it safe to revise field and method definitions, corresponding to a simultaneous change of these fields and methods in all instances of the class and its subclasses. Transparency of objects, on the other hand, decouples the clients S_O of objects O from the decision what class c of objects O to supply as parameter, assign to the variable, or create. It makes it safe to revise this decision statically or dynamically on a case by case basis, and thus revise from which class module the method comes that implements the client's invocations on O. That is, object transparency is the condition under which a foundation of object-oriented programming is safe: mixing objects from different classes (*polymorphism*) and executing method code depending on the receiver object's class (*dynamic binding*). (The programmer must not forget that the methods' externally visible behavior is not an implementation detail, and must be preserved, cf. *behavioral subtyping* [Ame87, LW94, DL97].)

3.2 Information Hiding and Encapsulation

1. MODULAR ESTABLISHMENT OF MODULARITY. A division into components is not automatically a modular one. Transparency and integrity of a component X depends not just on X itself, but requires also that no other component, respectively, depends on, or interfers with, X's internal working in any way—something very difficult to check in general. Hiding and encapsulation are two prominent programming principles that make the context check superfluous or at least independent from X's interior (i.e., a modular check), and for which techniques for their automatic enforcement exist. They allow a component to establish its own transparency and integrity.

Different concretizations of the notions of hiding and encapsulation exist in the literature, as we will encounter in §3.6. Often the two are used interchangeably for a principle addressing both integrity and transparency, with encapsulation being rather the technique and hiding rather the abstract property it achieves. In the following, the term 'encapsulation' will be used in a very particular sense that opposes it to 'hiding' w.r.t. the direction of the tackled dependency (cf. [KM95]):

a) ***Hiding*** removes the component's internal properties and working from external *view*, wraps the component in black (*black box*). It reduces the context's potential dependency on the component (furthering implementation transparency and substitutivity), namely dependency on the component's design, by making it impossible to develop a dependency on its internals. According to Parnas's famous information hiding principle, "A module is characterized by its knowledge of a design decision which it hides from all others" [Par72].

b) ***Encapsulation*** protects the component's internal properties and workings from external *manipulation*, wraps the component in a capsule (*protective box*). It reduces the component's dependency on the context (furthering implementation integrity and verifyability), namely dependency on how the context uses the component, by making it impossible to develop a dependency on (the benignity of) its manipulations. "If a language enforces encapsulation, [context-]independent reasoning about modules is on a sound foundation. Otherwise, it isn't and a complete proof requires a global analysis" [Lis92].

Observe that hiding as well as encapsulation remove neither the context's dependency on the component's *external* behavior, nor the component's dependency on how it is used by the context, nor the component's dependency on the context's implementation of *imported* services. Also, there is a gray area regarding imported services requested by the component: Can they be allowed to view and manipulate the component's internals? This will be considered in §3.5.

2. INTERIOR AND INTERFACE. The common view of hiding and encapsulation reduces component-internal "information" to mean the internal *parts* of a component that is an aggregation of subcomponents. This view presupposes a division of the subcomponents into two groups: Some are designated as exported parts or *interface parts*; the others are called internal parts or *private parts*. For hiding it then suffices to prohibit the outside's access to the component other than through its interface parts. This limits the context's dependencies on the component to that which is visible through its interface. And for encapsulation it suffices to prohibit the outside's modification of the component other than through its interface parts. This limits the component's dependencies on the context's manipulations to those possible through the interface. That is, a protection domain is established by, metaphorically speaking, the drawing of a barrier around the component—called *encapsulation barrier* in both cases—which has the private parts protected inside of it and the unprotected interface parts crossing it.

But *not all internal information is an internal part*, there are also internal *structure* and *state*, which are not parts. The reduction to internal parts works only if the number of subcomponents is fixed, as in a program module or in an implementation object. If their number can change—as in the case of composite objects—this is an aspect of the encapsulation unit's state and it is not necessarily represented in the state of its private parts. This issue will be picked up again in the discussion of composite object encapsulation in §3.3, paragraph *3*.

3. CURRENT OBJECT-ORIENTED ENCAPSULATION. The two main mechanisms by which parts of any software system may interact are the access to *shared variables* and the exchange of *messages*. Interaction through a shared variable creates a coupling that is considered worse (tighter) than that through the exchange of messages. Procedural programming has been scolded for its tight coupling of distant program parts through global variables and global data structures [Mez98].

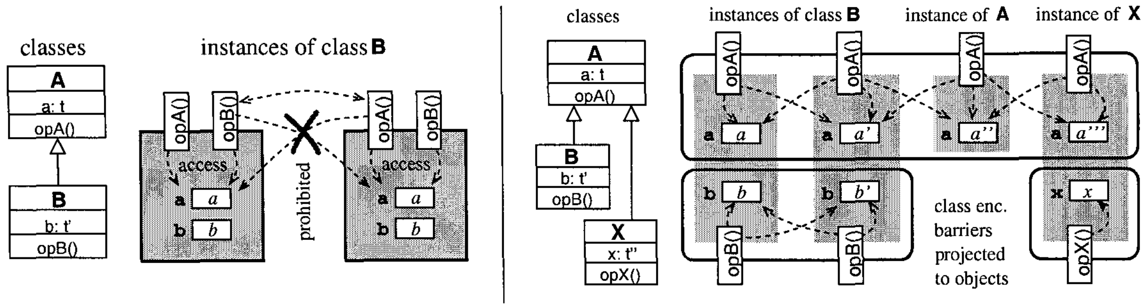


Figure 3.1: Encapsulation models in object-oriented programming

In object-oriented programming, *encapsulation* (in a sense that includes hiding) is an essential feature. While general introductions to object-oriented programming [Weg90, Qui95, Bru96, AC96, Cas97] present the *implementation object* as the encapsulated unit, most object-oriented programming languages support the encapsulation of the *class module*. The addition of either kind of encapsulation is an improvement since it contains all the "bad," variable-based coupling within the units, while between them there is only the weaker coupling by message exchange. (On a larger scale, the problem with shared variability reoccurs—see paragraph *2* in §3.3).

3a. THE ELEMENTARY OBJECT CAPSULE. The first object-oriented language, Simula, developed 1967 for concurrent system simulations [Bi+80], had objects and a class definition construct but no encapsulation. The second object-oriented language, Smalltalk [GR83], designed between 1972 and 1980, defined the canonical understanding of encapsulation object-orientation (see the left hand side of figure 3.1):

The implementation object is the encapsulated unit, with the fields as private parts and the methods as interface parts. Only an object's methods can access its fields (irrespective the class modules in which both were defined). Objects with a reference to another object can use it to send operation requests but not to access the target's fields. For the modularity *of the system* it is irrelevant in which modules the fields and methods were defined. The object's implementation code as a whole, in its class and superclasses, can be verified, and the object can be substituted by another one with different fields and/or different implementation of the methods.

The technique by which Smalltalk enforced this is that it simply provides no syntax $E.x$ for accessing a particular object's fields. One can only write the identifier x to refer to the field x of the current object.

3b. MODULE-BASED ENCAPSULATION as introduced by C++ 1983/86 [Str94] and Eiffel 1986/88 [Mey88] made the object-oriented paradigm more acceptable to the software engineering community and consequently became the dominant form of encapsulation supported by object-oriented programming languages. It is based on *scope-rules*: The *names* of private fields are simply not available outside of the class module defining them. (Different visibility ranges can be specified, but this is not the issue here.) Hiding field names makes it impossible for other modules to express an access $E.x$ to a field x. It prevents dependency on the definition of the field (*linguistic coupling*). (However, if fields can not only be accessed by name but also by pointer, as in C++, then there may be a dynamic coupling, cf. §3.5.)

Consider what this means for the access to the fields at runtime (see the right hand side of figure 3.1): An encapsulation barrier is erected that contains as private parts from all objects the fields that were defined in the same class module. The fields can be accessed only by methods defined in the same class module; they are the interface parts. For the modularity *of the program* it is irrelevant that these methods are the methods of all instances of that class and its subclasses: The class module can be verified and revised since it simultaneously defines the accessed fields and the accessing methods of all these instances.

(This model can be extended, as in Java, by another encapsulation barrier associated with multi-class packages. It can enclose the encapsulation barriers of the class modules in them, and contains package-private fields and methods demanded by Szyperski's "no paranoia rule" [Szy92], as well as package-private classes.)

3.3 The Need to Encapsulate Composite Objects

1. UNSUFFICIENCY: REFERENCE-INDUCED COUPLING. The above two standard models of object-oriented encapsulation contain bad, shared variable-based coupling within implementation objects or class modules. They clearly separate computation in-the-small with tight coupling from computation in-the-large with weaker coupling at the objects' or modules' boundaries.

However, because the complexity of each implementation object is limited, groups of objects have to collaborate for larger tasks, which leads to problematic coupling also by message exchange. The Demeter system tried to reduce coupling by the design rule "*Law of Demeter*" [LH89] that deprecated calls through temporary references, so that the direct effects of a method invocation were limited to the objects referenced by the receiver's fields and the method's parameters.

In particular, an object can act as an abstract variable whose sharing between several objects or modules leads to a coupling similar to that through global variables. The combination of sharing and mutable state has repeatedly been identified as causing serious problems [NVP98, Cla01], and making object systems so notoriously hard to reason about [Wil92, Ho+92, Alm97]. This was already observed very early by Jones and Liskov [JL76], who saw this leading to "the need to exercise some control over exactly how the object should be shared."

The praxis of object-oriented programming has shown that problems by aliasing "do not manifest themselves in the vast majority of programs" [NVP98]. But this depends solely on a self-disciplined manner of using references. One documented example were this discipline failed is a bug in the Java Development Kit (JDK) version 1.1.1 that caused a security hole in Sun's HotJava web browser [SIP97]: The

JDK `Class` object o_c that reified Java (downloaded) classes c returned, instead of a copy, the actual array object a holding the "digital signatures" for c. By overwriting its signatures with signatures from trusted classes, class c can rid itself from HotJava's security restrictions. This can be understood as a problem of aliasing or write access, of encapsulation with a as private component of composite object o_c or as private to JDK's class `Class`.

Any limitation of how objects are accessed through references (***access control***) or of the existence of sharing-enabling reference aliases (***alias control***) reduces the coupling in the object system and simplifies reasoning about its dynamics. Alias control, in particular, has a long tradition in reasoning about procedural programs with multiple names for the same variable (e.g., through parameter passing by reference) [Rey78], or with pointers (subclassified by Euclid's *collections* [La$^+$77], or effects systems' *regions* [LG88]). It was also employed for safe parallelization of programs with pointer data structures [HHN92, KS93, HHN94] and for optimized memory deallocation (*linear types* [Wad90, Bak95], *regions* [TT94], *escape analysis* [Bla99], *calculus of capabilities* [CWM99], *alias types* [WM00]).

However, the aimless reduction of coupling cannot ensure verifyability and substitutivity. For the question of *modularity*, alias/access control and a reduction of coupling is of interest only in so far as it concerns references and couplings that cross the boundaries of some runtime system components.

(A radical solution for the problems with object references would be to replace them by a more abstract mechanism of referring to objects [Kri94], e.g., by communicating through *out-ports* which are connected to in-ports by the enclosing composite object [GM93, MC94, AKC01], by *acquaintance categories* [Bos96], or by paths of *composite-local names* for its component objects [RBF98]. But how much this would actually reduce the effective coupling in the system remains unclear.)

2. UNSUFFICIENCY: GRANULARITY TOO SMALL. Elementary object encapsulation allows one to verify the implementation of an implementation object's external behavior and the substitution of an implementation object by another one that implements the same external behavior. Similarly, class module encapsulation allows one to verify the implementation of its instance's module-external behavior and the substitution of the module by another one that implements the same module-external behavior. This suffices for simple data abstractions like calendar dates that are realized by a single implementation object.

But for an implementation `MapImp` of abstract map data structures this is unsatisfactory since also the communication of a `MapImp` instance with its entry-set component and entry-pair components is external behavior for `s` *as an implementation object*, the representative. It is unsatisfactory to merely verify that representatives correctly communicate with their components and correctly produce results depending on the replies (cf. §2.6). It is unsatisfactory to merely replace representatives or their definition module `MapImp` by another one with the same communication with components. The structural unit in the design of the object system is the composite

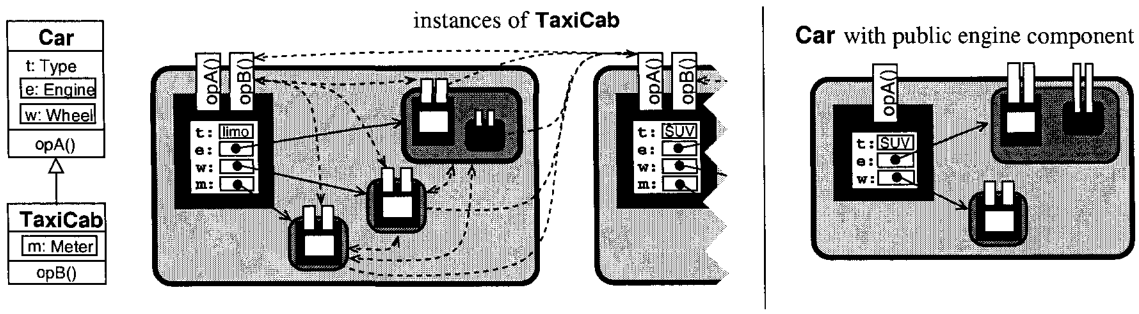


Figure 3.2: Encapsulated composite objects

object. We should to be able to verify the entire implementation of, e.g., an abstract map data structure by the composite object's external behavior, and to replace the entire composite map by another one with a different implementation, e.g., with two array objects (one for the keys and the other for the corresponding values).

3. COMPOSITE OBJECT ENCAPSULATION. Soon after object-oriented programming started to be taken seriously, demand rose for encapsulating entire groups of the objects. In particular those which were seen as one unit with object-like properties, and were later called *composite objects*, should also have the characteristic object property of encapsulation. Forms of composite object encapsulation were suggested in part-whole modeling in programs [BC87] and databases [KS92], in object-oriented system analysis [Cha91] and architectural modeling [AW+92, GM93, Bos96], and in formal methods in object-oriented programming [Hog91, Wil92, Utt92, Lei95]. This concern for larger-scale modularity in the runtime system was ignored by the development of object-oriented programming languages. (But it motivated in the 1990s, via document-centered architectures like OLE and OpenDoc, the development of *component system architectures* like COM, CORBA, and JavaBeans [MD95, OMG00, Ham97].)

Encapsulation or information hiding for composite object concerns (information about) their fields and their component objects. In the canonical case, these are the composite's interior, and its operations (implemented by its representative) are the only interface parts. For example, see the composite TaxiCab object with Engine, Wheel and Meter components in figure 3.2. In analogy to the notion of public fields in modular encapsulation, one could also have *public* component objects, whose interface parts are exported as additional interface parts of the composite. For example, a Car object might make its engine component public (fig. 3.2). And if the engine has a public oil-measure-shaft component, the Car object could export that too.

An encapsulation of composite objects is always understood to be added on top of an encapsulation of implementation objects. That is, the elementary object encapsulation barrier encloses the object's fields (cf. previous section, paragraph *3*). It and the (encapsulated) component objects are enclosed in the composite object encapsulation barrier (see figure 3.2). The composite's methods are the interface parts for both the elementary and the composite object encapsulation barrier.

Consequently, we can focus on hiding or encapsulating the information about the composite's components. It has two aspects, and these are both mutable (cf. *composite state* in §2.5), so that dependency on either of them is possible:

- First, what are the composite's components?
- Second, what are their current states?

The first aspect is often overlooked, and it is assumed that coupling between two composite objects can only be established by the existence of object references to the composite (good) or to its component objects (bad). Hiding and encapsulation would then be equivalent to controlling, respectively, the *existence* or *use* of inbound references from the outside to the component objects, i.e., to alias control or access control across composites' boundaries.

For an *information hiding* policy it is insufficient to prevent the external access to internal components that could observe (or change) their states, i.e., access control on inbound references. Even unused inbound references can represent the outside's knowledge of who the components are. Hence information hiding for composite objects *is* a form of alias control at composites' boundaries that prevents the outside's (non-contained) possession of references to components.

And for an *encapsulation* policy it is unsufficient to limit the use of inbound references to read-only access that does not modify any component object, i.e., a form of access control at composites' boundaries: It also has to exclude the external manipulation of the composite's set of components. The concretization of encapsulation w.r.t. this aspect depends on how it is determined in the implementation object system what a composite's components are at a particular point in time—there are different approaches, which will be considered in §3.4. If it is determined by paths of certain references (cf. §3.4), the existence of such references must be controlled by controlling the state of objects holding them. And if the paths can go through (fields of) external objects, write access to these external objects will have to be controlled as well so that the composite cannot be manipulated w.r.t. its component set by the update of fields. Paradoxically as it may sound, to encapsulate composite objects determined this way, we need access control beyond the composite's boundaries—because it is not self-contained. This is a case where the reduction of encapsultion to a protection of internal parts (cf. §3.2, paragraph *2*) does not suffice for the protection of a piece of internal information.

Observe that preventing inbound references for information hiding entails encapsulation if reference paths determine object composition: It removes the basis for observing access to components and excludes the use of paths to components through external objects. But in case that membership in the composite object is determined, e.g., by containment in a local store (cf. §3.4), there might be a special operation for adding a *new* object to that store, and thus change the composite's composition, without requiring a reference to any of its *old* components. This is a case where the hiding of all internal information (who are the components and what is their state) does *not* entail encapsulation w.r.t. them (cf. §3.2, paragraph *1*).

3.4 Directions of Research in Encapsulation Units

Research in the encapsulation of composite objects has produced many concretizations of the notions of hiding and encapsulation in particular contexts. They differ widely in how the encapsulation barriers are drawn, what precisely is allowed to cross them outside-in (discussed in the next section), and how this is called. (One may interpret this as a sign for lack of maturity of the field.)

1. Classes of Module-Private Instances. If a class module c is private to a package or class module M, we might expect the c-instances to be in some way private to M. That is, in the runtime system model there is a protection domain D_M associated with M that contains, besides all object fields that were declared in M, also all the c-instances. This is not the case. A scope restriction of the class name c to the package or module M cannot guarantee that c-instances are accessible only from code in M: In Java, *all* classes have a non-private superclass, namely the special class `Object`. But if class c has a superclass c' that is not private in M, then references to c-instances can be leaked as references of static type c' to code outside of M. It can then invoke c'-operations on the c-instance.

To solve this problem, the notion of ***confined types*** was developed by Vitek and Bokowski [VB99], and later refined by Grotthoff, Palsberg and Vitek [GPV01]: A class declared `confined` is not just a private module in the enclosing Java package but also its instances are private, i.e., can never be referenced at runtime from fields and methods defined outside the package. All the code accessing the instances is located in the enclosing package.

Instead of defining a new `confined` class each time one wants package-private objects, one could also take classes c from any package M' and assume a generic, ad-hoc subclass c_M of it for every class module or package M in the program such that c_M is confined to M. That is, only methods defined in M have full access to the instances of M-qualified class c_M. This idea is realized in the ***type universes*** system of Müller and Poetzsch-Heffter's Universes system [MP99a]: c<T> is the class of c-objects private to the package M in which class T is defined. The "type universe" of T is the collection U_T of all instances of T's classes c<T>, c'<T>, etc. The union of all universes U_T of classes T in package M is the protection domain D_M associated with M. The encapsulation policy of universes, *representation encapsulation* (§3.6), allows external read access to the instances of confined class c<T>. But all *write* access is limited to code in T's package M. The c<T> instances in each universe $U_T \subseteq D_M$ "can only be manipulated by methods implemented in [M]. Therefore, type universes provide sufficient sharing control for modular reasoning, since all "dangerous" code is located in one [package]."

One can use confined types and type universes for the encapsulation of *composite objects* of class T by using as the types of component objects only, respectively, `confined` classes or qualified class c<T>.

Observation 1. Consider that a c-object ω used as components of T-composite o is a composite object with component q of class d, and assume that T, c, and d are defined in different packages (only then are type universes a real advantage over confined types). Since o's component ω is of qualified class c<T>, only code in T's package has full access to it. For components q (of qualified class d<c>), this means that it cannot have full access to its own composite ω (unless through methods inherited from classes in T's package). Since all classes should be qualifyable (to obtain component objects) it would be unsafe to program composite classes whose instances give their components a writable back-link. This is a restriction of composite objects' internal working that is not necessary for composite object integrity and substitutivity, but an idiosyncrasy of encapsulating objects in modules.

Observation 2. Type universes work well only for composite objects which create their components themselves. With patterns of flexible object creation and composition, problems arise since the class of the component's composite must be fixed at creation time. Consider first `SetImp` objects, which create iterators over their elements (the Abstract Factory design pattern). The iterators from the `SetImp` object `s` used as the entry-set in a `MapImp` composite needs to be wrapped in a `FirstIt` object to produce iterators over keys. In order to create iterators as components of composites of different classes T, one would need to make set objects' factory method `elements` polymorphic with class parameter T. But since T = `FirstIt` is in a different package than `SetImp` (see fig. 2.6), type universes would prohibit `s` from accessing the newly created iterator (e.g., for initializing it to the right position). Second, consider an abstract parser class `AParser` which provides an operation for configuring the parser with a scanner component (a generalization of Leino's example [DLN98]): The parameter's type can only be `Scanner<AParser>`. But then no parser implementation is possible where the scanner object is a component of one of the `AParser`'s subcomponents. Also parser implementations in a different package than `AParser` cannot make any use of the scanner object.

2. FIELDS WITH OBJECT-PRIVATE TARGETS. A fundamentally different—though superficially similar—idea is not based on generalizing the privacy of classes to their instances but on generalizing the privacy of reference fields to their targets.

The simplest version is to let all objects reachable from a given object o along object references captured in (private) fields be private to o. The applicability for the encapsulation of composite object is limited, though. It makes sense only for composite object without captured references to objects in its context. If all objects were encapsulated this way, no cyclic linked data structures could be constructed and an object could not be stored in two set container objects at the same time. It is telling that the two techniques supporting this form of encapsulation apply it only to selected objects, called ***islands*** [Hog91] or ***balloons*** [Alm97].

A variation is to distinguish those reference fields whose target we want to be private, ***component fields***, from normal fields. Historically, this idea was implemented first without encapsulation, e.g., in the object-based KI system LOOPS of

1983 with the keyword `part` [SB85], and the object-oriented database ORION with special fields of "composite references" proposed 1987 by Kim et al. [Ki+87, Ki+88]. Different forms of encapsulation of the component fields' targets were added in the language Sina with keyword `internals` [AW+92], in an extension of Modula-2 with the keyword `private` [Lei95], of Eiffel with the keyword `unique` [Min96], or of Java with the keyword `unshared` [GB99], and in the specification language Object-Z with proposed declaration annotations Ⓢ, ©, and Ⓔ with different sharing constraints [DD95a, DD95b]. In two formal reasoning techniques [Wil92, DLN98], the component status of field x of object o is implicit in the specification that o's abstract state is represented in, or depends on, some field of x's target.[1]

As a general solution for the encapsulation of composite objects this is too inflexible: Since the number of fields is fixed, the number of private component objects would be bound. An implementation of `Set` with an internal, dynamically-growing, cyclically-linked storage structure would be impossible to encapsulate.

3. REAL COMPOSITE OBJECT ENCAPSULATION establishes an encapsulation barrier around all the fields *and* component objects of a composite object, independently from packages and fields. Since composite objects can be nested recursively to a composition hierarchy, the encapsulation of all of them produces a hierarchy of nested encapsulation barriers. (They combine without intersection with the smaller elementary object encapsulation barriers around just the private fields.) Various, sometimes overlapping, ways have been used for defining a composite object's private objects without the above described problems:

a) On one hand, object reference-based determination of privacy can be extended to using entire *paths* of references in the object graph. Not only composition references must be distinguished, also other kinds of object references must be distinguished according to how they combine to *composition paths* whose final object is private to the initial object. This seems to be the unspoken idea behind ***flexible alias protection*** [NVP98].
b) Also the class qualification approach can be developed further by assuming *for every object o*, a generic, ad-hoc subclass c_o of any class c all of whose instances are private to o. Such classes have been described as o's ***local classes*** [KS92], classes or object types with (main) ownership parameter o (***ownership types***) [CPN98, Cla01], or o's "copy" of class c [MP01].
c) Effectively similar is to associate every object o with a protection domain D_o so that all objects that are in it become private to o. D_o is either a variation of Euclid's collection [Utt92] later called o's ***local store*** [Utt96], a so-called ***rep context*** (providing "a nested partitioning of the object store") [CPN98, Cla01],

[1] Dong and Duke [DD95a] and Almeida [Alm97] observed that `expanded` classes in Eiffel protect against aliasing: Reading the value from `expanded` fields means to copy the object ω in it, means to create a clone of ω. It is however not clear if also the reading of the `this` variable in methods of ω inherited from non-`expanded` superclasses creates a clone.

"a partition of the object store" called ***object universe*** [MP99a], or a protection domain called ***object space*** [CR00]. Either D_o is a real runtime construct and an object is made a member by creating ω "in" domain D_o [Utt92, CR00]. Or D_o is a metaphor for being private to o by reference path or qualified class [CPN98, MP99a, Cla01].

d) A more direct expression of object composition is Kent and Maung's notion of ***object ownership*** [KM95]: All objects ω have an implicit attribute, called their *owner*, to which they are private if it is non-`null`. For example, the component objects of COM ("inner objects") have an implicit owner attribute ("outer object"), which they return when asked for their IUnknown interface [MD95]. An object's owner is either fixed implicitly at the time of its creation relative to the creator [KM95], is set implicitly by converting a `unique` reference targeting it to an `owned` reference [ACN02], is set by a special operation on the component [MD95] (before the first IUnknown query [SM97]), or derived from membership in class c_o or domain D_o [CPN98, Cla01, MP99a, MP01].

All these approaches can in principle encapsulate all interesting composite objects. Since the three non-path-based approaches are independent from the existence of references between objects, they are slightly more flexible in drawing encapsulation barriers around composite objects, whatever their internal structure: They support objects that are private to o (members of c_o or D_o) but which o cannot reach. In path-based flexible alias protection [NVP98], if all composition paths from composite o to component ω are destroyed, ω cannot but lose its official status as private component of o.

The qualified class approaches [KS92, CPN98, MP99a, MP01, Cla01] have a principle problem with flexible object creation and composition: The owner must be fixed before the class can be instantiated, and changing it later would amount to changing the object's class ("metamorphism"). Even Clarke, whose work is the most advanced, admits that "this is unlikely to be sound" [Cla01]. Clarke's owner-polymorphic method can solve many simple cases. But, as Clarke shows, heavy restructuring of the control flow is necessary for a more elaborated example like the configuration of a parser object with unknown internal structure by an externally created scanner object from [DLN98] (cf. paragraph *1* above).

For two real domain-based and ownership-based approaches, that are not derived from qualified classes, the authors consider the explicit switch of an object's owner by operations `transfer` [Utt96] or `acquire` [KM95]. This would seem to support in principle all patterns of flexible object creation and composition, although some conditions might be needed to make transferring ownership a clean and safe affair.

Flexible alias protection has one type of object reference particularly for flexible object creation and composition [NVP98]: The initial reference to a new object is `free`, and `free` references can be passed between objects, and converted to any other type of reference by assignment to a corresponding variable.

4. VARIATION: PRINCIPAL-WITH-PROXIES COMPLEXES. In Clarke's Unique Representation Calculus $\mathsf{Dl}\varsigma$, ***aggregates*** are runtime components made of one (principal) composite object together with (proxy) objects for accessing it [Cla01]. Effectively the same happens in AliasJava, where one (principal) composite object can create other objects which have full access to its interior but are not its components [ACN02]. For instance, the encapsulation barrier of a set object s may be extended to include also the iterators (proxies) over it. This is shown in the left side of figure 3.3.

Formally, this is the same as composite object encapsulation with *public components*; there is only the semantic distinction whether the additional interface object (the proxy) is a component of the composite or not. Public components are supported in Microsoft's component standard COM by the notion of "***aggregation***," in which the composite object ("outer object") can return (interfaces of) aggregated components ("inner objects") for direct use by clients. Note that COM does not come with a mechanism that would help to enforce any encapsulation policy: The working of COM aggregation relies on the unverified assumption that references to components, or more precisely to their COM interfaces, are only ever exported to clients via the special QueryInterface operation [SM97]. Since standard COM containers return their iterators through operations like EnumObjects or EnumViews [Mic02], this appears to mean that COM does not make iterators additional interface objects that would extend the composite container object to a principal-with-proxies aggregate.

General aggregates of a principal with proxies are not a standard design abstraction of object-oriented programming. There is a structural problem too: Principal-with-proxies aggregates do not scale well with the parallel composition of composite principals and composite proxies. A *map* object d (also called dictionary object) implemented with a set component s has its iterators composed from s's iterators. If d's iterators want to have their s-iterator components within their own (principal-with-proxies) encapsulation barriers then they will have to be included into s's principal-with-proxies barrier. This results in an unbalanced structure where a (multi-level) composite proxy has to be located in the smallest principal-with-proxies aggregate from whose proxies it is (indirectly) composed. This gives the proxy (d's iterators) unjustified privileges on the intermittent components' (i.e., s's) private parts.

5. VARIATION: COLLECTIVE RUNTIME COMPONENTS. An aggregation of objects, like a husband and a wife, does not need to be reified in a separate object, the family object (which represents the family as a whole, carries the family attributes, and provides the family operations). Clarke models it as one encapsulated collective aggregate F consisting of husband object o_1 and wife object o_2 as interface parts, and optional private car object ω [Cla01].

If we understand a map and its iterators as one collective aggregate without a distinguished principal object, the structural difference between principal and proxy disappears (see the right hand side of figure 3.3): Not only do set and map objects give up protection vis-à-vis their iterators, also the iterators give up protection vis-à-vis their principals (and other proxies). This would avoid the unbalanced structure

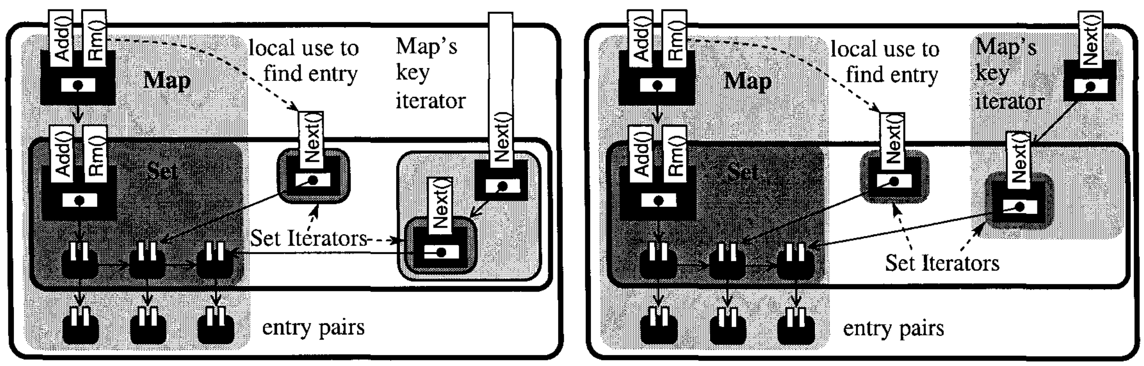


Figure 3.3: Nested principal-with-proxies aggregate, nested collective aggregate

and unnecessary privileges of the treatment as a principal-with-proxies aggregate.

Collective components are supported by the ***aggregates*** of Clarke's Owners-as-Cutsets Calculus Dlς [Cla01]. They allow to draw the encapsulation barriers in the map example as shown in figure 3.3. (Moreover, collective aggregates can overlap, so that one object is interface part in multiple collective aggregates: Husband o_1 and other person objects may be the interface parts (club-members) to a collective aggregate *book club* with books as private parts, while wife o_2 and other person objects are the interface parts to a collective aggregate *music club* with CDs as private parts.)

A kind of collective aggregates is also supported by the ***Object Spaces*** model [CR00]: The objects in one object space S collectively own the objects in all child spaces of S. S's objects are the interface parts and the child spaces' objects are private to them. For example, map object `d` and its key-iterator `w` are created in the same object space D_1. In one child space D_2, all their components with access to `s`'s `Node` components are created, namely, `Set` object `s`, and set-iterators `i` and `i'`. The nodes are created in a child space D_3 of D_2. The remaining components of `d` and `w` without access to the nodes, namely entry `Pair` objects `e1` through `e3`, could also be created in D_2, or in an extra child space D_2' of S.

3.5 External Access despite Encapsulation?

Is hiding or encapsulation violated if the context views or manipulates a component's internals (only) as part of a service requested by that component? The answer to this question distinguishes many proposed concretizations of hiding and encapsulation policies for composite objects (§3.6). One may say 'No' because the resulting dependency of the context on the component affects only the service user, i.e., the component itself. Consequently, one may distinguish strict, *absolute*

relaxed hiding	$\Rightarrow$	relaxed encapsulation
$\Uparrow$		$\Uparrow$
absolute hiding	$\Rightarrow$	absolute encapsulation

notions of hiding and encapsulation, which exclude any external view or manipulation of internals, from *relaxed* notions, which allow it *iff* it is confined to services requested by the component.

The relaxed/absolute question poses itself not only for composite objects but already for class-module and elementary objects encapsulation (§3.2) in languages, like C++, with pointers to fields or parameter passing (of fields) by reference. For example, a class `Point` of 2D-points may implement the `transpose` operation by calling the `swap` operation of a `Util` object to exchange the coordinate values. `swap` has by-reference passed parameters, through which the `Util` object can access the `Point` object's `x` and `y` fields.

```
class Point {
private: int x, y;
public:  void transpose() { (new Util)->swap(x,y); }  // (attn: space leak)
         ...
};

class Util {
public: void swap(int &a, int &b)  { int a0 = a; a = b; b = a0; }
        ...
};
```

This example clearly shows that the technique of prohibiting field access expressions $E.x$ (generally or outside of the module defining x, respectively) cannot by itself guarantee that there is never any access to an x-field from outside the object or module. The access is excluded *only* in the context of programming languages where naming a field is the only way to access it, as in Smalltalk and Java.

The external access violates hiding and encapsulation *in their absolute form.* But *modularity* is not lost since the external access is initiated by code in the `Point` module and is formulated without knowledge of `Point` objects' fields: The `Point` module can be *verified* based on the standard meaning of the imported swap operation. The `Point` module can be *changed* to a polar coordinate implementation (which has no use for the swap operation) without impact on the `Util` module.

Relaxed hiding and encapsulation holds since `Util` objects retain no access after they finished the swap-service, establish no covert channel to `Point` objects' fields. It would be different if method `swap`, in violating of the meaning of swapping, captured a parameter's address in global pointer variable `intptr` by `intptr = &a`. Through this pointer, other modules could observe the fields and become dependent on how module `Point` uses them, or modify them and thus interfere with how `Point` uses them. On the other hand, capturing the pointer would be safe if `intptr` is a variable accessible only directly or indirectly from module `Point`. Also in this case, all access through the captured reference would be contained to operations of module `Point`.

The question of what happens with a pointer or reference passed as parameter to an operation is known in the access control literature as the *confinement* problem [Lam73], the *conservation* problem [CJ75], or as *server containment* [CR00].

3.6 Review of Proposed Encapsulation Policies

The success and acceptance of an enforced programming discipline depends on the extent to which it supports or constrains the programs that programmers actually want to write or reuse [NVP98, Lei01]. Hence let us analyze in how far the proposed variants of encapsulation support or constrain common types of objects. (The example that was considered the most frequently in the literature is the encapsulation of standard container objects like sets, stacks, and maps [Hog91, KM95, GTZ98, NVP98], in particular, including support for iterator objects to access their content [NVP98, CPN98, MP01, Cla01, ACN02].)

1a. STRICT HIDING is the policy most easy to define, to achieve, and to reason about. It means that there are simply never any inbound references. In different contexts, this policy was called ***type isolation*** [Wil92], ***local referential integrity*** [KS92], or *principle of* ***no representation exposure*** [NVP98]. Also COM's ***containment*** [MD95] describes the components ("inner objects") of a composite ("outer object") as "completely hidden" for external objects and never receiving requests from the outside (as opposed to the case of COM's 'aggregation', which means a public component, see §3.4, paragraph *4*).

But absolute hiding is unnecessarily restrictive for the programmer, excluding more common practices of object-oriented programming than the other policies: It makes it impossible to use the design patterns Iterator and Visitor [Ga+95] for working with internal structures. It makes it impossible to implement the functional `union` operation between sets or the mutating `unifyWith` operation more efficiently by one set object exposing its internal structure to the other set object (in the case that they are both of the same implementation class `SetImp`).

1b. RELAXED HIDING allows inbound references iff they are contained. This is concretized by Minski's ***concept of hiding*** (of component objects), which allows access to component objects only while control is in the representative [Min96].

- An obvious specialization of this policy is to constrain the *existence* of inbound references to methods (indirectly) called by the representative. Minsky enforced this by limiting inbound references to parameters which the representative passed by-reference [Min96]. The Object-Oriented Effects System enforced it by prohibiting to capture inbound reference parameter values in fields [GB99].
- ***Representation containment*** is a specialization enforced by the Ownership Types system [CPN98] limiting which objects may possess inbound references: External object q may posses references to o's components iff o hides q in its *shadow* [PNC98], i.e., iff all paths from the initial object to q pass through o. In graph-theoretical terms, o is q's *dominator* or *articulation point.*

Relaxed hiding enables Visitor objects and Iterator *methods* (internal iterators), but not Iterator *objects* (external iterators). It also enables `union` and `unifyWith`. It thus provides some of the flexibility which a *modular* version of private objects introduced in class `SetImp` would provide (cf. §3.4).

2. NO CONSTRAINT EXPORT. The advantage of a hiding policy is that it imposes restrictions only on the composite object itself [NVP98]: It, i.e., the representative and the internal sub-objects, must never hand out references to internal sub-objects objects (in case of absolute hiding), or can pass them out only to methods known not to conserve them (in case of relaxed hiding). All encapsulation techniques export constraints into the context that govern the use of inbound references returned through the composite object's interface.

3. EXCESSIVE CLONING AND FORWARDING is the fundamental weakness of all hiding notions [BC87, Bos96, KT99, HLS00, MP01]:

Cloning. It is, for example, impossible to implement a map's `getEntrySet` operation by returning the internal `Set` component in which the entry `Pairs` are stored (in cases where such a component exists). Instead, a clone of the `Set` component has to be created and returned. The cloning solution has several disadvantages: First, there is the obvious *inefficiency* of cloning large internal objects. Second, it is *no general solution* since cloning cannot be sensibly defined for all objects: Can there be a clone of a Singleton object [Ga$^+$95], or a BankAccount object? Third, cloning means to duplicate mutable data even in cases where *sharing is desired* because it is a conceptual part of the application. For example, if clients of a Company object want to know its address, returning a reference to the Address component can provide for address information that never becomes "out-of-date" [HLS00] "without the need to propagate the changes to the clients" [MP01] and without having to "cope with duplicate data and keep track of conceptual object identity" [KT99]. Finally, it is an *inconvenience* to the programmer to check if an automatic replication of memory structures suffices or a special cloning procedure has to be written for object like GUI-Windows, Threads, Files, reference-counting smart pointers, Semaphores, etc. And the manual implementation of cloning (and a change propagation strategy) is a potential source of new *programming errors.*

Forwarding. Alternatively, maps could themselves provide all set-operations on the entry-set which the client might need: `entryset_contains`, `entryset_elements`, and perhaps also `entryset_Add` and `entryset_Remove`. `MapImp` maps can straightforwardly implement these operations by forwarding the requests to their entry-set component. Also the forwarding solution has its disadvantages [BC87, Bos96]: First, its recursive application produces more and more operations that would unnecessarily *inflate the interface* of the composite [HLS00]: Instead of being able to expose one entry component through an operation `entryset_getFirstEntry`, the map would need operations with the intimidating names `entryset_getFirstEntry_first` and `entryset_getFirstEntry_second`, and maybe `entryset_getFirstEntry_Set` (cf.

the stick-figure object in [BC87]). Second, the deeper nested the composite object is, the longer is the *chain of forwarding* down the composition hierarchy, letting the inefficiency increases linearly with the depth of the accessed component. Third, the inlining of interfaces introduces a *coupling* between the definitions of *interfaces* `Map` and `Set`: When `Set` operation `Add` is renamed to `Insert`, consistency demands to rename `entryset_Add` to `entryset_Insert`. Fourth, the recursive inlining "completely flattens the part hierarchy and so removes the conceptual advantage of factoring knowledge in an intuitive manner" [BC87]. Even if the programmer still recognizes behind names starting with `entryset_` the notion of a map with a set of entries, he cannot make use of standard operations with `Set` parameters for further analyzing the map's entry-set, handing it to a print procedure, subtract it from another set, creating a multiset from it, etc.

Note that the hiding of *fields* did not pose all these problems: Where needed, it is possible to offer operations `get-`x and `set-`x for accessing their values with mimimal performance penalty, with little chance for programming errors, and with negligible cluttering of the interface. When the representation of the abstract data in the fields is changed, e.g., from calendar dates with three integers to dates with two integers, operations `get-`x and `set-`x can be reimplemented accordingly. No remaning affects the client. The client does not notice a thing (module transparency/substitutivity).

4a. STRICT ENCAPSULATION means that the outside neither changes the set of components nor their state. The latter aspect was concretized as the policy of ***representation encapsulation***, and enforced in the Universes system [MP01]: All inbound references must read-only, i.e., cannot be used to modify the target.

Giving up on the aim of hiding makes it possible to let clients of a Company object directly observe its Address object for up-to-date address information without change propagation. It enables the programming of Iterator and Visitor objects, as long as they are not used for modifying the structure they are traversing/visiting, and enables the efficient implementation of `getEntrySet` and `union`.

4b. RELAXED ENCAPSULATION allows the outside to change the set of components and their state iff this is done in a contained way. This policy has not been described in the literature yet. In analogy to Minsky's concept of hiding, it may be concretized as allowing a change only while control is in the representative, i.e., only through its methods. W.r.t. change of the composite's *state-representing* components (§2.5), this is covered by our policy of ***state encapsulation***: The composite object's state changes only through its own methods. Since other, *behavioral* components are parts of the representative's methods, they exist only while it is control; relaxed encapsulation follows. A restricted form of relaxed encapsulation, where the writable inbound references are contained in calls (not in dominated objects), is enforced by the No Abstract Aliasing methodology [DLN98].

Relaxed encapsulation additionally supports efficient `unifyWith` and Visitor which modify the visited structure.

5. VARIATION: SANDWICHES. Several authors allow inbound references that are neither read-only nor contained, if they cannot be captured in fields. This property has been generally called ***Sandwiches*** [GTZ98], or ***containment invariant*** [Cla01]. ***Islands*** [Hog91] and ***Balloons*** [Alm97] are a special form of Sandwiches without *out*bound captured references. The possibility of the context changing the interior without the representative's control leaves no doubt that here the composite object is not encapsulated any more (for the question of encapsulation it does not matter whether the reference used for the mutation is captured or not).

Nevertheless, one may argue that there is no real problem on the implementation integrity side here: Inbound references not contained by the representative can be created without storage in fields only if the representative returns them (a *simple upward leak* [DLN98]). Hence, under the worst case assumption of arbitrary modifications by the context, the composite object's implementation can be verified and reused in any context. And on the transparency side, the restriction not to capture the inbound reference in a field makes it easier to reason about whether the context develops a dependency on what it sees through this reference. (Even if then there is no modularity between the composite object and its context, the restriction makes it easier to show that the system as a whole works correctly, since inbound reference not captured in fields cannot cause "unpleasant surprises at an arbitrarily distant point in an execution" of an object's method [Hog91].)

6. ENCAPSULATION PROBLEM: MUTATING ITERATORS. If a `Set` object creates and returns a structure-sharing iterator object, then this is described as "encapsulating" from the `Set`'s client, and within the iterator object, the reference (or the access) to the internal structure [Ga$^+$95]. In many cases, iterators may even be used—in excess of the design pattern—to modify the internal structure: For instance, the iterators over Java's standard collections have a `remove` operation. Intuitively there is no problem here, despite the undoubted breach of the composite `Set` object's encapsulation by the possibility of the iterator modifying the `Set`'s interior in a non-contained way. The crucial point is that the inbound reference is stored in an object which the `Set` object created itself, so that the iterator's implementation class is known (and can be inspected for verification). A worst case analysis of what the iterator could possibly do with the given reference can show that the `Set`'s inner working is never in danger. The `Set` object is reusable is any context (with an iterator implementation), no matter what the context does to the `Set`'s interior through the iterator (as long as it uses the iterator's operation interface).

AliasJava's "capability-based encapsulation model" supports relaxed hiding with mutable iterators and similar objects [ACN02]: Object classes can be parameterized with "ownership parameters" that specify the composites to whose components (in addition to its own components) the instances may have writable references. When an object instantiates a class, it can make accessible its own components and any composite's components to which it has access itself.

7. Encapsulation Problem: Notification Messages. Event-based systems [SG96] and systems with the Observer patterns [Ga+95] are an important class of object-oriented systems: Objects ω register with an event-dispatcher or Subject q for notification about the occurrence of certain events (GUI inputs, state changes, etc.). There is little use for ω having a notification message sent by q if this would not give ω the opportunity of changing its state. But then there is a problem with registering a nested component object ω with an external event-dispatcher or Subject: q's sending of the notifying mutator messages along an inbound reference would circumvent ω's representative. It is unclear how one could not see a violation of encapsulation in this, even though normally there seem to be no adverse effect on the system's modularity—on the contrary, event-dispatching and the Observer pattern are specifically used for decoupling different system parts and improving modularity. Observe that Sandwiches, too, cannot handle this case.

Imposing the policy of encapsulation on such systems without restructuring them, i.e., without changing the object references, can be possible only by adding a ***filtering mechanism*** to the semantics of message passing: In the *composition filter model* [AW+92] and the *layered object model* [Bos96], the representative defines filters for the messages to its components and their subcomponents. Filters are like methods that are implicitly invoked on the representative to decide whether to accept, reject, or reimplement sent messages. This guarantees the representative's control over all changes, and thus appears to be a clear case of relaxed encapsulation. But the problems is that it is questionable on what behavior the clients of a potentially nested component object can still rely on (including invariants and history properties), and how the representative could judge whether messages to abstract components' implementation-specific subcomponents (e.g., an entry-set's nodes) are benign w.r.t. the way how it is using the component (e.g., as a set of key/value pairs).

Similarly, but more coarsely, the ***Object Space*** model [CR00] subjects the delivery of all messages to a *security policy*: The interface object(s) of a runtime component can select dynamically whether messages sent to private objects from objects in a particular other runtime component should be delivered or raise an exception.

8. Varying the Unit of Encapsulation lets the same encapsulation policy mean a different effective property of encapsulation for the composite object. In the extreme, iterators are possible despite absolute hiding if one does not hide the internal storage nodes but places them outside the encapsulation barrier [NVP98]. Mutating iterators are possible despite encapsulation if the nodes are not behind the composite object's encapsulation barrier but behind the encapsulation barrier of the package that contains the set class as well as the iterator class [MP99a]. And mutating iterators are possible despite the Sandwich policy if the iterator is not outside the encapsulation barrier that contains the storage nodes, but is made another interface part of it in a Clarke-style aggregate (§3.4, paragraphs *4/5*) [Cla01]. (Obviously, a mutating iterator can be *simulated* in any proper encapsulation discipline by letting the iterator forward the `remove` message to the `Set` object [MP01].)

Chapter 4

Related Work

Don't you see that the whole aim of Newspeak is to narrow the range of thought?
In the end we shall make thoughtcrime literally impossible,
because there will be no words in which to express it.

George Orwell, "1984" (1949)

Related work was already mentioned in the previous chapters where it applied to different issues (the notion of composite object, units of encapsulation in the runtime system, encapsulation policies). This chapter focuses on the related works themselves, taking now in particular their technical and linguistic aspects into consideration.

4.1 Encapsulation Approaches

1. PROBLEM IDENTIFICATION. Blake and Cook were the first to characterize the problem of composite object encapsulation [BC87]: "When an object is assembled from its parts these parts are no longer independent. A part belongs to the local state of the whole ..." They warned that the common handing out of references to part objects enables clients to modify them in a way violating the integrity of the whole, which "subverts the idea that objects can hide and control their local state."

Looking at objects modeling a parthood hierarchy, like a stickfigure (cf. §2.7), the authors recognized that a hiding policy would be inappropriate and blow up the whole's interface (cf. §3.6). They proposed an encapsulation policy where the whole "mediates" or "censors" the access to the parts it made visible, but give no precise definition. They offered ***compound messages*** as an alternative for returning part references, but no enforcement of a mediated access policy.

Mediation in a sense was built into some higher-level runtime system models by authors approaching composite objects from the system architecture perspective (§2.7). For example, all boundary crossing messages are routed through a special ***forwarding*** or ***filtering mechanism*** of the composite object (cf. §3.6) in de Champeaux's top-down system analysis method [Cha91], in Aksit's language *Sina* with "composition filters" [AW$^+$92], and in Bosch's *layered object model* [Bos96]. Or all boundary crossing messages are routed through ***communication ports*** whose connection is fixed by the enclosing composite object, as in Gangopadhyay and Mitra's executable visual *ObjChart* models [GM93], and in Aldrich, Chambers, and Norkin's *ArchJava* embedding of an architecture definition language into Java [AKC01].

Such architecture-level concepts are too far away from the practice of object-oriented implementation-level programming, are too general if used to encapsulate composite objects, and incur too much runtime overhead if used for all objects.

Let us focus on encapsulation by constraints on object references.

2. HOGG'S ISLANDS [Hog91], were not about composite objects, but a combination of three techniques for "making object interaction more predictable" that are also useful for composite object encapsulation: First, Hogg brought the function/procedure distinction (observer/mutator) and statically checked ***read-only*** references to object-oriented programming (access control). Second, with the help of static checks and a *destructive* read operation, the ***uniqueness*** of certain references (alias control) was ensured, while still allowing to store them in container objects, to retrieve them, and to borrow them to called methods. Third, Hogg was the first to impose a ***structural constraint*** on the object graph that isolated a region in the object graph called an Island: For the transitive closures of so-called *bridge objects*, he ensured a Sandwich policy (§3.6), i.e., there could be no field-captured inbound references to any object reachable from a bridge along field-captured references (also called *"full alias encapsulation"* [NVP98, Cla01]). More precisely, into and out of an Island, there could be only uncaptured references that were read-only or aliases of a unique reference.

On the linguistic side, Hogg contributed a system of ***access mode*** annotations to distinguish read-only references (`read`), unique references with temporary aliases (`unique`), and references without any aliases (`free`) from ordinary references. He gave a complete set of rules for the inference and static checking of modes based on the modes with which variables, parameters, results, and methods were annotated.[1] Unfortunately, Hogg did not define his system formally so that his claims about guaranteed properties cannot be verified. Also the encapsulation of transitive closures, as explained in §3.4, cannot be a general solution for all objects.

3. THE FIRST SYSTEM realizing the vision of encapsulating all composite objects without distinction by constraints on references (after filter-based Sina and communication port-based ObjChart, see *1*) was presented by Kent and Maung [KM95], who introduced several crucial concepts. The authors applied the general notions of information hiding and encapsulation to *container objects* with internal arrays or

[1]To demonstrate the orthogonality of modes to traditional typing (w.r.t. objects' classes) he presented his system in the language Smalltalk without static typing. In order to check method calls without infering the receiver's type, he assumed the modes of parameters, result, and this to be encoded in the method's name. In Smalltalk, this reduces a mode mismatch to a message-not-understood runtime error. A statically typed language would exclude this kind of error, and make mode checking a completely static affair.

linked nodes as components. In place of Hogg's distiction between objects with and without encapsulation, they distinguished a container object's parts from its content. They imposed a structural constraint on the object graph that isolated not transitive closures but only what belongs to the composite object's implementation object expansion, i.e., "***flexible alias encapsulation***" [Cla01]. (While they seem to aim at absolute hiding, they actually achieved only a Sandwich policy, as explained below.)

The major innovation was the notion of ***object ownership***: While previous work required component references from composite to component (as in LOOPS [SB85], ORION [Ki+87, Ki+88], Sina [AW+92], etc.), Kent and Maung's object composition was represented by a hidden *owner* attribute in each object set at creation time. References were classified uniformely in terms of their target's (relative) owner (not heterogeneously in terms of aliasing and access properties): References to top-level objects are references whose target has no owner (the default). Component references are references whose target has the source as owner (annotated with '`private`'). Kent and Maung defined the new class of horizontal internal references between two components, which we call ***co-references***, as references whose target has the same owner as the source (annotated with '`protected`'). Observe that no inbound references can be classified in this scheme. But since it was used only for the references in variables, parameters and results, unstored temporary references could well be inbound.

However, the authors did not believe in the possibility of statically checking their annotations since "object ownership is a run-time notion." Hence they checked the ownership annotations on variables, parameters and results at runtime against the owner of contained references' targets. As observed by Clarke [CPN98], these checks do not prevent the breach of encapsulation through unstored references, as in `x.getPrivate().modify()`. Consequential work will show that completely static ownership systems are possible and can cover unstored references.

The authors also consider generic classes whose formal type parameter T is a placeholder for a class *with* ownership annotation. For example, `private Set<protected Figure>` types a reference from o to a set component ω whose elements are figures that are co-objects *of* ω, and thus components of o. However, this semantics make generic classes rather unintuitive to use: A T-result or parameter in the interface of class `Set` is for o not a result or parameter of type `protected Figure` but of type `public Figure`. And `private Set<private Figure>` would be a useless set object with its own elements as components. Consequential work rectified this.

4. FLEXIBLE ALIAS PROTECTION (FAP), by Noble, Vitek, and Potter [NVP98], was the first convincing proposal of a system for the encapsulation of composite objects and for working with such encapsulated composite objects. It combined a static mode system like Hogg's with the distinction of representation from transitive closure, like Kent and Maung did. FAP addressed the coupling caused through the sharing of mutable state by a two-pronged strategy: On one side, FAP enforced absolute hiding, i.e., the absence of all inbound references into composite objects' representation—called the principle of ***no representation exposure***. On the other side, FAP supported immutable objects, and the independence of container objects from their contents' state—the principle of ***no argument dependence***.

FAP qualified reference types (in declarations and type inference) with ***aliasing modes*** for a five-fold classification of object references w.r.t. ownership, aliasing and access: Mode `rep` marks component references. Mode `var` is apparently useable for any outbound reference (in particular, a reference to a top-level object). Mode `arg` is used for the (outbound) references stored in a container object. Through them, only state-independent, "clean" methods may be accessed, i.e., methods accessing only immutable fields, immutable objects, or clean methods. Mode `free` of alias-free references is "taken directly ... from Islands" to support flexible object creation and composition. Mode `val` marks references to immutable objects like, e.g., a `String`. Variables, parameters, results and type parameters were annotated with these modes (also `this`'s mode was supposedly specifyable, but no syntax is given).

A crucial innovation for the scaleable, flexible internal structuring of composite objects was the subclassification of `arg` and `var` references by ***roles*** in conjunction with an improved semantics for mode parameters: The authors observed that the objects in a container object may play different roles, like a hash-table's keys *vs.* its items (which in object-oriented modeling would be modeled by two different *associations*). Containers are expected not to mix up objects stored in them under different roles—the principle of ***no role confusion***. To distinguish different roles of `arg` and `var` references, these can be annotated with a role. In FAP, generic (container) classes's type parameters are annotated with modes `arg` or `var`, which are usually qualified with a role: `class Hashtable<arg k Hashable, arg i Item>{...}` is a class of hastables with `k` arguments (keys) and `i` arguments (items, aka. values). FAP's "***aliasing mode parameter binding***" makes `rep Array<rep Object>` the type of references to array components whose elements are the components of the composite (and not of the array, as in Kent and Maung's substitution semantics).

The encapsulation policy of absolute hiding is simple and, as the authors explain, avoids exporting into the context any usage constraints on inbound references (like in Islands) since the context can never obtain any. But it also suffers the general shortcomings of hiding elaborated in §3.6, in particular, it excludes the iterator objects so important for using container objects. Also, the entire presentation was only informal, leaving some issues open, in particular concerning mode parameters, that are necessary to verify the mode system's correctness. The piggybacking of FAP's mode parameter binding semantics on the substitution semantics of class parameters is somewhat awkward. These shortcomings will be solved by the next system.

5. OWNERSHIP TYPES (OT) [CPN98] was the first system of composite object encapsulation presented with complete formal definitions (typing rules, interpretation of annotations, encapsulation property) and a proof sketch. The authors Clarke, Potter, and Noble devised it as a formalization of Flexible Alias Protection's encapsulation aspects, at whose heart is "the intuition underlying Kent and Maung." Crucial was the insight that static checking is possible and owner attributes require no runtime representation since the meaning of ownership annotations in each object is fixed for the object's lifetime and ownership is orthogonal to computation.

FAP's `rep` references were reinterpreted as targeting objects owned by the source.[2] `norep` references replace FAP's role-less `var` references as targeting owner-less, top-level objects. A (re)invention is the distinction of references between objects with the same owner (co-references), called `owner` references. It solves FAP's problems with moding `this` and data structure links. OT gave the roles α in FAP's formal mode parameters `var` α a clear meaning as ***ownership parameters***, called "context parameters," and divorced them from type parameters: These can be used in ownership-polymorphic classes as the modes of outbound references to objects whose owner is the object bound to the context parameter before instantiation.

For example, `class Pair<fst,snd>{...}` defines a class of pairs storing a `fst` and a `snd` reference. The static types of references to pair objects are *ownership types* like $t =$ `norep Pair<rep,owner>`. The context-parameter bound class from which a pair object ω is instantiated is an *ownership structure* $\tau =$ `Pair<`$o|o_1, o_2$`>`, which encodes o as ω's owner and o_1 and o_2 as the owners of o's `fst` and `snd` objects. That is, for ω the modes in class `Pair` are mapped to owners as follows: `owner` $\mapsto o,$ `fst` $\mapsto o_1,$ `snd` $\mapsto o_2$. With this substitution σ_ω, all ownership types t' in class `Pair` are interpreted relative to ω as ownership structures $\tau' = \sigma_\omega(t')$. An object ω may be targeted from different objects $q_1, \dots, q_n$ by references typed with different static ownership types $t_1, \dots, t_n$. But all their source-relative interpretations $\sigma_{q_i}(t_i)$ must yield ω's dynamic type τ.

The authors introduced the graph-theoretical notion of *dominator* or *articulation point* they had elaborated in [PNC98], to define the novel, relaxed hiding policy of ***representation containment*** (aka. *owners-as-dominators* [Cla01]): An object q may possess references to o's components iff all paths from the initial object to q pass through o, even if q's ownership status unequivocally classifies it as external to o, e.g., if q is the target of o's `norep Pair<rep,owner>` reference.

Like any hiding policy, OT excludes iterators and other common patterns, as we elaborated in §3.6. An OT-specific technical problem is the ***loss of ownership information*** that prevents support for subclassing:[3] Subclasses must be free to change, like type parameters, also the context parameters of their base class, in particular. But the subsumption of `norep Pair<rep,owner>` under supertype `norep Object<>` would hide the ownership information necessary to guarantee the target's domination by the source. These shortcomings were solved or alleviated in the three subsequent systems.

6. Clarke's Calculus. Clarke's dissertation [Cla01] is the most thorough work so far, a foundational work on the isolation of regions in the object graph with several technical innovations. Clarke generalized the Ownership Types system to cover the

[2]Actually, each object o was said to *own* a protection domain D_o—called its *rep context,*—that holds or—in the authors' terminology—*is the owner of* o's components.

[3]In retrospect, already FAP seems to suffer from this problem. OT merely brought it to light.

missing language features and make it more flexible, and he reformalized it as an object calculus based on Abadi and Cardelli's sigma calculus [AC96].

The decisive step towards more flexibility was to loosen the connection between the structure of object composition and the nesting of protection domains, the ***ownership contexts***: As in OT (cf. footnote 2), each object o stores its components in a unique ownership context D_o, called o's *rep context*, but now several objects can use the same context for their components. As in OT, o's rep context D_o is nested to the context D in which o is stored, but now it can be several nesting levels deeper. Each object is consequently characterized and typed by two ownership properties: the context D which contains, or "owns," o, and the rep context D_o which contains, or "owns," o's components. This allowed Clarke to put encapsulation barriers around composites with private and public components, even around aggregates of several composites with all their representatives and public components as interface objects. (Their internal context nesting structure distinguishes them into the principal-with-proxies aggregates and collective aggregates of §3.4.)

Also Clarke introduced ***context polymorphic*** methods, with context parameters bounded above or below by a context. They allowed to shift from the problematic context-parameterization of OT's classes to a parameterization of the corresponding constructors, so that subclassing became easy to integrate. And the lack of a `free` mode could be (partially) compensated for by parameterizing methods creating, e.g., iterators, with the context to hold the new object.

Additional flexibility was obtained by switching to the Sandwich policy: Citing Almeida [Alm97], Clarke deemed dynamic aliases acceptable "since they are essential for implementing real programs." (Clarke's "containment invariant" is defined over the store like OT's "representation containment," but in his substitution-style calculus method-local references do not appear in the store.) Methods returning a reference to a component (`rep` results in FAP or OT) can be called by other objects using the `expose` construct to create the needed (temporary) name for the result's context.

Apart from formal matters, Clarke's work is harder to evaluate since it is a calculus, not a programming language, and since the most complex examples he elaborates in his calculus are cars and linked list. Switching between different variants of his calculus, Clarke shows how wanted behavior can be programmed and unwanted behavior be excluded. This approach makes it hard to judge which variant could be the best compromise. The calculus with unique interface objects ("flexible alias encapsulation") is too restrictive to be generally useful: It suffers from the principal problems of a Sandwich policy explained in §3.6, like the exclusion of iterator objects. But all variants with multiple interface objects ("fractal alias encapsulation") suffer from the lack of limitiation on the creation of additional interface objects, both conceptually and technically: Clarke does not manage to give a general intuition what abstraction his multiple interface aggregates represent; there is no guideline for how much should (not) be included in an aggregate; it seems, whenever access to a rep context is desired, a new interface object can be included. The calculus allows this, but this is dangerous since any unsafe or malicious code could get unconstrained access to the representation stored in a context through a corresponding interface object it created to this end—Clarke calls this a "vampiric" interface object. There is no protection by a read-only limitation for all the additional interface objects.

7. UNIVERSES, by Müller and Poetzsch-Heffter [MP99a, MP01], is the first system that enforces a policy of encapsulation without hiding which others had only offered to support by transitive `readonly` references [KT99, HLS00], or enforced by specification [DLN98]. It is based on Ownership Types but technically less ambitious since the authors use it in the context of modular verification. The authors reevaluated (in more detail in [MP99b]) what the real problem is with sharing mutable objects, not limiting their attention to objects reifying wholes in parthood hierarchies [BC87] nor to container objects [KM95, NVP98]. Similar to FAP, but with more precision, they identified the problem with outbound references to lie in the possible *dependency* of the composite's abstract values and invariants on external objects' (mutable) state, and the problem with inbound references to lie in possibility of invariant-breaking *modification* through them.

On the technical side, Universes simplify OT by replacing OT's problematic context parameters by ***runtime ownership checks*** (which their verification technique can make superfluous in most cases). Only three classes of references are distinguished: references to objects owned by the source (mode `rep`), references to objects with the same owner (the default), and references making no statement about ownership that can connect any two objects (mode `readonly`). Through the third class of references, dependency of the composite's abstract values on the target's state is prohibited and modification of the target's state is prohibited. In conjunction, these two restrictions mean that the abstraction, e.g. abstract data structure, represented by the composite object can change state only through the composite's operations. References are stored in container objects as `readonly` references; the target's owner can retrieve the `readonly` reference and convert it back to a `rep` reference—which is where ownership is checked dynamically—and then modify the target.

The authors presented the type system aspect of Universes not in the standard type-theoretic formalism. The obvious shortcoming as a stand-alone type system without a verification technique to fall back on, is the reliance on runtime ownership checks and thus the need to represent ownership at runtime. Also, Universes prevent flexible object creation and composition by fixing new objects' owner always to their creator. These are not unsolvable problems, and they will be solved in JaM.

8. ALIASJAVA. Aldrich, Kostadinov, and Chambers [ACN02] treat object ownership and ownership parameters to classes in a manner that seems much closer to a concretization of Flexible Alias Protection than its official formalization by Ownership Types. They characterize their AliasJava system as ***capability-based*** (not ownership-based). It combines aliasing annotations with ownership annotations to make aliasing patterns explicit, support reasoning about ownership, and enforce a relaxed hiding policy. The authors were the first to develop a constraint-based algorithm for inferring the new annotations, and the first to report on the usability of their system for real-world software like Java's standard library, and the circuit layout application Aphyds (12,500 lines of code).

AliasJava classifies references five-fold by aliasing properties: `shared` references are ordinary references not aliased by `unique` and `owned` references, i.e., targeting top-level objects. `lent` references are "time bounded aliases" (e.g., of `unique` and `owned` references) that can neither be captured in fields nor returned, i.e., ***borrowed references*** (cf. *9* below). `unique` references have only `lent` aliases. `owned` references make the source the unique "owner" which "controls who may access" the target object: They can be aliased only by other `owned` references of the owner, by `lent` references, and by references classified by an ownership parameter that is bound to the owner. A class can have ***ownership parameters*** that, for instantiation, must be bound to the creator or it ownership parameters (ownership parameters flow along creator relationships). An ownership parameter α bound to object o grants the class's instance the right to possess an α-reference to an object targeted by o's `owned` references and other objects' β-references with β bound to o. Like FAP, AliasJava does not distinguish co-references; the mode of `this` is `lent` by default, but can be specified explicitly to `shared`, `unique`, or an ownership parameter.

AliasJava solved OT's iterator problem: A container object can grant its iterator full access to the representation by instantiating it from a class with ownership parameter bound to `owned`. This effectively extends the encapsulation barrier to include the iterator as another interface object in a principal-with-proxies aggregate (§3.4). It is the one case of adding an interface object that was identified as "safe" by Clarke [Cla01]. And AliasJava solved OT's problem with lost ownership information by recovering it dynamically when references are cast to subtypes with more ownership parameters: Since objects are instantiated from classes with ownership parameters bound to owners, Java's runtime check against the target's class must in AliasJava also ***runtime check the ownership parameters***.

The obvious shortcoming of AliasJava is the reliance on runtime checks of ownership parameters and thus the need to represent ownership parameters at runtime. This is not just an overhead for "heavyweight" objects, as the authors write: Also small data structure components like `Pairs` and `Nodes` have ownership parameters. As presented, ownership parameters must be bound to an owner, so that ownership-polymorphic container classes cannot be used for containers of `shared` objects.

9. UNIQUENESS AND BORROWING. Minsky [Min96] introduced a simple but effective form of hiding component objects that is independent from the previous ones (*3-8* above): The composite possesses the *only* reference(s) to its components (which it may "lend" to other objects for the duration of its methods). This ensures a relaxed hiding policy. In Minsky's case, references in `unique` fields have no alias, but are effectively lent to others by passing fields as by-reference parameters. (By reading the value out of them, the receiver is able to take over the component object.) Islands had `unique` references with uncapturable aliases earlier, but not for use as component references. Greenhouse and Boyland's *Object-Oriented Effects System* [GB99] keep unique component references in `unshared` fields. `borrowed` aliases can be created in the composite's methods, but cannot survive since they can neither be captured in fields nor returned (a companion paper [Boy01] describes the details). Detlefs, Leino and Nelson's specification-based *no abstract aliasing method* [DLN98] keeps component references in "*pivot*" fields. They can have aliases in other fields of the composite and borrowed aliases, as well as uncaptured, "read-only by specification" aliases from the time before the component's capture in the pivot field.

Uniqueness-based encapsulation disqualifies itself as a general solution by its limitatation of composite objects' internal structure to a *tree* (with a bounded degree of branching since composites have only a fixed number of fields to hold their component references). This excludes (double) linked lists and rings, and requires preventing composites from giving their components capturable back-links to themselves.

Borrowing is independent from uniqueness and makes sense also in combination with ownership to grant method-contained external access to the interior. AliasJava supports this through `lent` inbound references. (Clarke's context-polymorphism for methods is not them same; it does not prevent the capture of inbound parameter references in new "vampiric" interface objects.)

Uniqueness has a better use in cleanly moving new objects from their creators to their final owners. Islands and Flexible Alias Protection [Hog91, NVP98] support this through the mode `free` of alias-free references, and AliasJava through the mode `unique` of references with only `lent` aliases. In the context of their specification system, Leino et al. are able to relax uniqueness to "*virgin*" references [LS97, DLN98] which can have any number of dynamic aliases, but never had an alias captured in a field. The above described drawbacks of uniqueness (no linked lists, no back-link in their components) now apply to new objects before fixing their owner. But even this is not necessary, and JaM will show that new objects can be moved safely even with captured aliases. (The necessary weak uniqueness property is more difficult to describe but no more difficult to enforce than Island's freedom.)

10. MECHANISM, NOT POLICY is supported by Kniesel [Kni96] and by Boyland, Nobles and Retert [BNR01]. Kniesel reanalyzes the notion of encapsulation in object systems and offered for the protection of the reachable state a system of *access rights.* He distinguished the right to read, to write, to call functional methods, to capture the reference in fields, and to transfer the reference to other objects. Boyland, Nobles, and Retert designed their "capability system for pointers" [BNR01] to bring order into the many reference annotations that have been proposed in the field. It encoded them as combinations of the right to read, to write, and to test identity of the target, the guarantee for exclusivity of each of these rights, and finally an "ownership" capability which permits one to revoke rights on other references to the same object and protects from the revocation of rights by others. These systems enforce no encapsulation policy since the extent of composite objects cannot be specified.

4.2 Discussion

The reviewed systems for composite object encapsulation by restricting references cover all encapsulation policies (§3.6). The three most recent systems support external iterators over encapsulated container objects: Universes through `readonly` references, Clarke's calculus through multiple interface objects, and AliasJava through access granting ownership parameters. However, Clarke cannot prevent "vampiric" interface objects, and the other two need runtime ownership checks.

In all systems, the notion of *ownership* is, or could be, applied to intuitively describe the special relation which any encapsulating composite (or its representative) has towards its encapsulated components. Despite superficial, linguistic similarities between the systems, two fundamentally different directions can be distinguished:

In the ***ownership-based*** systems of Kent and Maung, OT, Clarke's calculus, and Universes, objects have an owner attribute (with runtime representation or not). The information based on which the permissibility of access or references is judged lies in the respective object (in form of the owner attribute). Modes like `rep` in the static types of references are *descriptive statements* about runtime ownership that can be correct or not (with the owner attribute as the primitive basis).

In the ***capability-based*** systems of AliasJava, of the uniqueness-based approaches, and presumably also of FAP, the object references are labeled with a mode (with runtime representation or not). The permissibility of access or references is judged based on information in the access-establishing references (in form of the mode label). A reference is what the access control literature calls a *capability* [CJ75, BNR01]. Object ownership is just a notion derived from (appropriately labeled) references: without references, no ownership. Modes like `rep` in the static types of references are *declarative definitions* of ownership relations that are not correct or incorrect but can only be consistent or inconsistent with the other declarations in the system.

Technically, JaM will extend the capability-based approach to a ***reference path-based*** approach by moving ownership parameterization from objects' classes to references' modes. *All* ownership information is removed from the objects, thus solving the loss of ownership information problem of subclassing. Roles `fst` and `snd` are not placeholders for reference targets' owners, but uninterpreted type tags on object references. Similar to class tags on objects distinguishing instances from equally defined classes, role tags distinguish references of different roles or—as one would say in object-oriented modeling—of different ***associations*** [OMG00]. The available roles are not limited by a parameter list, nor by the references targeting it. The ownership parameterization of the references by ***correlations*** only configures the source's mode-interpretation of the association roles on the target's side. It is used for the translation of exchanged references and the derivation of ownership from paths of object references. Consequently, in JaM, the targets of α-references need not have a particular owner; clients can store in container objects also their `free` and `read` references (and `lent` references, had we included this class of references).

Chapter 5

The Base-JaM Fragment

Writing can be either readable or precise,
but not at the same time.
Bertrand Russell (1872-1970)

The formally precise description and analysis of a full-featured real-life programming language like Java is a complex undertaking. In the investigation of new features for programming languages, it is customary to reduce complexity from the side of the base language by the omission of non-fundamental features and the explicit syntactic representation of implicit operations ("desugaring"). In order to make the precise definition of Java with Modes (JaM) and the demonstration of its properties more easy to digest, we will look at a further simplified version first: ***Base-JaM*** is a simplified and desugared Java subset with a simplified mode system that omits association roles and correlations. The extension to the full mode system, with the complex treatment of association roles and correlations, is postponed to the next chapter.

After a first overview (§5.1), the introduction of base-JaM starts with the *untyped* language in order to focus on semantic aspects: First, a standard operational semantics (§5.2), then JaM's higher-level view with object graphs, paths, and composite objects (§5.3). Type- and mode-system are then added to match the semantics and define the *legal* base-JaM programs (§5.4). Proofs for important properties of JaM execution states and steps will be developed: the standard property of type correctness, and, based on it in §5.5, JaM's new higher-level properties (state encapsulation, control of mutator executions, uniqueness of ownership). Basic familiarity with Java-like object-oriented languages and their formal treatment is assumed.

5.1 Base-JaM Programs

1. SUMMARY OF SIMPLIFICATIONS. The notable simplifications from Java to JaM and base-JaM are the following:

- (Base-)JaM omits all non-basic object-oriented features like packages, static members, user-defined constructors, overloading, nested classes, exceptions, and arithmetics. The entire program is considered one package, there is no visibility other than implicit package-privacy. Object references are the only first-class values, and their types the only types in the program. The number of statement and expression types is reduced to a minimum.

- (Base-)JaM does not go beyond *class-based* object-orientation: There is no inheritance and no subclass-polymorphism, and consequently, there are neither Java interfaces nor abstract classes. (There is, however, *mode*-conversion in assignment and parameter passing—a kind of "ad-hoc polymorphism" like the conversion between different number formats [CW85].)
- (Base-)JaM makes the read access to a variable explicit. Like in the AliasJava formalization [ACN02], a *destructive read* access is provided and distinguished from the normal, non-destructive one in order not to complicate the formal type system by the integration of a live variable analysis *à la* Boyland [Boy01]. In a full implementation of JaM, such an analysis would ensure that a `free` variable from which a `free` reference was read is overwritten before it can be read again.
- Base-JaM simplifies JaM's full system of modes: Association modes $\alpha \in \mathbb{A}$ are omitted together with the correlations that configure (in other modes) the extension by association paths. Hence modes in base-JaM are just the base-modes `free`, `rep`, `co`, and `read`.

2. SYNTACTIC DOMAINS. The simplifications reduce the syntactic variability of (base-)JaM programs to a manageable size so that the grammar of base-JaM can be shown succinctly in figure 5.1. There are three JaM-specific additions to the Java subset, which are underlined and will be explained further below.

A program p is a sequence of class definition modules.

A class module D starts with the keyword `class` followed by the class name c and, enclosed in curly braces, a sequence of field declarations and methods (operation implementations).[1] (Base-)JaM adds `obs` or `mut` in front of each method.

A type term t in declarations of fields, results, parameters, and local variables can in JaM's Java subset only be an object reference type. All the class names used as object reference types (but not the classes named for instantiation in `new`) are qualified in (base-)JaM with a (base-)mode μ.

A statement s in a method's body can be an assignment, `return` with return expression, else-less `if`, `while`, or a sequence of these. Due to the lacking support for Boolean expressions, the guards in `if` and `while` are just direct comparisons between two object reference-valued expressions for equality or inequality.

An expression e in a statement can be the `null` reference, a read access to a variable ν (field, local variable, or method parameter), an object creation expression

[1]For simplification of the syntax specification, the *commas* separating parameter declarations in methods and parameter expressions in operation calls are omitted, although they will occur later in discussed program terms. To be formally correct, the commas should be specified in the syntax.

program	$p \in$ P ::=	D*
class defn.	$D \in$ D ::=	`class` $\mathbb{C}$ { (T *Id*;)* Mth* }
method	Mth ::=	$\underline{\mathcal{K}}$ T *Id*((T *Id*)*) {(T *Id*;)* S}
method kind	$\kappa \in \underline{\mathcal{K}}$ ::=	$\underline{\texttt{mut} \mid \texttt{obs}}$
type term	$t \in$ T ::=	$\underline{\text{M}}\ \mathbb{C}$
base-mode	$\mu \in \underline{\text{M}}$ ::=	$\underline{\texttt{free} \mid \texttt{rep} \mid \texttt{co} \mid \texttt{read}}$
statements	$s \in$ S ::=	S S | N=E; | `return` E; | `if`(E Ψ E){S} | `while`(E Ψ E){S}
relational op.	$\psi \in \Psi$ ::=	`==` | `!=`
expression	$e \in$ E ::=	$\underline{\texttt{val}}$(N) | $\underline{\texttt{destval(N)}}$ | `null` | `new` $\mathbb{C}$`()` | E$\underline{\Leftarrow}$*Id*(E*)
variable	$\nu \in$ N ::=	*Id* | `this.`*Id*

Given identifier sets:
- classes $c, d \in \mathbb{C}$
- variables, fields, methods $x, y, z, f \in Id$ (includes `this`, excludes `null`)

Figure 5.1: Syntax of base-JaM programs

(`new`), or an operation call.[2] Keywords `val` and `destval` are added to make the read access (non-destructive and destructive, respectively) explicit.

Observe that, as in Smalltalk, it is ensured through the syntax of field access that objects can only access their own fields.

3. Meaning of Constructs. The meaning of all original Java constructs is unchanged and should require no explanation.

Added '`val`' and '`destval`' make explicit the, respectively, non-destructive and destructive read access to a variable ν. Destructive access resets the variable to `null` after having read the value out of it. Non-destructive access copies the value out of it. In case of a `free` reference value, the mode of the copy is weakened to `read`.

Added '`obs`' or '`mut`' declare a method as, respectively, an observer or mutator, i.e., a method which guarantees not to change, or offers to change, the composite state. The type system will ensure that `obs`-methods cannot change non-`free` objects' states. It does not ensure that `mut`-methods indeed make some change.

The added modes $\mu \in \mathcal{M}$ in the types $t = \mu\ c$ declared for object reference-valued variables, parameters and results fix the modes of these reference. Through the mode-controlled combination of references to moded paths (defined formally in §5.3.2), the programmer can indirectly define the structure of object ownership (or composition) and place the state representation into the representative's sanctuary:

- By giving a variable, parameter or result of object o the mode `rep`, the corresponding reference to an object ω is defined to mean that o is ω's owner and includes ω in its sanctuary. In the execution of legal base-JaM programs, it is ensured that ω has no other owner (the Unique Owner property).

[2]The dot in operation call expressions has been replaced by '$\Leftarrow$' for distinction from field access.

- By mode `free`, the reference is defined to mean that o is ω's owner and that ω is in no sanctuary. In legal base-JaM programs, it is ensured that ω has no other owner and indeed belongs to no sanctuary, and that no second `free` reference can target ω or start `free` reference paths to ω (the Unique Head property).
- By mode `co`, the reference is defined to mean that ω and o have the same owner (which is unique in legal base-JaM programs) and belong to the same sanctuaries (the owner's sanctuary and enclosing sanctuaries).
- By mode `read`, the reference is defined to have no meaning for the target's ownership and sanctuary membership. A secondary meaning entailed by the enforcement of composite state encapsulation is the restriction of access to calling observers (`obs`-qualified operations) on the target—hence the mode's name '`read`'.

All this will be defined more precisely in §5.3.2 based on a formalization of the notion of object graph.

4. Program Meaning. Like all object-oriented programs, (base-)JaM programs mean, on one hand, a set of definitions of named classes of objects (*static meaning*) and, on the other hand, a computational process in an object system constituted by these classes's instances (*computational meaning*):

Each module D in a program p defines a name $c \in \mathbb{C}$ for a new class of objects. It defines the names x_i and range types τ_i of their fields (the *instance record type* of c-instances), and defines what their methods are (the *method suite* of c-instances). In legal programs, there are no two modules defining the same class name, and no two definitions of the same field or method name within a class module (no overloading).

Since JaM has no static method `main` as Java, program execution—the computer's realization of the program's computation meaning—is defined to begin with the call of the `main` method on a new instance of the last class in the program. That is, the meaning of p as a computational process is the evaluation of the term `new` c_n`().main()` in the context of p's definitions (static meaning), where c_n is the name defined by the last class module D_n in p.

5.2 Formalization of Program Meaning

A precise, formal definition of the execution of JaM programs is needed as a basis for proving that the proposed mode system guarantees composite state encapsulation, i.e., that during program execution the representative controls each and every state change in its current state representation. Various formalizations for more or less large subsets of Java have been provided by different authors in order to reason about type safety [IPW99, Sym97, DE97, Ohe01]. While adequate for reasoning about the outcomes of computations, these formalizations are not so well suited for reasoning about the change steps and invariants during a computation.

This section develops a formalization of the execution of base-JaM programs in the style of a so-called *structured operational semantics*, *small-step semantics*, or

$$\frac{p \equiv D_1 \ldots\ \texttt{class}\ c_i\ \texttt{\{}\overline{t_i\ x_i}\texttt{;}\ \overline{\kappa_i\ t_i\ f_i\texttt{(}\pi_i\texttt{)\{}b_i\texttt{\}}}\texttt{\}}\ \ldots\ D_n}{\vdash \mathit{FldsMths}(c_i) = \langle\{\overline{x_i : \mathsf{ref}\ t_i}\}, \{\overline{f_i \mapsto \kappa_i\ t_i\ f_i\texttt{(}\pi_i\texttt{)\{}b_i\texttt{\}}}\}\rangle}$$

Figure 5.2: Program's meaning as defining object classes

reduction semantics. Such a semantics defines the *stepwise* transformation (reduction, evaluation) of program terms in the context of a stack $\vec{\eta}$ of environments for the ongoing method invocations, a store $\mathfrak{s}$ for the variables' values, and an object-map om to describe the objects in the system (their fields and their methods).

Specifically for accommodating reasoning about mode and composite objects, this formalization contains three non-standard features: First, an object reference $o \xrightarrow{\mu} \omega$ from the object (identified by) o to the object (identified by) ω is formalized not simply by the object identifier ω (in o's fields or methods) but by the triple $\langle o, \mu, \omega\rangle$, called a *handle.* Second, the *call-links*, i.e., the references through which on-going method invocations were made and which will return the result back to the caller, are recorded in the computational state. Third, in order to make explicit what the current object graph is and how the computation steps change it, object graphs will be included as an explicit fourth context $\mathfrak{g}$ of the term's reduction, and manipulated explicitly (in parallel to the handles) in the term reduction rules.

1. Static Meaning. The meaning of program p as a set of definitions is formalized by the tuples $\mathit{FldsMths}(c_i) = \langle\Gamma_i, F_i\rangle$ of the instance record type Γ_i and the method suite F_i of the instances of each class c_i defined by some class module D_i in p. In JaM without class inheritance, this meaning is easy to extract from the program as figure 5.2 shows. The instance record type Γ_i is the collection of the names x_i and range types τ_i of the fields defined in D_i to the type assignment $\{\overline{x_i : \mathsf{ref}\ \tau_i}\}$. (ref is added to the fields' types since the fields are not τ_i-values but τ_i-variables, i.e., x_i denotes a location in the store that contains a τ_i-value.) The method suite F_i is a mapping from operation names f to the corresponding method definitions in D_i. (By not expanding the (computational) meaning of the methods, matters are simplified compared to a denotational-style semantics.)

2. Computational Meaning. The meaning of program p as a computational process is formalized as a sequence of reduction steps $e, \vec{\eta}, \mathfrak{s}, om, \mathfrak{g} \Longrightarrow e', \vec{\eta}', \mathfrak{s}', om', \mathfrak{g}'$ transforming the term e in the implicit, static context of the program p, and in the dynamic contexts $\vec{\eta}, \mathfrak{s}, om, \mathfrak{g}$ (environment stack, store, object-map, object graph). It starts with the start-up expression $e_0 =_{\text{df}}$ `new` c_n`().main()` in the initial contexts $\eta_0, \mathfrak{s}_0, om_0, \mathfrak{g}_0 =_{\text{df}} \emptyset^{\mathsf{obs}}_{\langle\mathsf{nil},\mathsf{read},\mathsf{nil}\rangle}, \emptyset, \emptyset, \emptyset$:

$$\begin{aligned}\texttt{new}\ c_n\texttt{().main()},\ \emptyset^{\mathsf{obs}}_{\langle\mathsf{nil},\mathsf{read},\mathsf{nil}\rangle}, \emptyset, \emptyset, \emptyset &\Longrightarrow e_1,\ \vec{\eta}_1, \mathfrak{s}_1, om_1, \mathfrak{g}_1\\ &\Longrightarrow e_2,\ \vec{\eta}_2, \mathfrak{s}_2, om_2, \mathfrak{g}_2\\ &\Longrightarrow \ldots\end{aligned}$$

The following two sections explain first the contexts and then the reduction steps.

5.2.1 Computational States and Values

3. STORE-BASED RUNTIME MODEL. The standard basis for the definition of the computational meaning of a term in languages with mutable variables (i.e., computations in the imperative paradigm) is made of an "environment" and a "store" (since Strachey and Burstall's work on pointers in the late 1960s [Gor00]): The ***store*** $\mathfrak{s}$ is an abstract model of the current memory state which maps locations $\ell \in \mathcal{Loc}$ (abstract memory addresses) to the values $v \in \mathcal{V}$ currently at these locations. Each ***location*** in the store can represent a program variable (local variables in method invocations, fields in instance records, etc.). The identifiers $x \in Id$ of local variables valid in term e are mapped by the ***environment*** η to the store locations ℓ holding their current values. In this model, the current environment changes during execution when blocks with local variable declarations are entered or exited. And the store changes when variables are initialized or updated by assignment.

For object-oriented programs, also the identifiers $x \in Id$ of the objects' fields (instance variables, slots) must somehow be bound to the store locations of their current values; there must be a "field-environment" ϱ_o for each object o. Since in Java, unlike in C++'s *memory object* model, objects are not variables, they are described in a separate component of the computational state: The ***object-map*** maps object identifiers to the field environment and method suite of each implementation object.

In the context of environments, store and object-map, each reduction step replaces in the term e one subterm, the *redex*, by another term. In particular, locations $\ell \in \mathcal{Loc}$ are substituted for identifiers x (using η) and for field names `this`.x (using $\varrho_{\texttt{this}}$) as "l-values", variables' values $v \in \mathcal{V}$ are substituted for read access expressions (using $\mathfrak{s}$) as corresponding "r-values", and method bodies are substituted for operation call expressions (using om). Through these substitutions, the transformed terms are not just the statements and expressions of the program syntax, but belong to the larger category R of ***runtime terms***. Their syntax (figure 5.3) adapts that of program statements and expressions by replacing occurences of S and E to R, except in the non-initial statement of a sequence, the then-branch of an `if` statement, and the body and condition of a `while` statement, since evaluation never takes place there (cf. paragraph *7*). Each nesting level i of method bodies expanded in the runtime term needs its own environment η_i for associating the identifiers of local variables in it with the corresponding location.

The reduction of terms will consequently be defined w.r.t. the following three contexts (see figure 5.3):

1. A dynamic stack $\vec{\eta}$ of ***environments*** $\eta_i \in Env$ handles the identifiers at each method invocation nesting level. Formally, this stack is a sequence $\eta_1, \ldots, \eta_n$ with η_1 as the bottom and η_n as the top element. The extension of $\vec{\eta}$ by a new top (or, in the context rules, by a to-be-discarded bottom) η' will be written $\vec{\eta} \bullet \eta'$

environment	$\eta_h^\kappa \in Env$	$=_{\mathrm{df}} (Id \nrightarrow \mathcal{L}oc) \times \mathcal{K} \times \mathcal{V}$
store	$\mathfrak{s} \in \mathfrak{Store}$	$=_{\mathrm{df}} \mathcal{L}oc \nrightarrow \mathcal{V}$
object-map	$om \in Omap$	$=_{\mathrm{df}} \mathbb{O} \nrightarrow ((Id \nrightarrow \mathcal{L}oc) \times (Id \nrightarrow Mth))$
object graph	$\mathfrak{g} \in \mathfrak{Graph}$	$=_{\mathrm{df}} \mathbf{N}^{\mathbb{O}\times\mathcal{M}\times\mathbb{O}}$
runtime term	$e \in R$	$::= R\ \mathrm{S} \mid R$`=`R`;` $\mid$ `return` R`;`
		$\mid$ `if(`$R\,\Psi\,R$`){S}` $\mid$ `while(E`$\,\Psi\,$`E){S}`
		$\mid$ N $\mid$ `val(`R`)` $\mid$ `destval(`R`)` $\mid$ `null` $\mid$ `new` $\mathbb{C}$`()` $\mid R{\Leftarrow}Id(R^*)$
		$\mid \mathcal{L}oc$ — *location of a variable (l-value)*
		$\mid \mathcal{V}$ — *expression value (r-value)*
		$\mid \ll R \gg$ — *inlined executing method*

with $\mathcal{L}oc$, $\mathcal{V}$ from fig. 5.5; S, E, N, Ψ, $\mathbb{C}$, *Id* from program syntax

Figure 5.3: Runtime model

(or $\eta' \bullet \vec{\eta}$), using standard sequence concatenation '$\bullet$'. In order to formalize the integrity property Mutator Control (Path) (§5.3.2), actual environments η are extended to η_h^κ by annotating them with the corresponding method's kind κ and the call-link $h \in \mathcal{V}$ through which the method was called. The call-link is saved in the environment since it is still needed to explain the result's return to the caller and must not completely disappear from the system before that.

2. A changing ***store*** $\mathfrak{s} \in \mathfrak{Store}$ maps locations $\ell \in \mathcal{L}oc$ to the values $v \in \mathcal{V}$ currently at these locations. In base-JaM, these values are always "handles," the formalization of object references introduced below.
3. A growing ***object-map*** $om \in Omap$ that maps identifiers $o \in \mathbb{O}$ of created objects to object "values": a field environment ϱ_o (mapping field names to locations), and a method suite F_o (mapping operation names to methods).[3]

Additionally, the reduction rules update in parallel an object graph $\mathfrak{g} \in \mathfrak{Graph}$ as a high-level model of the objects' interconnections by object references. This side of the semantics will be ignored in this section and explained in §5.3.1.

The term and the four dynamic contexts together are the formalization of the computation's state traditionally called ***configuration***.

For uniformity, the special identifier '`this`' and the identifiers of parameters are treated within a method like the identifiers of local variables. Explicit read access is necessary to get at their values. As in Java, parameters can be updated and the update of '`this`' is only prevented by a special check in the typing rules (§5.4.1).

4. JaM's Formalization of Reference Values. Values—more precisely, *first-class values*—are those things to which expressions can evaluate *and* which can be stored in variables *and* passed as parameters and results (and which are classified

[3] om's graph can be understood as a *set of implementation objects* formalized as triples $\langle o, \varrho_o, F_o\rangle$.

$$
\begin{array}{lll}
\models_{\mathfrak{s}} om \Leftrightarrow_{\mathrm{df}} \forall o, x, \tilde{o}, \tilde{\mu}, \tilde{\omega}\bullet & om(o) \doteq \langle \varrho, F\rangle \wedge \mathfrak{s}(\varrho(x)) \doteq \langle \tilde{o}, \tilde{\mu}, \tilde{\omega}\rangle & \Rightarrow \tilde{o} = o \\
\models_{\mathfrak{s}} \vec{\eta} \quad \Leftrightarrow_{\mathrm{df}} \forall \eta^{\kappa}_{\langle \mathbf{s}, \mu, \mathbf{r}\rangle}, x, \tilde{o}, \tilde{\mu}, \tilde{\omega}\bullet & \eta^{\kappa}_{\langle \mathbf{s}, \mu, \mathbf{r}\rangle} \in \vec{\eta} \quad \wedge \mathfrak{s}(\eta(x)) \doteq \langle \tilde{o}, \tilde{\mu}, \tilde{\omega}\rangle & \Rightarrow \tilde{o} = \mathbf{r} \\
\models_{\mathfrak{s},\vec{\eta}} e \ \Leftrightarrow_{\mathrm{df}} \forall \eta^{\kappa}_{\langle \mathbf{s}, \mu, \mathbf{r}\rangle}, \vec{\eta}', \tilde{o}, \tilde{\mu}, \tilde{\omega}\bullet & \vec{\eta} = \eta^{\kappa}_{\langle \mathbf{s}, \mu, \mathbf{r}\rangle} \bullet \vec{\eta}' & \\
\quad \Rightarrow \ e = \langle \tilde{o}, \tilde{\mu}, \tilde{\omega}\rangle \vee (e \in \mathcal{L}oc \wedge \mathfrak{s}(e) \doteq \langle \tilde{o}, \tilde{\mu}, \tilde{\omega}\rangle) & \Rightarrow \tilde{o} = \mathbf{r} & \\
\quad \wedge \ \forall \tilde{e}\bullet \quad e \in \{\mathtt{val}(\tilde{e}), \mathtt{destval}(\tilde{e}), \mathtt{return}\ \tilde{e}\mathtt{;}\} & \Rightarrow \models_{\mathfrak{s},\vec{\eta}} \tilde{e} & \\
\quad \wedge \ \forall \tilde{e}, \hat{e}, s, \psi\bullet \quad e \in \{\tilde{e}\ \hat{e}, \mathtt{if}(\tilde{e}\psi\hat{e})\mathtt{\{}s\mathtt{\}}, \tilde{e}\mathtt{=}\hat{e}\mathtt{;}\} & \Rightarrow \models_{\mathfrak{s},\vec{\eta}} \tilde{e} \ \wedge \ \models_{\mathfrak{s},\vec{\eta}} \hat{e} & \\
\quad \wedge \ \forall f, e_0, \ldots, e_k\bullet \ e = e_0 \Leftarrow f\,(e_1, \ldots, e_k) & \Rightarrow \forall i \in \{0, \ldots, k\}\bullet \models_{\mathfrak{s},\vec{\eta}} e_i & \\
\quad \wedge \ \forall \tilde{e}\bullet \quad e = \ll \tilde{e} \gg & \Rightarrow \vec{\eta}' \neq \epsilon \wedge \models_{\mathfrak{s},\vec{\eta}'} \tilde{e} &
\end{array}
$$

Figure 5.4: Handle source consistency

by the types t in the program). Java has *primitive values* of boolean and numeric types, and *reference values*, i.e., references to dynamically created objects [GJS00]. Base-JaM restricts itself to just reference values.

Normally, a reference value is formalized as an ***object identifier***: Each time a new object is created, a fresh identifier ω is drawn for it from a given set $\mathbb{O}$, and used henceforth to refer to that object.[4] And the notion of a null reference (denoted by `null`) is formalized by the special value $\mathsf{nil} \notin \mathbb{O}$ that does not identify any object.

The base-JaM semantics uses an extended formalization of object references as so-called ***handles***: A handle is not just the object-identifier ω of the referred-to object (the reference's *target*), but a triple $h = \langle o, \mu, \omega\rangle$ which includes also the identifier o of the referring object (the reference's *source*) and the mode μ of o's reference to ω. This extension will simplify to specify which object graph edges $o \xrightarrow{\mu} \omega$ are added and removed during an execution step.

It is expected, and will be shown to be the case in paragraph *8*, that the sources in all handles in the store and the runtime term coincide with the object to which the corresponding store location or method nesting level belongs (***source consistency***). Put the other way around, at locations $\ell = \varrho_o(x)$ of fields x of object o, we expect to find only handles $\mathfrak{s}(\ell) = h$ whose source is o. Then the object-map is source consistent, in symbols, $\models_{\mathfrak{s}} om$. This is defined formally in figure 5.4. Analogously, at locations $\ell = \eta_i(x)$ of local variables and parameters x in environments η_i of invocations with receiver $\mathbf{r}$, we expect to find only handles $\mathfrak{s}(\ell) = h$ whose source is $\mathbf{r}$. Then the object-map is source consistent, $\models_{\mathfrak{s}} \vec{\eta}$. And at all method nesting levels in the runtime term e with corresponding receiver $\mathbf{r}$, we expect to find only handles h with source $\mathbf{r}$, and locations ℓ containing handles $\mathfrak{s}(\ell) = h$ with source $\mathbf{r}$. If this is the case then the runtime term is source consistent, in symbols, $\models_{\mathfrak{s},\vec{\eta}} e$. In figure 5.4, this is defined inductively from the outermost method nesting (corresponding to the

[4] An object value $\langle \varrho_\omega, F_\omega\rangle$ cannot be a formal model of reference values, since then the reference to all instances of all empty classes c, c' (no fields, no methods) would be the same: $\langle \emptyset, \emptyset\rangle$. But then `new` c`()` `==` `new` c'`()`, contrary to the semantics of Java.

location (l-value)	$\ell \in \mathcal{L}oc$	$=_{\text{df}} \biguplus_{\tau \in \mathcal{M} \times \mathbb{C}} \mathcal{L}oc_\tau$
handle (value)	$h \in \mathcal{V}$	$=_{\text{df}} (\mathbb{O} \cup \{\mathsf{nil}\}) \times \mathcal{M} \times (\mathbb{O} \cup \{\mathsf{nil}\})$
object-identifier	$o \in \mathbb{O}$	$=_{\text{df}} \biguplus_{c \in \mathbb{C}} \mathbb{O}_c$
object value	$\langle \varrho, F \rangle \in$	$(\mathit{Id} \nrightarrow \mathcal{L}oc) \times (\mathit{Id} \nrightarrow \mathit{Mth})$

infinite countable sets $\mathbb{O}_c$ given for all $c \in \mathbb{C}$ and $\mathcal{L}oc_\tau$ for all $\tau \in \mathcal{M} \times \mathbb{C}$

$$
\begin{array}{llll}
\tau \in \mathcal{T} ::= \mathsf{ref}\,\mathcal{M}\;\mathbb{C} & [\![\mathsf{ref}\,\mu\; c]\!] & =_{\text{df}} & \mathcal{L}oc_{\mu\; c} \\
\quad\mid\ \mathcal{M}\;\mathbb{C} & [\![\mu\; c]\!] & =_{\text{df}} & (\mathbb{O} \cup \{\mathsf{nil}\}) \times \{\mu\} \times (\mathbb{O}_c \cup \{\mathsf{nil}\}) \\
\quad\mid\ \mathsf{obj}\;\mathbb{C} & [\![\mathsf{obj}\; c]\!] & =_{\text{df}} & \{\langle \varrho, F\rangle \mid\ \vdash \mathit{FldsMths}(c) = \langle \Gamma, F\rangle \text{ and } \varrho \models \Gamma\} \\
\quad\mid\ \mathsf{Cmd} & [\![\mathsf{Cmd}]\!] & =_{\text{df}} & \{\epsilon\}
\end{array}
$$

$$
\begin{array}{lllr}
\eta \models \Gamma & \Leftrightarrow_{\text{df}} & \mathrm{dom}(\eta) = \mathrm{dom}(\Gamma) \ \wedge\ \forall x \in \mathrm{dom}(\Gamma)\textbf{.}\quad \eta(x) & \in [\![\Gamma(x)]\!] \\
\models \mathfrak{s} & \Leftrightarrow_{\text{df}} & \forall \tau \in \mathcal{M} \times \mathbb{C}, \ell \in \mathrm{dom}(\mathfrak{s})\textbf{.}\ \ell \in \mathcal{L}oc_\tau \Rightarrow \mathfrak{s}(\ell) & \in [\![\tau]\!] \\
\models \mathit{om} & \Leftrightarrow_{\text{df}} & \forall c \in \mathbb{C}, o \in \mathrm{dom}(\mathit{om})\textbf{.}\ o \in \mathbb{O}_c \Rightarrow \qquad \mathit{om}(o) & \in [\![\mathsf{obj}\; c]\!]
\end{array}
$$

Figure 5.5: Semantic values, types, and type-consistency

bottom of the environment stack), to deeper nesting levels (corresponding to higher levels in the environment stack). Note that `while` statements and the then-branch of `if` statements can be ignored since as program terms they can neither contain handles nor locations.

The complete set of reference values, and thus the set $\mathcal{V}$ of values in base-JaM, is $(\mathbb{O} \cup \{\mathsf{nil}\}) \times \mathcal{M} \times (\mathbb{O} \cup \{\mathsf{nil}\})$: Handles with $\mathsf{nil} \notin \mathbb{O}$ instead of the *target* identifier, "nil-*handles*," formalize the notion of a null reference. (This formalization of null references by multiple semantic values ensures that all handles have a uniform triple-structure.) Handles with nil instead of the *source* identifier can be understood as "*global references*" not belonging to any object. Since in base-JaM there are no static variables nor methods, a handle with source nil occurs only as the call-link for the environment η_0 in which the start-up expression e_0 is interpreted, and as the handle to which the object creation expression in e_0 evaluates.

5. MORE SEMANTIC VALUES AND TYPES. For uniformity in the formal treatment, the notion of value is sometimes generalized beyond first-class values to include also store locations (the value to which names x and `this.`x of variables "evaluate", *l-value*), the empty sequence ϵ (as the "value" to which non-returning statements reduce), and object values (to which object-identifiers are mapped by om). For the typing of these "second-class" values, and for typing runtime terms by the type of value they reduce to, the type terms t from the program syntax are generalized to type terms $\tau \in \mathcal{T}$ shown in figure 5.5. The *extensional interpretation* (or denotation) $[\![\tau]\!]$ of a type term τ is the set of all conceivable values of type τ.

It is straight-forward to define what it means for an environment η as a mathematical structure to be a model for, or *consistent* with, a set $\Gamma = \{x_1 : \tau_1, \ldots, x_n : \tau_n\}$ of type assumptions (a ***type assignment***), in standard logical symbols, $\varrho \models \Gamma$: Both must be defined for the same identifiers x_i, and the environment must assign them semantic values $\eta(x_i)$ from the set $[\![\tau_i]\!]$ denoted by the corresponding type assumption $x_i : \tau_i$ in Γ. Treating Γ as partial mapping, we can write

$$\eta \models \Gamma \Leftrightarrow_{\mathrm{df}} \mathrm{dom}(\eta) = \mathrm{dom}(\Gamma) \ \wedge \ \forall x \in \mathrm{dom}(\Gamma)\textbf{.} \ \eta(x) \in [\![\Gamma(x)]\!]$$

The set $[\![\mathsf{obj}\ c]\!]$ of possible ***object values*** for instances of class c consists of those tuples $\langle \varrho, F\rangle$ where the field environment ϱ is consistent with class c's instance record type Γ (assignment of types to field names), and where the method-suite F is precisely the one which c defines for its instances:

$$[\![\mathsf{obj}\ c]\!] =_{\mathrm{df}} \{\langle \varrho, F\rangle \mid \ \vdash \mathit{FldsMths}(c) = \langle \Gamma, F\rangle \text{ and } \varrho \models \Gamma\}$$

The set $\mathbb{O}$ of ***object-identifiers*** is assumed to be partitioned according to class names $c \in \mathbb{C}$ into disjoint subsets $\mathbb{O}_c$ reserved as identifiers for instances of class c. The object-map is *type-consistent*, written $\models om$, if it maps object-identifiers in class c's partition $\mathbb{O}_c$ only to object values of c-instances:

$$\models om \Leftrightarrow_{\mathrm{df}} \forall c \in \mathbb{C}, o \in \mathrm{dom}(om)\textbf{.} \ o \in \mathbb{O}_c \Rightarrow om(o) \in [\![\mathsf{obj}\ c]\!]$$

Handles $h \in \mathcal{V}$ are classified into *handle types* $\mu\ c \in \mathcal{M} \times \mathbb{C}$ according to their mode μ and their target's object class c: The extensional interpretation $[\![\mu\ c]\!]$ of handle type $\mu\ c$ is the set of handles with any object-identifier or nil as source, μ as mode, and any object-identifier in partition $\mathbb{O}_c$, or nil, as target.[5]

$$[\![\mu\ c]\!] =_{\mathrm{df}} (\mathbb{O} \cup \{\mathsf{nil}\}) \times \{\mu\} \times (\mathbb{O}_c \cup \{\mathsf{nil}\})$$

The set of store ***locations*** $\ell \in \mathcal{L}oc$ is assumed to be partitioned into disjoint subsets $\mathcal{L}oc_\tau$ according to the type τ of values which the location is supposed to hold. Since the only first-class values in base-JaM are handles, the partitioning is by handle types $\tau = \mu\ c \in \mathcal{M} \times \mathbb{C}$. Store $\mathfrak{s}$ is *type-consistent*, written $\models \mathfrak{s}$, if it maps locations in each type τ's partition $\mathcal{L}oc_\tau$ only to values in type (term) τ's extension $[\![\tau]\!]$:

$$\models \mathfrak{s} \Leftrightarrow_{\mathrm{df}} \forall \tau \in \mathcal{M} \times \mathbb{C}, \ell \in \mathrm{dom}(\mathfrak{s})\textbf{.} \ \ell \in \mathcal{L}oc_\tau \Rightarrow \mathfrak{s}(\ell) \in [\![\tau]\!]$$

Given this organization of the store, variables ranging over τ-values are represented in the store at locations $\ell \in \mathcal{L}oc_\tau$: Hence the interpretation $[\![\mathsf{ref}\ \tau]\!]$ of the type of τ-variables found in type assumptions $(x : \mathsf{ref}\ \tau) \in \Gamma$, is the set $\mathcal{L}oc_\tau$.

$$[\![\mathsf{ref}\ \mu\ c]\!] =_{\mathrm{df}} \mathcal{L}oc_{\mu\ c}$$

$\models \mathfrak{s}, om$ can be written as short-hand for $\models \mathfrak{s}$ and $\models om$.

[5]For the inclusion of subclass-polymorphism in JaM (see §7.2.2), it is necessary to clarify that $\mathsf{obj}\ c$ is the *monomorphic* type of the object values of the *direct* instances of class c (and not its subclasses), and that $\mathbb{O}_c$ is the set of identifiers only for *direct* c-instances. Subclass-polymorphism for handle types $\mu\ c$ would require on the semantic side to generalize $\mathbb{O}_c$ in the definition of $[\![\mu\ c]\!]$ to the subclass-closure $\bigcup_{c' \leq_{\mathrm{c}} c} \mathbb{O}_{c'}$. And true mode-polymorphism (inclusion polymorphism instead of mere ad-hoc polymorphism through mode-conversions) would require us to generalize $\{\mu\}$ in the definition of $[\![\mu\ c]\!]$ to the set $\bigcup_{\mu' \leq_{\mathrm{m}} \mu} \{\mu'\}$ of all modes μ' mode-compatible to μ (see §5.4.2).

$$\frac{\mathcal{E} \in R_1^{\Box} \quad e,\, \vec{\eta}, \mathfrak{s}, om, \mathfrak{g} \longrightarrow e',\, \vec{\eta}', \mathfrak{s}', om', \mathfrak{g}'}{\mathcal{E}[e],\, \vec{\eta}, \mathfrak{s}, om, \mathfrak{g} \Longrightarrow \mathcal{E}[e'],\, \vec{\eta}', \mathfrak{s}', om', \mathfrak{g}'}$$

$$\frac{\mathcal{E} \in R_1^{\Box} \quad e,\, \vec{\eta}, \mathfrak{s}, om, \mathfrak{g} \Longrightarrow e',\, \vec{\eta}', \mathfrak{s}', om', \mathfrak{g}'}{\mathcal{E}[\ll e \gg],\, \eta_h^{\kappa} \bullet \vec{\eta}, \mathfrak{s}, om, \mathfrak{g} \Longrightarrow \mathcal{E}[\ll e' \gg],\, \eta_h^{\kappa} \bullet \vec{\eta}', \mathfrak{s}', om', \mathfrak{g}'}$$

$$\begin{aligned} R_1^{\Box} ::= {} & \Box \\ & \mid \texttt{val}(R_1^{\Box}) \mid \texttt{destval}(R_1^{\Box}) \mid R_1^{\Box}\Leftarrow \mathit{Id}(\mathrm{E}^*) \mid \mathcal{V}\Leftarrow \mathit{Id}((\mathcal{V},)^*\, R_1^{\Box}\,(,\mathrm{E})^*) \\ & \mid R_1^{\Box}\mathrm{S} \mid R_1^{\Box}\texttt{=}\,\mathrm{E}\texttt{;} \mid \mathcal{L}oc\,\texttt{=}R_1^{\Box}\texttt{;} \mid \texttt{return}\ R_1^{\Box}\texttt{;} \mid \texttt{if(}R_1^{\Box}\Psi\, R\texttt{)\{}\mathrm{S}\texttt{\}} \mid \texttt{if(}\mathcal{V}\,\Psi\, R_1^{\Box}\texttt{)\{}\mathrm{S}\texttt{\}} \end{aligned}$$

Figure 5.6: Top-level reduction rules

5.2.2 Computational Steps

6. SELECTION OF THE REDEX. The definition of reduction steps $\hat{e}, \vec{\eta}, \mathfrak{s}, om, \mathfrak{g} \Longrightarrow \hat{e}', \vec{\eta}', \mathfrak{s}', om', \mathfrak{g}'$ can be split into two complementary aspects: On one side are twelve cases of subterms that can be completely substituted in one step to a new term, without any unchanged term-context around it. This substitution, in conjunction with corresponding changes in the dynamic contexts, will be captured in redex replacement rules $e, \vec{\eta}, \mathfrak{s}, om, \mathfrak{g} \longrightarrow e', \vec{\eta}', \mathfrak{s}', om', \mathfrak{g}'$. On the other side is the selection of the substitutable subterm in $\hat{e}$ to substitute in this step, the *redex*. This selection can be conveniently specified with the help of Wright and Felleisen's notion of a ***reduction context*** [WF94]: These are explicit, syntactic contexts for the substitution that can be defined with the standard grammar formalism. A reduction context $\mathcal{E}^\star$ is a runtime term "with a hole" symbolized by '$\Box$'. A complete runtime term $\hat{e} = \mathcal{E}^\star[e]$ is obtained by filling a runtime term e into the hole, i.e., by substituting e for '$\Box$'. Reduction steps then are written $\mathcal{E}^\star[e], \vec{\eta}, \mathfrak{s}, om, \mathfrak{g} \Longrightarrow \mathcal{E}^\star[e'], \vec{\eta}', \mathfrak{s}', om', \mathfrak{g}'$. The advantage is that, instead a contextual reduction rule for each syntactical alternative, with reduction contexts one rule can unite all cases in which the dynamic contexts $\vec{\eta}, \mathfrak{s}, om, \mathfrak{g}$ change the same way.

In base-JaM, one rule handles substitution of a redex e at the *same* method nesting level as $\hat{e} = \mathcal{E}^\star[e]$, while another rule handles substitution across one level of method nesting (see figure 5.6). For these rules we do not need the general, multi-level reduction contexts $\mathcal{E}^\star$, but single-level reduction contexts $\mathcal{E} \in R_1^{\Box}$ that do not increase the inserted term's method nesting level. (A general reduction context $\mathcal{E}^\star$ with the hole at method nesting level n corresponds to the nesting of n single-level reduction contexts $\mathcal{E}_i \in R_1^{\Box}$: $\mathcal{E}^\star = \mathcal{E}_1[\ll \mathcal{E}_2[\ll \ldots [\ll \mathcal{E}_{n-1}[\ll \mathcal{E}_n \gg] \gg] \ldots \gg] \gg]$.)

1. If e is a redex reducing $e, \vec{\eta}, \mathfrak{s}, om, \mathfrak{g} \longrightarrow e', \vec{\eta}', \mathfrak{s}', om', \mathfrak{g}'$, then the same reduction is possible in any single-level reduction context $\mathcal{E} \in R_1^{\Box}$, i.e., from $\mathcal{E}[e]$ to $\mathcal{E}[e']$.
2. If e is reducible $e, \vec{\eta}, \mathfrak{s}, om, \mathfrak{g} \Longrightarrow e', \vec{\eta}', \mathfrak{s}', om', \mathfrak{g}'$, then e's nesting in a method inlining $\hat{e} = \ll e \gg$ can be reduced to $\hat{e}' = \ll e' \gg$ if the environment stack is

extended *at the bottom* by some environment η_h^κ for the new outer-most nesting level. Additionally, $\hat{e}$ and $\hat{e}'$ can be wrapped in a single-level context $\mathcal{E} \in R_1^\square$.

The role of the reduction contexts $\mathcal{E} \in R_1^\square$ defined in figure 5.6 is to determine the place of substitution *within* a method nesting level:

- If $\hat{e}$ is a substitutable subterm or an inlined method body then the two rules replace $\hat{e}$ without any context, i.e., "context" $\mathcal{E}$ is nothing but the hole $\square$, so that $\hat{e} = \mathcal{E}[\hat{e}]$.
- If otherwise $\hat{e}$ has the form $\texttt{val}(\tilde{e})$, $\texttt{destval}(\tilde{e})$ or $\texttt{return}\ \tilde{e}$ then the next reduction step must change the only proper subterm e'' in the unchanged context of $\mathcal{E} = \texttt{val}(e^\square)$, $\texttt{destval}(e^\square)$, or $\texttt{return}\ e^\square$ (where the hole is, or is contained in, $e^\square$).
- Before replacing operation call expression $\hat{e} = e_0 \Leftarrow f(e_1, \ldots, e_n)$ itself, its subterms e_0 to e_n must be evaluated left to right: The subterm e_i in which the substitution is to take place has only subterms already reduced to values to its left and only proper expressions to its right: Context $\mathcal{E} = e^\square \Leftarrow f(e_1, \ldots, e_n)$ where the subterms e_i are expressions in E directs the substitution first to the receiver expression. Then context $\mathcal{E} = v_0 \Leftarrow f(v_1, \ldots, v_{i-1}, e^\square, e_{i+1}, \ldots, e_n)$ with values $v_i \in \mathcal{V}$ and expressions $e_i \in E$ directs it to the left-most unevaluated argument expression.
- If $\hat{e}$ is a sequence $s_1\ s_2$ of two statements, substitution takes place only in the first statement. s_2 is context until s_1 has reduced to ϵ and only s_2 is left: $\mathcal{E} = e^\square\ s_2$.
- Before the reduction step that can execute assignment $\hat{e} = e_1 \texttt{=} e_2\texttt{;}$ itself, context $\mathcal{E} = e^\square \texttt{=} e_2\texttt{;}$ directs the substitution to the left-hand side until e_1 is an irreducible location ℓ. Then context $\mathcal{E} = \ell \texttt{=} e^\square\texttt{;}$ makes the substitution continue in the right-hand side until e_2 has evaluated to an irreducible value.
- Before an `if` statement can be executed, the expressions it compares must be evaluated. To this end, contexts $\mathcal{E} = \texttt{if}(e^\square \Psi e)\texttt{\{}s\texttt{\}}$ and $\mathcal{E}' = \texttt{if}(h \Psi e^\square)\texttt{\{}s\texttt{\}}$ direct substitution first to the left hand expression and then to the right hand expression.

7. Substitution of the Redex. Now consider the possible replacements of a subterm, and how environment, store, and object-map change with it (figures 5.7 and 5.8). The object graph component will be ignored in this section; the changes there will be discussed in §5.3. All steps work on the top-level environment η_h^κ only, except for return steps that works with the finished top-level environment $\eta^{\star}{}^{\kappa^\star}_{\langle \mathbf{s}, \mu_\mathbf{r}, \mathbf{r}\rangle}$ and the environment η_h^κ to which the execution will return.

{var$_l$} The identifier $x \in Id$ of a local variable or parameter reduces to that location which the environment defines for x, i.e., the location $\eta(x)$ to which the actual environment η in annotated η_h^κ maps x. Nothing else changes.

{var$_f$} A field name $\texttt{this.}x$ reduces to that location ℓ' which is specified for x in the field environment ϱ of the current object, i.e., of the target ω of the handle $\mathfrak{s}(\ell)$ at the location $\ell = \eta(\texttt{this})$ denoted by `this` in the top-level environment η_h^κ.

{rd$_{cp}$} Non-destructive read access $\texttt{val}(\ell)$ to a location ℓ copies the value from the store (at location ℓ) to the runtime term (at the redex position). In base-JaM, this value is always a handle $\langle o, \mu, \omega\rangle$. In case of a `free` handle, an exact copy

$$\{\mathrm{var}_l\}\ \frac{\eta(x) \doteq \ell}{x,\ \eta_h^\kappa, \mathfrak{s}, om, \mathfrak{g} \longrightarrow \ell,\ \eta_h^\kappa, \mathfrak{s}, om, \mathfrak{g}}$$

$$\{\mathrm{var}_f\}\ \frac{\eta(\texttt{this}) \doteq \ell \quad \mathfrak{s}(\ell) \doteq \langle o, \mu, o\rangle \quad om(o) \doteq \langle \varrho, F\rangle \quad \varrho(x) \doteq \ell'}{\texttt{this}.x,\ \eta_h^\kappa, \mathfrak{s}, om, \mathfrak{g} \longrightarrow \ell',\ \eta_h^\kappa, \mathfrak{s}, om, \mathfrak{g}}$$

$$\{\mathrm{rd}_{cp}\}\ \frac{\mathfrak{s}(\ell) \doteq \langle o, \mu, \omega\rangle \quad \mu' = \mu[\texttt{read}/\texttt{free}]}{\texttt{val}(\ell),\ \eta_h^\kappa, \mathfrak{s}, om, \mathfrak{g} \longrightarrow \langle o, \mu', \omega\rangle,\ \eta_h^\kappa, \mathfrak{s}, om, \mathfrak{g} \oplus o \xrightarrow{\mu'} \omega}$$

$$\{\mathrm{rd}_{dst}\}\ \frac{\mathfrak{s}(\ell) \doteq \langle o, \mu, \omega\rangle}{\texttt{destval}(\ell),\ \eta_h^\kappa, \mathfrak{s}, om, \mathfrak{g} \longrightarrow \langle o, \mu, \omega\rangle,\ \eta_h^\kappa, \mathfrak{s}[\ell \mapsto \langle o, \mu, \mathsf{nil}\rangle], om, \mathfrak{g}}$$

$$\{\mathrm{null}\}\ \frac{h \doteq \langle \mathbf{s}, \mu_{\mathbf{r}}, \mathbf{r}\rangle}{\texttt{null},\ \eta_h^\kappa, \mathfrak{s}, om, \mathfrak{g} \longrightarrow \langle \mathbf{r}, \texttt{free}, \mathsf{nil}\rangle,\ \eta_h^\kappa, \mathfrak{s}, om, \mathfrak{g}}$$

$$\{\mathrm{new}\}\ \frac{\begin{array}{l} h \doteq \langle \mathbf{s}, \mu_{\mathbf{r}}, \mathbf{r}\rangle \quad \vdash \mathit{FldsMths}(c) \doteq \langle \{\overline{x_i : \mathsf{ref}\ \mu_i\ c_i}\}, F\rangle \\ \text{fresh } \mathbf{o} \in \mathbb{O}_c \quad \overline{\text{fresh } \ell_i \in [\![\mathsf{ref}\ \mu_i\ c_i]\!]} \quad \varrho = \{\overline{x_i \mapsto \ell_i}\} \quad \overline{h_i = \langle \mathbf{o}, \mu_i, \mathsf{nil}\rangle} \end{array}}{\texttt{new}\ c\texttt{()},\ \eta_h^\kappa, \mathfrak{s}, om, \mathfrak{g} \longrightarrow \langle \mathbf{r}, \texttt{free}, \mathbf{o}\rangle,\ \eta_h^\kappa, \mathfrak{s}[\ell_i \mapsto h_i], om[\mathbf{o} \mapsto \langle \varrho, F\rangle], \mathfrak{g} \oplus \mathbf{r} \xrightarrow{\texttt{free}} \mathbf{o}}$$

$$\{\mathrm{call}\}\ \frac{\begin{array}{l} \mathbf{r} \in \mathbb{O}_c, \quad om(\mathbf{r}) \doteq \langle \ldots, F\rangle, \quad F(f) \doteq \kappa^\star\, \tau\, f\,(\overline{\mu_i\ c_i\ y_i})\texttt{\{}\overline{\mu'_j\ c'_j\ z_j}\texttt{;}\ s\texttt{\}} \\ \text{fresh } \ell \in [\![\mathsf{ref}\ \mathsf{co}\ c]\!],\ \overline{\text{fresh } \ell_i^y \in [\![\mathsf{ref}\ \mu_i\ c_i]\!]},\ \overline{\text{fresh } \ell_j^z \in [\![\mathsf{ref}\ \mu'_j\ c'_j]\!]} \\ \eta^\star = \{\texttt{this} \mapsto \ell, \overline{y_i \mapsto \ell_i^y}, \overline{z_j \mapsto \ell_j^z}\} \\ \mathfrak{s}' = \mathfrak{s}[\ell \mapsto \langle \mathbf{r}, \mathsf{co}, \mathbf{r}\rangle, \overline{\ell_i^y \mapsto \langle \mathbf{r}, \mu_i, \mathbf{o}_i\rangle}, \overline{\ell_j^z \mapsto \langle \mathbf{r}, \mu'_j, \mathsf{nil}\rangle}] \\ \mathfrak{g}' = \mathfrak{g} \ominus \overline{\mathbf{s} \xrightarrow{\mu''_i} \mathbf{o}_i} \oplus \mathbf{r} \xrightarrow{\mathsf{co}} \mathbf{r} \oplus \overline{\mathbf{r} \xrightarrow{\mu_i} \mathbf{o}_i} \end{array}}{\langle \mathbf{s}, \mu_{\mathbf{r}}, \mathbf{r}\rangle \Leftarrow f\,(\overline{\langle \mathbf{s}, \mu''_i, \mathbf{o}_i\rangle}),\ \eta_h^\kappa, \mathfrak{s}, om, \mathfrak{g} \longrightarrow\ \ll s \gg,\ \eta_h^\kappa \bullet \eta^{\star\kappa^\star}_{\langle \mathbf{s}, \mu_{\mathbf{r}}, \mathbf{r}\rangle}, \mathfrak{s}', om, \mathfrak{g}'}$$

Figure 5.7: Reduction of expression redices

would immediately violate the uniqueness of free paths' heads (the Unique Head property). Prohibiting through the type system that `free` variables are read non-destructively would be too restrictive, as explained in §5.4.2, since then no (observer) call to a `free` object can be made without losing the `free` handle to it. The copy is safe *if* its mode is weakened to `read`, since the aliasing by `read` references is irrelevant for the integrity invariants (cf. paragraph *5*). We use the standard notation $\mu[\texttt{read}/\texttt{free}]$ for the substitution of `read` for `free` in mode μ. In base-JaM, substitution merely means to replace $\mu = \texttt{free}$ to `read` and leave other μ's unchanged. In full JaM, it will mean to replace any occurence of base-mode `free` in the full mode $\mu = \mu\texttt{<}\delta\texttt{>}$ to `read`, and besides this leave μ unchanged. The object graph transformation will be discussed in §5.3.

{rd_{dst}} Destructive read access $\texttt{destval}(\ell)$ evaluates to the value at location ℓ, but resets the store at ℓ to a nil-handle (with the same source and mode as before).

{null} The expression `null` evaluates to a nil-handle whose source is the current object, i.e., the target $\mathbf{r}$ of the top-level environment's call-link, and whose mode is free. Note that this mode is compatible to all other modes (see §5.4.2).

{new} The evaluation of an object creation expression instantiates a class c to a new object with fresh object-identifier $\mathbf{o}$, and evaluates to an initial, free handle from the current object $\mathbf{r}$ (the creator) to the new object $\mathbf{o}$. Being fresh implies in particular that $\mathbf{o}$ is neither source nor target of any edge in the object graph. Let $\{\overline{x_i : \mathsf{ref}\, \mu_i\ c_i}\}$ be the instance record type Γ and F the method suite which class c defines for its instances. Then instantiating c means to take fresh locations ℓ_i of respective types $\mathsf{ref}\, \mu_i\ c_i$, initialize them to nil-handles with source $\mathbf{o}$ and modes μ_i, and map $\mathbf{o}$ to an object value $\langle\{\overline{x_i \mapsto \ell_i}\}, F\rangle$ with the field names mapped to these locations, and with the method suite F.

{call} An operation call is executed when all its subexpressions, receiver and arguments, have evaluated (to handles). The execution will then continue with the body s of the method $F(f)$ by which the call's receiver $\mathbf{r}$ implements the called operation f. To prepare this continuation and the return into the context of the call, the call expression is replaced by the body s put into double angle brackets. The environment for s's subsequent evaluation contains this, and the parameters and local variables of method $F(f)$ bound to fresh locations of corresponding ref-types initialized with, respectively, a handle to the receiver (of mode co), argument expression values adapted to the parameters' modes, and nil-handles of the local variables' modes. In order to talk and reason about the kinds of executing methods and the modes of the call-links used to make the call and to return the result, the new environment is annotated with the kind of method $F(f)$, and with the handle to which the receiver expression evaluated.

{ret} A return statement is executed when its return expression has evaluated to a result handle, provided it is the remains of an inlined method body in double angle brackets, and there is an environment η_h^κ below the current top-level environment. Then evaluation will continue in environment η_h^κ with the result handle adapted to the calling context, i.e., with the sender as the new source and with a mode adapted to the sender's perspective. How modes of returned handles are adapted will be elaborated in §5.4.2. The current top-level environment is removed from the stack and the locations of the names in it (parameters, locals, and this) are removed from the store.

{upd} An assignment statement is executed when the left-hand side has reduced to a location ℓ and the right-hand side to a value $\langle o, \mu', \omega'\rangle$. It updates the store at ℓ to the handle with the mode adapted according to the location's store partition. As opposed to Java, assignments in base-JaM have no value, but are statements reducing to the empty term, so that the statement following it in the full term $\mathcal{E}[e]$ will be next in the order of execution.

{if$_t$}, {if$_f$} A conditional statement is executed when the compared expressions have evaluated to handles $\langle o, \mu, \omega\rangle$ and $\langle o, \mu', \omega'\rangle$, respectively. Then the if statement reduces either to the guarded statement s, the "then-branch," or it reduces to the empty term ϵ to let execution continue with the statement following the if

$$\{\text{ret}\}\ \frac{\mathfrak{s}' = \mathfrak{s}[\ell \mapsto \bot \mid \ell \in \mathrm{im}(\eta^\star)] \quad \mathfrak{g}' = \mathfrak{g} \oplus \mathbf{s} \xrightarrow{\mu_\mathbf{r} \circ \mu} \mathbf{o} \ominus \mathbf{s} \xrightarrow{\mu_\mathbf{r}} \mathbf{r} \ominus \mathbf{r} \xrightarrow{\mu} \mathbf{o} \ominus \mathfrak{s}(\mathrm{im}(\eta^\star))}{\ll\!\texttt{return}\ \langle \mathbf{r}, \mu, \mathbf{o} \rangle\,\texttt{;}\!\gg,\ \eta_h^\kappa \bullet \eta^{\star\kappa^\star}_{\langle \mathbf{s}, \mu_\mathbf{r}, \mathbf{r}\rangle}, \mathfrak{s}, om, \mathfrak{g} \longrightarrow \langle \mathbf{s}, \mu_\mathbf{r} \circ \mu, \mathbf{o} \rangle, \eta_h^\kappa, \mathfrak{s}', om, \mathfrak{g}'}$$

$$\{\text{upd}\}\ \frac{\ell \in \mathcal{L}oc_{\mu\ c}}{\ell \texttt{=} \langle o, \tilde{\mu}, \tilde{\omega} \rangle\texttt{;},\ \eta_h^\kappa, \mathfrak{s}, om, \mathfrak{g} \longrightarrow\ \epsilon,\ \eta_h^\kappa, \mathfrak{s}[\ell \mapsto \langle o, \mu, \tilde{\omega} \rangle], om, \mathfrak{g} \ominus o \xrightarrow{\tilde{\mu}} \tilde{\omega} \ominus \mathfrak{s}(\ell) \oplus o \xrightarrow{\mu} \tilde{\omega}}$$

$$\{\text{if}_t\}\ \frac{[\![\psi]\!](\omega, \omega')}{\texttt{if}\,\texttt{(}\langle o, \mu, \omega \rangle\, \psi\, \langle o, \mu', \omega' \rangle\texttt{)}\ \texttt{\{}s\texttt{\}},\ \eta_h^\kappa, \mathfrak{s}, om, \mathfrak{g} \longrightarrow\ s,\ \eta_h^\kappa, \mathfrak{s}, om, \mathfrak{g} \ominus o \xrightarrow{\mu} \omega \ominus o \xrightarrow{\mu'} \omega'}$$

$$\{\text{if}_f\}\ \frac{\neg[\![\psi]\!](\omega, \omega')}{\texttt{if}\,\texttt{(}\langle o, \mu, \omega \rangle\, \psi\, \langle o, \mu', \omega' \rangle\texttt{)}\ \texttt{\{}s\texttt{\}},\ \eta_h^\kappa, \mathfrak{s}, om, \mathfrak{g} \longrightarrow\ \epsilon,\ \eta_h^\kappa, \mathfrak{s}, om, \mathfrak{g} \ominus o \xrightarrow{\mu} \omega \ominus o \xrightarrow{\mu'} \omega'}$$

$$\{\text{wh}\}\ \frac{}{\texttt{while(}e_1 \psi\, e_2\texttt{)\{}s\texttt{\}},\ \eta_h^\kappa, \mathfrak{s}, om, \mathfrak{g} \longrightarrow \texttt{if(}e_1 \psi\, e_2\texttt{)\{}s\ \texttt{while(}e_1 \psi\, e_2\texttt{)\{}s\texttt{\}\}},\ \eta_h^\kappa, \mathfrak{s}, om, \mathfrak{g}}$$

where $[\![\texttt{==}]\!](\omega, \omega') \Leftrightarrow_{\text{df}} \omega = \omega'$ and $[\![\texttt{!=}]\!](\omega, \omega') \Leftrightarrow_{\text{df}} \omega \neq \omega'$

Figure 5.8: Reduction of statement redices

statement in the full term $\mathcal{E}[e]$. The choice depends on whether the two handles' targets ω and ω' are equal and whether the comparison operator ψ is `==` or `!=`. The first choice (rule $\{\text{if}_t\}$) is taken iff ψ is '`==`' and ω is ω', or if ψ is '`!=`' and ω is not ω'. Otherwise, the second choice is taken (rule $\{\text{if}_f\}$).

{wh} A `while` loop is reduced the standard way by unfolding it to an `if` statement guarding the first repetition of the loop's body followed by a copy of the loop.

8. Source Consistency. Before turning to higher-level views, let us verify that JaM's semantics adds the right source objects to its "handle" formalization of object reference values:

Proposition 1 If $e_0,\ \eta_0, \mathfrak{s}_0, om_0, \mathfrak{g}_0 \Longrightarrow^* e,\ \vec{\eta}, \mathfrak{s}, om, \mathfrak{g}$ then

$$\models_\mathfrak{s} om \ \wedge\ \models_\mathfrak{s} \vec{\eta} \ \wedge\ \models_{\mathfrak{s},\vec{\eta}} e$$

Proof by induction on the number N of reduction steps from e_0 to e: In the base case $N = 0$, source consistency is trivial since store $\mathfrak{s}_0 = \emptyset$ contains no handles, and term $e_0 = \texttt{new}\ c\texttt{().main()}$ contains no handles and no locations. In the induction step $N \to N+1$, reduction $e_0,\ \eta_0, \mathfrak{s}_0, om_0, \mathfrak{g}_0 \Longrightarrow^* e_N,\ \vec{\eta}_N, \mathfrak{s}_N, om_N, \mathfrak{g}_N$ is continued $e_N,\ \vec{\eta}_N, \mathfrak{s}_N, om_N, \mathfrak{g}_N \Longrightarrow e,\ \vec{\eta}, \mathfrak{s}, om, \mathfrak{g}$. By induction hypothesis, $\models_{\mathfrak{s}_N} om_N$ and $\models_{\mathfrak{s}_N} \vec{\eta}_N$ and $\models_{\mathfrak{s}_N, \vec{\eta}_N} e_N$.

Consider $\models_\mathfrak{s} om$ and $\models_\mathfrak{s} \vec{\eta}$. Reductions with $\{\text{var}_l\}$, $\{\text{var}_f\}$, $\{\text{rd}_{cp}\}$, $\{\text{null}\}$, $\{\text{if}_t\}/\{\text{if}_f\}$ and $\{\text{wh}\}$ change neither $\mathfrak{s}$ nor om nor $\vec{\eta}$; $\{\text{ret}\}$ neither adds to $\mathfrak{s}$ nor $\vec{\eta}$; and $\{\text{upd}\}$ and $\{\text{rd}_{dst}\}$ do not change the source of the handle at the updated location. Hence still $\models_\mathfrak{s} om$ and $\models_\mathfrak{s} \vec{\eta}$ in all these cases. In case of $\{\text{new}\}$ and $\{\text{call}\}$, om does not change for old objects, and $\vec{\eta}$ does not change in old environments. $\mathfrak{s}$ changes only at locations that are *fresh*. These locations are added only to, respectively, the new object's value or the new environment. All the handles with which these fresh locations are initialized have the right object as their source: the new object $\mathbf{o}$, or the receiver $\mathbf{r}$, respectively. Hence $\models_{\mathfrak{s}} om$ and $\models_{\mathfrak{s}} \vec{\eta}$ again.

Consider $\models_{\mathfrak{s},\vec{\eta}} e$. In case of a reduction with $\{\mathrm{var}_l\}$, the redex x in the maximal method nesting depth is replaced to location $\eta(x)$ from the top-level environment η^{κ}_{h}. Hence $\models_{\mathfrak{s}_N} \vec{\eta}_N$ ensures that $\mathfrak{s}(\ell)$'s source is the target $\mathbf{r}$ of call-link h in top-level environment η^{κ}_{h}, and thus the right one for a location at maximal nesting depth: $\models_{\mathfrak{s},\vec{\eta}} e$. And in case of $\{\mathrm{var}_f\}$, we have a handle $h = \mathfrak{s}(\ell) = \langle o, \mu, o\rangle$ at the location $\ell = \eta(\mathtt{this})$ of `this` in top-level environment η^{κ}_{h}. On one hand, $\models_{\mathfrak{s}_N} \vec{\eta}_N$ ensures that h's source o is the target $\mathbf{r}$ of call-link h in η^{κ}_{h}. On the other hand, $\models_{\mathfrak{s}_N} om_N$ ensures that the handle $\mathfrak{s}(\ell')$ at location ℓ' of o's field x has o as source. Hence the location ℓ' inlined in the runtime term at maximal nesting depth refers to a handle $\mathfrak{s}(\ell')$ in the store with the necessary top-level environment's receiver $o = \mathbf{r}$ as source.

In case of $\{\mathrm{rd}_{cp}\}$ and $\{\mathrm{rd}_{dst}\}$, the redices $\mathtt{val}(\ell)$ and $\mathtt{destval}(\ell)$ at nesting level n imply by induction hypothesis that $\mathfrak{s}(\ell)$ is a handle with the right source for nesting level n. Hence this handle can be copied into the runtime term at nesting level n with no problem. Although $\{\mathrm{rd}_{dst}\}$ does update the store, it does not change the source of the handle at location ℓ. $\models_{\mathfrak{s},\vec{\eta}} e$ is preserved.

In case of $\{\mathrm{null}\}$ and $\{\mathrm{new}\}$, a handle is added to the term that has as source specifically the receiver of the top-level environment's call-link, and thus the right one for a handle at maximal nesting depth. At the same time the environments remain unchanged and the store changes at most at fresh locations (but not at locations that might be contained in the term). Hence $\models_{\mathfrak{s},\vec{\eta}} e$.

Reductions with $\{\mathrm{call}\}$, $\{\mathrm{upd}\}$, $\{\mathrm{if}_t\}/\{\mathrm{if}_f\}$ and $\{\mathrm{wh}\}$ add neither handles nor locations to the term, At the same time the environments at old nesting levels remain unchanged and the store changes at most either at fresh locations ($\{\mathrm{call}\}$) or without changing the source of the handle at the updated location. Hence still $\models_{\mathfrak{s},\vec{\eta}} e$. ■

5.3 JaM's Higher-Level View

The runtime model consisting of terms, environments, stores and object-maps is a formal model of the computation's state well suited for defining program execution. It is less convenient for reasoning about relationships between objects and groupings of objects. More appropriate is the *object graph* model as a higher-level view of computational state that captures (only) the objects' interconnection by object references.

5.3.1 The Object Graph in the Computation

1. Object Graph View of State. The notion of an object graph in a computational state formalized as configuration $(e, \vec{\eta}, \mathfrak{s}, om)$ is a graph $\mathfrak{g}$ whose nodes are the (identifiers of) objects in om, and which has an edge $o \xrightarrow{\mu} \omega$ for every non-nil-handle $\langle o, \mu, \omega\rangle$ contained as value in $\mathfrak{s}$ (stored reference), call-link in $\vec{\eta}$ (reference in use as connector), or subterm in e (intermediate reference). The current object graph can always be calculated from the current configuration with the help of an abstraction function, and is then transformed indirectly in the reduction steps by the configuration's modification. But in order to make these transformations more obvious, the reduction rules given in figures 5.7 and 5.7 showed explicitly the manipulation of the current object graph as a separate component of the configuration. It is of course necessary to demonstrate consistency of this parallel object graph with the calculated object graph, which will be done further below.

In the reduction rules it is easy to *add* edge $o \xrightarrow{\mu} \omega$ to the graph whenever a handle $\langle o, \mu, \omega\rangle$ appears new in $\mathfrak{s}$, $\vec{\eta}$, or e. Harder is the *removal* of edge $o \xrightarrow{\mu} \omega$ exactly when handle $\langle o, \mu, \omega\rangle$ exists *nowhere* in e, $\vec{\eta}$, and $\mathfrak{s}$ any more. This can elegantly be handled if the object graph is not formalized as a set $\mathfrak{g} \in 2^{\mathbb{O}\times\mathcal{M}\times\mathbb{O}}$ of edges representing the existing of corresponding handles, but as a *multiset* $\mathfrak{g} \in \mathbf{N}^{\mathbb{O}\times\mathcal{M}\times\mathbb{O}}$ of edges whose multiplicity represents the *number* of the corresponding handles' occurrences in $\mathfrak{s}$, $\vec{\eta}$, or e: Multiplicities of edges are increased and decreased in accordance with the addition and removal of handles to/from e, $\vec{\eta}$ and $\mathfrak{s}$, so that the multiplicity of edge $o \xrightarrow{\mu} \omega$ in $\mathfrak{g}$, written $mult(o \xrightarrow{\mu} \omega, \mathfrak{g})$, reaches zero (meaning it disappears from the graph) exactly when the last occurrence of $\langle o, \mu, \omega\rangle$ is removed from $\mathfrak{s}$, $\vec{\eta}$ and e.

Definition 1 An ***object graph*** is a *multiset* $\mathfrak{g} \in \mathfrak{Graph} =_{\text{df}} \mathbf{N}^{\mathbb{O}\times\mathcal{M}\times\mathbb{O}}$ of directed, mode-labeled edges $o \xrightarrow{\mu} \omega \in \mathbb{O}\times\mathcal{M}\times\mathbb{O}$ between two object-identifiers $o, \omega \in \mathbb{O}$ called *source* and *target*, respectively.

W.r.t. this definition, we can now give precise meaning to the often cited notion of "the" object graph in a particular computational state: It is the abstract view of a configuration $(e, \vec{\eta}, \mathfrak{s}, om)$ as an object graph which contains every edge as often as e, $\vec{\eta}$, and $\mathfrak{s}$ contain the corresponding handle. It can be constructed from the current configuration by an abstract function ogr:

Definition 2 Let $n_h = num(h, e) + num(h, \vec{\eta}) + num(h, \mathfrak{s})$ be the combined number of occurrences of a handle $h \in \mathbb{O}\times\mathcal{M}\times\mathbb{O}$ as intermediate value, as call-link, and as stored value. Then the object graph $ogr(e, \vec{\eta}, \mathfrak{s}) \in \mathfrak{Graph}$ in configuration $(e, \vec{\eta}, \mathfrak{s}, om)$ is calculated by adding n_h-times every possible handle $h \in \mathbb{O}\times\mathcal{M}\times\mathbb{O}$ as an edge:

$$ogr(e, \vec{\eta}, \mathfrak{s}) =_{\text{df}} \biguplus_{h\in\mathbb{O}\times\mathcal{M}\times\mathbb{O}} \biguplus_{i=1}^{n_h} \{h\}$$

where $\uplus$ is the multiset union that adds up elements' multiplicities.

The number $num(h, \mathfrak{s})$ of locations at which h occurs in $\mathfrak{s} \in \mathfrak{Store}$ is

$$num(h, \mathfrak{s}) =_{\text{df}} \Big|\, \{\ell \in \text{dom}(\mathfrak{s}) \,|\, \mathfrak{s}(\ell) = h\} \,\Big|$$

The number $num(h, \vec{\eta})$ of environments with call-link h in stack $\vec{\eta}$ (of size n) is

$$num(h, \vec{\eta}) =_{\mathrm{df}} \left| \{ i \in \{1, \ldots, n\} \mid \eta_i = \eta_h^{\kappa} \} \right|$$

The number $num(h, e)$ of occurrences of h in runtime term $e \in R$ can be determined inductively as follows:

$$\begin{array}{lll@{\qquad}ll}
num(h, x) & =_{\mathrm{df}} 0 & & num(h, \mathtt{val}(e)) & =_{\mathrm{df}} num(h, e) \\
num(h, \mathtt{this}.x) & =_{\mathrm{df}} 0 & & num(h, \mathtt{destval}(e)) & =_{\mathrm{df}} num(h, e) \\
num(h, \ell) & =_{\mathrm{df}} 0 & & num(h, \ll s \gg) & =_{\mathrm{df}} num(h, s) \\
num(h, h') & =_{\mathrm{df}} & \begin{cases} 1 \text{ if } h' = h \\ 0 \text{ if } h' \neq h \end{cases} & \begin{array}{l} num(h, \mathtt{return}\ e\,\mathtt{;}) \\ num(h, s_1\ s_2) \end{array} & \begin{array}{l} =_{\mathrm{df}} num(h, e) \\ =_{\mathrm{df}} num(h, s_1) \end{array} \\
num(h, \mathtt{null}) & =_{\mathrm{df}} 0 & & num(h, e_1\mathtt{=}e_2\mathtt{;}) & =_{\mathrm{df}} num(h, e_1) + num(h, e_2) \\
num(h, \mathtt{new}\ c\mathtt{()}) & =_{\mathrm{df}} 0 & & num(h, \mathtt{if}\,\mathtt{(}e_1 \psi e_2\mathtt{)}\{s\}) & =_{\mathrm{df}} num(h, e_1) + num(h, e_2) \\
num(h, \mathtt{while}\,\mathtt{(}e\mathtt{)}\{s\}) & =_{\mathrm{df}} 0 & & num(h, e_0 \Leftarrow f\,\mathtt{(}e_1, \ldots, e_n\mathtt{)}) & =_{\mathrm{df}} \sum_{i=0}^{n} num(h, e_i)
\end{array}$$

The definition of $num(h, e)$ can ignore the body and condition of `while` statements, the then-branch of `if` statements, and the second statement in sequences since these are never partially evaluated, and thus always free of handles (cf. the syntax of runtime terms in §5.2.1).

2. Object Graph View of Steps. Transformations of the object graph can be decomposed into what looks like additions $\mathfrak{g} \oplus h$ and removals $\mathfrak{g} \ominus h$ of edges, but which are actually increases and decreases of edges' multiplicities. Such an "addition" and "removal" does not change the graph at all if the target or source in handle h is nil since object graphs model only the connections between *objects*. The "addition" $\oplus$ and "removal" $\ominus$ used in the semantics are reduced as follows to multiset-union '$\uplus$' and multiset-subtraction '$\setminus\!\!\setminus$' (which add two multisets' element multiplicities, or subtract the second one's element multiplicities from those of the first one).

$$\mathfrak{g} \oplus o \xrightarrow{\mu} \omega =_{\mathrm{df}} \begin{cases} \mathfrak{g} & \text{if } \mathsf{nil} \in \{o, \omega\} \\ \mathfrak{g} \uplus \{o \xrightarrow{\mu} \omega\} & \text{otherwise} \end{cases} \qquad \mathfrak{g} \ominus o \xrightarrow{\mu} \omega =_{\mathrm{df}} \begin{cases} \mathfrak{g} & \text{if } \mathsf{nil} \in \{o, \omega\} \\ \mathfrak{g} \setminus\!\!\setminus \{o \xrightarrow{\mu} \omega\} & \text{otherwise} \end{cases}$$

Now, let us follow the transformations which the object graph undergoes by the different reduction steps defined in figures 5.7 and 5.7:

$\{\mathrm{var}_l\}$, $\{\mathrm{var}_f\}$, $\{\mathrm{null}\}$, and $\{\mathrm{wh}\}$ steps have no effect on the object graph since they do not change the number of non-nil handles in the configuration. Observe that `while` statements are pure program terms so that the subterms duplicated by the reduction to an `if` statement cannot contain any handles.

$\{\mathrm{rd}_{dst}\}$ leaves the object graph unchanged: The new occurence of handle $h = \langle o, \mu, \omega \rangle$ in the term is balanced by removing one occurence from the store: $num(h, e') + num(h, \mathfrak{s}') = num(h, e) + num(h, \mathfrak{s})$.

$\{\mathrm{rd}_{cp}\}$ increases in the graph the multiplicity of the handle $h = \langle o, \mu', \omega \rangle = o \xrightarrow{\mu'} \omega$ read from the store with substitution of `read` for `free` (unless o or ω is nil): $mult(h, \mathfrak{g}') = mult(h, \mathfrak{g}) + 1$. This models the redex's substitution to h, which increases the number of h's occurences in the term: $num(h, e') = num(h, e) + 1$.

$\{\mathrm{new}\}$ adds creator object $\mathbf{r}$'s initial reference to the new object ω to the object graph (except if $\mathbf{r}$ is nil, as in the first step of evaluating $e_0 \equiv$ `new` c_n`().main()`):

$\mathfrak{g}' = \mathfrak{g} \oplus \mathbf{r} \xrightarrow{\texttt{free}} \omega$. This models the redex's substitution to $\langle \mathbf{r}, \texttt{free}, \omega \rangle$.

{call} steps equip the receiver with a `this` reference $\mathbf{r} \xrightarrow{\texttt{co}} \mathbf{r}$ and with a parameter handle $\mathbf{r} \xrightarrow{\mu_i} \omega_i$ for every argument handle $\mathbf{s} \xrightarrow{\mu_i''} \omega_i$ supplied by the sender. That is, the multiplicity of $\mathbf{r} \xrightarrow{\texttt{co}} \mathbf{r}$ and edges $\mathbf{r} \xrightarrow{\mu_i} \omega_i$ increases, while that of edges $\mathbf{s} \xrightarrow{\mu_i''} \omega_i$ decreases. This matches the arguments' disappearance from the term and the parameters' and the `this`-reference's appearance at fresh locations in the store. The call-link $o \xrightarrow{\mu} \omega$ is not changed: Its disappearance from the term is balanced by its occurence in the new top-level environment.

{ret} steps combine call-link $\langle \mathbf{s}, \mu_\mathbf{r}, \mathbf{r} \rangle$ and the edge $\mathbf{r} \xrightarrow{\mu} \omega$ returned by the receiver to the edge $\mathbf{s} \xrightarrow{\mu_\mathbf{r} \circ \mu} \omega$ in the sender, i.e., the former two edge's multiplicity decreases while the latter one's multiplicity increases. This matches the appearence of $\langle \mathbf{s}, \mu_\mathbf{r} \circ \mu, \omega \rangle$ in the runtime term and the disappearence of handle $\mathbf{r} \xrightarrow{\mu} \omega$ from the term and of call-link $\langle \mathbf{s}, \mu_\mathbf{r}, \mathbf{r} \rangle$ (together with the finished invocation) from the environment stack. Additionally, since the locations of the finished invocation's variables in the store are reset, the multiplicities of all (non-nil) handles lost by this are decreased to keep the object graph in sync.

{upd} steps convert a handle $o \xrightarrow{\mu'} \omega'$ to $o \xrightarrow{\mu} \omega'$, i.e., decrease the multiplicity of the first handle and increase that of the second one. This matches, respectively, the disappearence of the right-hand side handle $\langle o, \mu', \omega' \rangle$ from the term and the appearance the handle $\langle o, \mu, \omega' \rangle = o \xrightarrow{\mu} \omega'$ at location ℓ in the store. Additionally, the multiplicity of the old handle $\langle o, \mu, \omega \rangle = o \xrightarrow{\mu} \omega$ at location ℓ decreases since the update at location ℓ overwrites it.

{if_t} and {if_f} steps' discarding of the two compared handles means for the object graph a decrease of the corresponding edges' multiplicity.

All of this shows that the reduction rules accurately make explicit, as parallel transformations of the object graph, how the objects' interconnections change in each reduction step through the modification of term, environments, and store:

Proposition 2 If $e_0, \eta_0, \mathfrak{s}_0, om_0, \mathfrak{g}_0 \Longrightarrow^* e, \vec{\eta}, \mathfrak{s}, om, \mathfrak{g}$ then

$$\mathfrak{g} = ogr(e, \vec{\eta}, \mathfrak{s})$$

Proof by induction on the number N of reduction steps from e_0 to e': In the base case $N = 0$, $ogr(e_0, \eta_0, \mathfrak{s}_0) = \emptyset = \mathfrak{g}_0$ since term $e_0 = \texttt{new } c\texttt{().main()}$ and store $\mathfrak{s}_0 = \emptyset$ contain no handles, and environment stack $\eta_0 = \emptyset^{\mathsf{obs}}_{\langle \mathsf{nil}, \mathsf{read}, \mathsf{nil} \rangle}$ contains only a nil-handle. In the induction step, execution $e_0, \eta_0, \mathfrak{s}_0, om_0, \mathfrak{g}_0 \Longrightarrow^* e_N, \vec{\eta}_N, \mathfrak{s}_N, om_N, \mathfrak{g}_N$ is continued $e_N, \vec{\eta}_N, \mathfrak{s}_N, om_N, \mathfrak{g}_N \Longrightarrow e, \vec{\eta}, \mathfrak{s}, om, \mathfrak{g}$. As the above considerations showed, the last step's redex replacement changed the multiplicities in the graph the same way as the non-nil-handle occurences in the substituted subterm, the top-most environment(s), and the store. The context rules add the same term contexts and lower-level environments on both sides. Hence $\mathfrak{g} = ogr(e, \vec{\eta}, \mathfrak{s})$ follows from the induction hypothesis's $\mathfrak{g}_N = ogr(e_N, \vec{\eta}_N, \mathfrak{s}_N)$. ■

$$\frac{o \xrightarrow{\mu} \omega \in \mathfrak{g}}{\mathfrak{g} \vdash o \xrightarrow{\mu} \omega \in PAP(o,\mu,\omega)} \qquad \frac{\mathfrak{g} \vdash \pi_1 \in PAP(o,\mu,q) \quad \mathfrak{g} \vdash \pi_2 \in PAP(q,\texttt{co},\omega)}{\mathfrak{g} \vdash \pi_1 \bullet \pi_2 \in PAP(o,\mu,\omega)}$$

Figure 5.9: Potential access paths in object graphs labeled with base-modes

5.3.2 Moded Paths, Owners and Sanctuaries

3. Paths in the Graph are non-empty sequences $\pi = h_1, \ldots, h_n \in \mathcal{V}^+$ of contiguous edges, i.e., of object references $h_i = o_i \xrightarrow{\mu_i} \omega_i$ with $o_{i+1} = \omega_i$, thus also written $\pi = o_1 \xrightarrow{\mu_1} o_2 \ldots o_n \xrightarrow{\mu_n} o_{n+1}$. The mode-based classification of object references $o \to \omega$ according to their ownership- and sanctuary-meaning (cf. §5.1) generalizes to *paths* from o to ω (subsuming the case of object references as paths of length one):

- If the path's mode is `rep`, this means that o is ω's owner, which is expected to be unique, and ω belongs to o's sanctuary $Sanc(o)$. In legal base-JaM programs, ω has no other owner (the Unique Owner property).
- If the path's mode is `free`, this means that o is ω's owner, which is expected to be unique, and ω is expected not to belong to any sanctuary. In legal base-JaM programs, ω has no other owner, and all ownership paths to ω, i.e., `free` and `rep` paths, have the same first reference of multiplicity one (the Unique Head property).
- If the path's mode is `co`, this means that ω and o have the same owner (or none), and they belong to the same sanctuaries $Sanc(q)$. o and ω are called *co-objects.*
- If the path's mode is `read`, this means that it say nothing about ω's owner and membership in sanctuaries.

Paths π of mode μ between o and ω can be written $o \overset{\mu}{\dashrightarrow} \omega$ in abstraction from the intermediate objects and the intermediate references' modes. While they have the same meaning for ownership and sanctuaries like object references $o \xrightarrow{\mu} \omega$, paths can of course be used neither as data values nor as connectors to call operations on the target object ω. But a path $\pi = o \xrightarrow{\mu_1} o_2 \ldots o_n \xrightarrow{\mu_n} \omega$ can in principle anytime be "collapsed" to a single reference $o \xrightarrow{\mu} \omega$ by a sequence of calls from o to ω along the path which returns ω's `this`-handle $\langle \omega, \texttt{co}, \omega \rangle$ to o. If a path is classified as a μ-path then this collapsed reference must be of mode μ. That is, the combined adaption $\mu_1 \circ (\mu_2 \circ \ldots (\mu_{n-1} \circ (\mu_n \circ \texttt{co})) \ldots)$ of the returned handle's mode should be μ. That is, while a *μ-reference* is a *means* for the source o to directly access target ω (with some limitations imposed by the type system according to μ), a *μ-path* indicates o's (μ-bounded) *right*, in principal, to directly access ω. Hence these paths will be called ***potential access paths***.

Let $PAP(o,\mu,\omega)$ be the set of all those paths π from o to ω in graph $\mathfrak{g}$ which are potential access paths of mode μ. Which paths in $\mathfrak{g}$ are potential access paths, and what their mode is, is controlled by the modes of the edges in $\mathfrak{g}$. Through mode annotations specifying the references' modes, the program therefore indirectly also specifies the potential access paths. In base-JaM, the potential access paths $\pi \in PAP(o, \mu, \omega)$ are the graph's edges $o \xrightarrow{\mu} \omega \in \mathfrak{g}$ (that is, edges with $mult(o \xrightarrow{\mu} \omega, \mathfrak{g}) > 0$), and the concatenation $\pi_1 \bullet \pi_2$ of a μ-path $\pi_1 \in PAP(o, \mu, q)$ and a co-path $\pi_2 \in PAP(q, \texttt{co}, \omega)$ (see figure 5.9). This is easy to verify: A rep-path π_1 and co-path π_2 together imply that o is not only q's but also ω's owner and that not only q but also ω is in o's sanctuary. This is exactly what the classification of $\pi_1 \bullet \pi_2$ as `rep` says about ω. If π_1 is `free` then the co-path π_2 between q and ω implies that o is not only q's but also ω's owner and that not only q but also ω is in no object's sanctuary. Hence $\pi_1 \bullet \pi_2$ should be classified as `free`. Co paths π_1 and π_2 together mean that o and ω have the same owners and belong to the same sanctuaries, i.e., $\pi_1 \bullet \pi_2$ is `co`. Read path π_1 leaves q's owners and sanctuary memberships unspecified, and co-path π_2 equates them with ω's owners and sanctuary-memberships. They are thus left unspecified by $\pi_1 \bullet \pi_2$, so that `read` is the right mode.

4. OWNERSHIP AND SANCTUARIES. The potential access paths of modes `rep` and `free` define the ownership (or object composition) hierarchy between objects in the object graph; they are the ***ownership paths***. For convenience, we can define the set $Osh(o, \omega)$ of ownership paths between o and ω. $Osh(o, \omega) =_{\text{df}} PAP(o, \texttt{rep}, \omega) \cup PAP(o, \texttt{free}, \omega)$. And the transitive closure of potential access paths of mode `rep` defines the ***sanctuary*** $Sanc(o)$ of composite objects' representatives o.

However, potential access paths are only forward concatenations of handles, so that in the situation $o \xrightarrow{\texttt{rep}} \omega \xleftarrow{\texttt{co}} \omega'$, object o would own ω but not ω'. But when ω' calls an operation on ω, like `SetPrev`, to which it passes `this` as parameter of mode `co`, then the co-handle is inverted and o now also owns ω' through path $o \xrightarrow{\texttt{rep}} \omega \xrightarrow{\texttt{co}} \omega'$. In order to show for this step the preservation of the invariants over ownership and representations introduced below, the forward notion of ownership and representations is generalized to a co-symmetric one, in which both ends of co-handles and co-paths have the same owner and belong to the same representations.

This is achieved by the following technical trick: The sets $PAP_{\mathfrak{g}}(o, \mu, q)$ of potential access paths, in particular, ownership paths $Osh_{\mathfrak{g}}(o, \omega)$, used in the properties' definition are not the ones determined in the real object graph $\mathfrak{g}$, but those in an object graph $\mathfrak{g}^\star$ to which inverses $\omega \xrightarrow{\texttt{co}} \omega'$ for each co-handle $\omega' \xrightarrow{\texttt{co}} \omega$ have been added. This addition explicitly represents the semantic symmetry of co-handles.

Definition 3 Let $\mathfrak{g}^\star =_{\text{df}} \mathfrak{g} \uplus \{\omega \xrightarrow{\texttt{co}} o \mid o \xrightarrow{\texttt{co}} \omega \in \mathfrak{g}\}$. Then

$$
\begin{aligned}
PAP_{\mathfrak{g}}(o, \mu, q) &=_{\text{df}} \{\pi \mid \mathfrak{g}^\star \vdash \pi \in PAP(o, \mu, q)\} \\
Osh_{\mathfrak{g}}(o, \omega) &=_{\text{df}} PAP_{\mathfrak{g}}(o, \texttt{rep}, \omega) \cup PAP_{\mathfrak{g}}(o, \texttt{free}, \omega) \\
Sanc_{\mathfrak{g}}(o) &=_{\text{df}} \bigcup_{\omega \text{ su. th. } PAP_{\mathfrak{g}}(o, \texttt{rep}, \omega) \neq \emptyset} \Big(\{\omega\} \cup Sanc_{\mathfrak{g}}(\omega)\Big)
\end{aligned}
$$

Object graph index $\mathfrak{g}$ in $PAP_{\mathfrak{g}}$, $Osh_{\mathfrak{g}}$ and $Sanc_{\mathfrak{g}}$ can be dropped where $\mathfrak{g}$ is obvious.

The construction of $PAP_{\mathfrak{g}}$ based on $\mathfrak{g}^\star$ means that any two objects o and ω that are in $\mathfrak{g}$ connected by an undirected path $o \xrightarrow{\texttt{co}}{}^* o_1{}^* \xleftarrow{\texttt{co}} \dots \xrightarrow{\texttt{co}}{}^* o_n{}^* \xleftarrow{\texttt{co}} \omega$ of co-edges will be connected by a potential access paths of mode `co` ($PAP_{\mathfrak{g}}(o, \texttt{co}, \omega) \neq \emptyset$). Since

$$
\begin{array}{lll}
\mathfrak{g} \models \mathsf{UO} & \Leftrightarrow_{\mathrm{df}} \forall o, \tilde{o}, \omega \bullet & Osh_{\mathfrak{g}}(o,\omega) \neq \emptyset \;\wedge\; Osh_{\mathfrak{g}}(\tilde{o},\omega) \neq \emptyset \;\Rightarrow\; \tilde{o} = o \\
\mathfrak{g} \models \mathsf{UH} & \Leftrightarrow_{\mathrm{df}} \forall o, \tilde{o}, \omega, h, \pi, \tilde{h}, \tilde{\pi} \bullet & h \bullet \pi \in PAP_{\mathfrak{g}}(o, \mathtt{free}, \omega) \wedge \tilde{h} \bullet \tilde{\pi} \in Osh_{\mathfrak{g}}(\tilde{o}, \omega) \\
 & & \Rightarrow\; \tilde{h} = h \wedge mult(h, \mathfrak{g}) = 1 \\
\mathfrak{g}, \vec{\eta} \models \mathsf{MCP} & \Leftrightarrow_{\mathrm{df}} \forall i \in \{1, \ldots, n\} \bullet & \kappa_i = \mathtt{mut} \Rightarrow \exists j \leq i \bullet\; h_j \bullet \ldots \bullet h_i \in Osh_{\mathfrak{g}}(\mathbf{r}_{j-1}, \mathbf{r}_i) \\
\mathfrak{g}, \vec{\eta} \models \mathsf{MC} & \Leftrightarrow_{\mathrm{df}} \forall i \in \{1, \ldots, n\}, o \bullet & \kappa_i = \mathtt{mut} \wedge \mathbf{r}_i \in Sanc_{\mathfrak{g}}(o) \\
 & & \Rightarrow\; \exists k \leq i \bullet\; \mathbf{r}_k = o \wedge \kappa_k = \mathtt{mut} \\
\multicolumn{3}{l}{\text{where } \vec{\eta} = \eta_1{}^{\kappa_1}_{h_1} \bullet \ldots \bullet \eta_n{}^{\kappa_n}_{h_n} \text{ with call-links } h_i = \langle \mathbf{r}_{i-1}, \mu_i, \mathbf{r}_i \rangle}
\end{array}
$$

Figure 5.10: Base-JaM integrity invariants

`free` and `rep` paths are closed under `co`-paths, o and ω have the same owners and belong to the same sanctuaries. Hence the existence of a potential access path of mode `co` between two objects formalizes the informal notion of ***co-objects***.

5. INTEGRITY INVARIANTS OF BASE-JAM SYSTEMS. Base-JaM defines not only through mode annotations where owners and representations are in the object graph. Through the mode system introduced in §5.4.2 it will also guarantee the integrity properties introduced informally in chapter 1. Based on the formal semantics, these properties can now be formalized (and then be proved in §5.5). Composite state encapsulation will be formalized in the next subsection. The invariant properties of base-JaM executions are formalized in figure 5.10 w.r.t. the object graph $\mathfrak{g}$ and the call-links and method kinds of invocations on the environment stack $\vec{\eta}$.

- The ***Unique Owner*** property UO is the property characteristic of object ownership in JaM: It holds in graph $\mathfrak{g}$ if all objects have at most one owner, i.e., are at most target of a unique object's ownership paths.
- The ***Unique Head*** property UH is the property characteristic of `free` paths in JaM: It holds in graph $\mathfrak{g}$ if the initial edge in all ownership paths to a `free` object (i.e., target of a `free` path) is the same and has multiplicity one. Since this excludes `rep` paths, the `free` object cannot belong to any sanctuary.
- The ***Mutator Control Path*** property MCP is the property characteristic of ownership paths in JaM: It holds in graph $\mathfrak{g}$ and stack $\vec{\eta}$ if *mutators* were invoked on receiver objects $\mathbf{r}_i$ only through a sequence of calls along the edges $h_j, \ldots, h_i$ of an an ownership path to $\mathbf{r}_i$.
- The ***Mutator Control*** property MC is the property characteristic of sanctuaries in JaM: It holds in graph $\mathfrak{g}$ and stack $\vec{\eta}$ if members of o's sanctuary are executing *mutators* only nested to mutator executions of o, and thus (indirectly) initiated by o's mutators through a sequence of calls.

$\mathfrak{g}, \vec{\eta} \models \mathsf{UO}, \mathsf{UH}, \mathsf{MCP}, \mathsf{MC}$ is short for "$\mathfrak{g} \models \mathsf{UO}$ and $\mathfrak{g} \models \mathsf{UH}$ and $\mathfrak{g}, \vec{\eta} \models \mathsf{MCP}$ and $\mathfrak{g}, \vec{\eta} \models \mathsf{MC}$."

The move from $\mathfrak{g}$ to $\mathfrak{g}^{\star}$ strengthens the notion of uniqueness of owners and `free` path's initial edges. For mutator control paths, this is irrelevant because the call-links

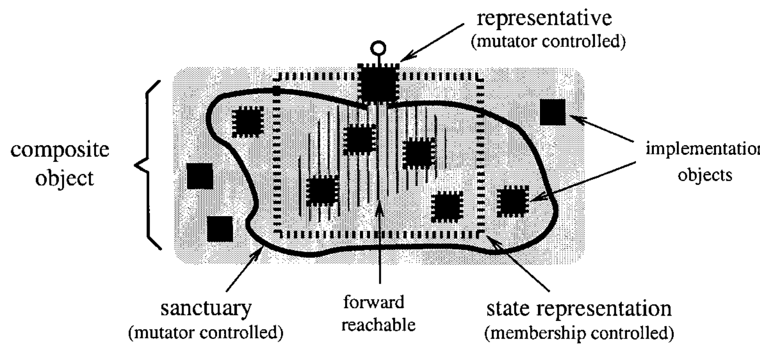


Figure 5.11: Composition of composite objects in JaM

in $\vec{\eta}$ are always real edges in $\mathfrak{g}$ (Proposition 2). But by inversion, all the additionally owned objects and representation members, i.e., objects not reachable by any real ownership path in $\mathfrak{g}$, are guaranteed not to execute mutators. They are immutable until a forward ownership path is established.

5.3.3 The Composite Object View

6. Composition of Composite Objects. A composite object O in JaM is constituted by its representative o and all the implementation objects reachable from there via sequences of `rep` and `free` paths. That is, the set of O's constituent objects is $composite(o) =_{\mathrm{df}} \{o\} \cup \bigcup_{Osh(o,\omega)\neq\emptyset} composite(\omega)$. Interaction between the constitutent objects is internal to O and abstracted away in the outside view of O. Interaction with any other object is external behavior of O (and should be included in its behavioral specification).

The *sanctuary* $Sanc(o)$ (see above) is the subset of the expansion $composite(o)$ which is reachable via `rep` path sequences only (cf. figure 5.11). o (indirectly) controls the *execution of mutators* in the sanctuary (mutator control), but it does not necessarily control the *membership* in the sanctuary: Through temporary `rep` or `co` references in the execution of observers (of o or members of $Sanc(o)$), new `rep` paths can be established that add an object to $Sanc(o)$. Even though this addition is only temporary, it is a change of the sanctuary not necessarily controlled by o.

The desired state encapsulation property does not require us to impose control on *temporary* additions since temporary members of the sanctuary can anyway not be used to represent the composite's state: To represent state, only a core of sanctuary members can effectively be used which remain in the sanctuary between method invocations and can be accessed via ownership paths from different method invocations of the representative. That is, the composite state representation $StRep(o)$ can only consist, besides representative o itself, of objects that are held in sanctuary $Sanc(o)$ through `rep` paths consisting entirely of references captured in fields.

However, the object graph, as it was defined above, is too abstract for the correct formalization of state representation $StRep(o)$ since it ignores the handles' storage status: stored *vs.* unstored, stored in fields *vs.* stored in locals. We have to look at the subgraph $fgr_{om}(\mathfrak{s}) \subseteq ogr(e, \vec{\eta}, \mathfrak{s})$ containing just the edges for handles $h = \mathfrak{s}(\ell)$ found in the store $\mathfrak{s}$ and at locations $\ell \in flds_{om}(o)$ that model an object's fields.

Definition 4 The set $flds_{om}(o)$ of object o's ***field locations*** is extracted from o's field environment ϱ in object-map om. The ***field-subgraph*** $fgr_{om}(\mathfrak{s})$ is the set (or multiset) of all non-nil handles at such field locations in the store. The ***state representation*** $StRep_{\mathfrak{s},om}(o)$ of o is o together with the state representations of its `rep` path targets *in the field-subgraph*.

$$\begin{aligned}
flds_{om}(o) &=_{\mathrm{df}} \mathrm{im}(\varrho) \quad \text{for } om(o) = \langle \varrho, F \rangle \\
fgr_{om}(\mathfrak{s}) &=_{\mathrm{df}} \bigcup_{o \in \mathrm{dom}(om)} \mathrm{im}(\mathfrak{s}\,|_{flds_{om}(o)}) \cap \mathbb{O} \times \mathcal{M} \times \mathbb{O} \\
StRep_{\mathfrak{s},om}(o) &=_{\mathrm{df}} \{o\} \cup \bigcup_{\omega \text{ su. th. } PAP_{fgr_{om}(\mathfrak{s})}(o,\mathtt{rep},\omega) \neq \emptyset} StRep_{\mathfrak{s},om}(\omega)
\end{aligned}$$

Proposition 3 The field "sub"-graph is in fact a subgraph of the object graph (as a set):

$$fgr_{om}(\mathfrak{s}) \;\subseteq\; ogr(e, \vec{\eta}, \mathfrak{s})$$

Proof: $h \in fgr_{om}(\mathfrak{s}) \;\Rightarrow\; h \in \mathrm{im}(\mathfrak{s}) \;\Rightarrow\; num(h, \mathfrak{s}) > 0 \;\Rightarrow\; h \in ogr(e, \vec{\eta}, \mathfrak{s})$. ■

Mutator Control (MC) means in particular that the representative controls all mutator executions in state representation $StRep(o)$, since the latter is a subset of the sanctuary $Sanc(o)$ modulo the representative (which trivially mutator controls itself):

Proposition 4 $\quad StRep_{\mathfrak{s},om}(o) \;\subseteq\; \{o\} \cup Sanc_{ogr(e,\vec{\eta},\mathfrak{s})}(o)$

Proof: Membership $\omega \in StRep_{\mathfrak{s},om}(o)$ presupposes a (possibly empty) sequence of `rep` paths from o to ω *in* $fgr_{om}(\mathfrak{s})$. This sequence exists also in $ogr(e, \vec{\eta}, \mathfrak{s}) \supseteq fgr_{om}(\mathfrak{s})$ (Proposition 3). If it is empty then $\omega = o$, otherwise $\omega \in Sanc_{ogr(e,\vec{\eta},\mathfrak{s})}(o)$. ■

7. Composite State. The notion of state representation $StRep(o)$ used here should not be mistaken as a kind of (concrete) *state*. It is the set of (identifiers for) the implementation objects which collectively represent the composite object's state $CState(o)$ by virtue of their shallow states $state(\omega)$. That is, $CState(o) = \bigcup_{\omega \in StRep(o)} state(\omega)$. Since objects' shallow states are in turn represented in their fields at store locations $\ell \in flds(o)$, the composite state is ultimately represented *in the store* at all the locations $\ell \in flds(\omega)$ for all $\omega \in StRep(o)$. (This set of locations is the *instance region* of [GB99] and the *demesne* of [Wil92].)

Definition 5 ***Shallow*** and ***composite state*** are then the restrictions of the system state, formalized as store $\mathfrak{s}$, to the corresponding location sets:

$$\begin{aligned}
state_{\mathfrak{s},om}(o) &=_{\mathrm{df}} \mathfrak{s}\,|_{flds_{om}(o)} \\
CState_{\mathfrak{s},om}(o) &=_{\mathrm{df}} \mathfrak{s}\,|_{\bigcup_{\omega \in StRep_{\mathfrak{s},om}(o)} flds_{om}(\omega)} = \bigcup_{\omega \in StRep_{\mathfrak{s},om}(o)} state_{\mathfrak{s},om}(\omega)
\end{aligned}$$

8. THE HIERARCHICAL VIEW. The above description of a composite object O as a flat set of constituent objects differs from the description of the composite object-oriented view of the runtime system as a nesting hierarchy of composite objects and their possibly composite component objects in §2.5. However, we can see all implementation objects o', also those in $composite(o)$, as representatives of a (possibly primitive) composite object O'. The *components* of composite O are those composite objects Ω_i to whose representatives ω_i the representative o of O has an ownership path. And the *state-representing components* are those components to whose representatives O's representative o has a ***rep*** path in the *field*-subgraph. Correspondingly, one could also give inductive definitions of $composite(o)$, $StRep(o)$, and $CState(o)$ based on representative o the (state-representing) components.

9. COMPOSITE STATE ENCAPSULATION. The notion of composite state encapsulation, which was introduced in chapter 1, can now be given a precise definition w.r.t. the JaM formalization: If an execution step $e, \vec{\eta}, \mathfrak{s}, om, \mathfrak{g} \Longrightarrow e', \vec{\eta}', \mathfrak{s}', om', \mathfrak{g}'$ changes a composite's state, i.e., $CState_{\mathfrak{s},om}(o) \neq CState_{\mathfrak{s}',om'}(o)$, then it is executing a mutator, i.e., there is an environment $\eta^{\mathtt{mut}}_{\langle \mathsf{s},\mu,o\rangle} \in \vec{\eta}$ of kind `mut` with receiver o:

$$\forall o \in \mathrm{dom}(om)\bullet\quad CState_{\mathfrak{s},om}(o) \neq CState_{\mathfrak{s}',om'}(o) \;\Rightarrow\; \exists \mathsf{s}, \mu, \eta\bullet\; \eta^{\mathtt{mut}}_{\langle \mathsf{s},\mu,o\rangle} \in \vec{\eta}$$

This property will be proved for legal base-JaM programs in §5.5.3.

5.4 Typed Base-JaM

Not all syntactically correct programs p are also *legal* programs. Type declarations are written in the program not just for fun but to have the actual use of values checked against a declared intention. This should ensure the orderly execution of programs, including in case of JaM the state encapsulation of composite objects. The component of a programming language which defines the checking of the program is called the *type system*.

5.4.1 The Type System

1. THE WELL-FORMEDNESS of Base-JaM programs is judged by the rules in figure 5.12:

[prog] A program $p \in P$ is a ***legal program*** (of typed base-JaM) whose execution starts with the evaluation of $e_0 \equiv \mathtt{new}\ c_n\mathtt{().main()}$, written $\vdash p$ **start** e_0, if it is *well-formed*: Each of the class modules in it is well-formed; no two class modules define the same class name; and the last module D_n defines the class c_n with a parameter-less operation `main`. For formal reason, this operation must be an *observer*: In the initial environment $\emptyset^{\mathsf{obs}}_{\langle \mathsf{nil},\mathsf{read},\mathsf{nil}\rangle}$, the only handle to the initially created c_n-instance o will be $\langle \mathsf{nil}, \mathtt{free}, o\rangle$. Since its source is $\mathsf{nil} \notin \mathbb{O}$, in

$$\text{[prog]}\ \frac{\vdash D_1\ \textbf{defs}\ c_1\ \cdots\ \vdash D_n\ \textbf{defs}\ c_n\quad \forall i,j=1,\ldots,n\bullet\ c_i=c_j\ \Rightarrow\ i=j \qquad \vdash \mathit{FldsMths}(c_n)=\langle R,F\rangle,\quad F(\mathtt{main})\doteq \mathtt{obs}\ \tau\ \mathtt{main()\{\ldots\}}}{\vdash D_1\ \ldots\ D_n\ \textbf{start}\ \mathtt{new}\ c_n\mathtt{().main()}}$$

$$\text{[class]}\ \frac{\vdash M_1\ \textbf{defs}\ x_1\ \cdots\ \vdash M_n\ \textbf{defs}\ x_n\quad \forall i,j=1,\ldots,n\bullet\ x_i=x_j\ \Rightarrow\ i=j}{\vdash \mathtt{class}\ c\,\mathtt{\{}M_1\ \ldots\ M_n\mathtt{\}}\ \textbf{defs}\ c}$$

$$\text{[meth]}\ \frac{\vdash t\ \textbf{ok}\quad \overline{\vdash t_i\ \textbf{ok}}\quad \overline{\vdash t'_j\ \textbf{ok}} \qquad \Gamma=\mathtt{this}:\mathsf{ref\ co}\ c,\ \overline{x_i:\mathsf{ref}\ t_i},\ \overline{z_j:\mathsf{ref}\ t'_j}\quad \vdash\Gamma\ \textbf{ok}\quad \Gamma,\kappa\vdash s:t}{\vdash \kappa\ t\ f\,\mathtt{(}\overline{t_i\ x_i}\mathtt{)\{}\overline{t'_j\ z_j;}\ s\mathtt{\}}\ \textbf{defs}\ f}$$

$$\text{[field]}\ \frac{\vdash t\ \textbf{ok}}{\vdash t\ x\mathtt{;}\ \textbf{defs}\ x} \qquad \text{[tassg]}\ \frac{\forall i,j=1,\ldots,n\bullet\ x_i=x_j\Rightarrow i=j}{\vdash x_1:\tau_1,\ldots,x_n:\tau_n\ \textbf{ok}}$$

$$\text{[rtype]}\ \frac{\mu\in\mathcal{M}\quad \vdash c\ \textbf{ok}}{\vdash \mu\ c\ \textbf{ok}} \qquad \text{[cname]}\ \frac{p\equiv D_1\ldots\ \mathtt{class}\,c\,\mathtt{\{\ldots\}}\ \ldots D_n}{\vdash c\ \textbf{ok}}$$

Figure 5.12: Legal base-JaM programs

the object graph no ownership path to o exists, so that a mutator call to o would violate Mutator Control Path. o's `main`, however, can then send mutators to `free` objects it created.[6]

[class] A class module D is a *well-formed* definition of class name c, written $\vdash D$ **defs** c, if each of the member definitions in it is well-formed and if no two member definitions define a member of the same name.

[meth] A method definition $M \equiv \kappa\, t\, f\,\mathtt{(}\overline{t_i\, x_i}\mathtt{)\{}\overline{t'_j\, z_j;}\ s\mathtt{\}}$ is a *well-formed* definition of member x, written $\vdash M$ **defs** x, under the following conditions: Its declared result and parameter types are valid types. The type assignment Γ made of the type assumptions for `this`, the parameter names and the local variable names is valid. And the method's body s is a well-formed term in the context of a κ-kind method and type assignment Γ whose (return) types is the method's result type. (The identifiers' assumed types all have the form $\mathsf{ref}\, t$, not the declared range type t, since the identifiers do not denote t-values but variables over them.)

[field] A field definition $M \equiv t\ x$ is a *well-formed* definition of member x, written $\vdash M$ **defs** x, if its declared range type t is a valid type.

[tassg] A list of type assumptions $x_i : \tau_i$ is a *valid* type assignment Γ, written $\vdash \Gamma$ **ok**, if it contains only one type assumption for each identifier x_i.

[rtype] Type term t is a *valid* range type for variables, parameters and results, written $\vdash t$ **ok**, if it is a valid class name c qualified by a mode $\mu \in \mathcal{M}$.

[6] Mutator `main` could be supported by reformulating the Mutator Control Path property or by assuming a given object $o_0 \in \mathbb{O}$ as the receiver in the initial environment: $\vec{\eta}_0 = \emptyset^{\mathtt{mut}}_{\langle \mathsf{nil},\mathtt{read},o_0\rangle}$

[cname] Identifier $c \in \mathbb{C}$ is a *valid* class name, written $\vdash c$ **ok**, if one of the program's class modules defines it.

2. TYPING RULES FOR PROGRAM TERMS (expressions and statements) have two functions: First, they infer the terms' types (*static types*) as a prediction of the types of the values to which these terms will evaluate in any possible computation (*dynamic types*). On top of that, conditions are incorporated in the typing rules which make the existence of a term typing a judgment on the term's well-formedness.

The typing judgment $\Gamma, \kappa \vdash e : \tau$ expresses that term e is legal in a method of kind κ and has static type τ in the context of type assumptions Γ for local variables. (The assumptions are met if they are type consistent with top-level environment η_h^κ, i.e., $\eta \models \eta_h^\kappa\Gamma$.) The rules for deriving typings in base-JaM are given in figure 5.13. The discussion of the aspects that belong to the mode system, namely the mode and method-kind checks, mode compatibility $\tau' \leq_\mathrm{m} \tau$, the signature $\Sigma(\mu\ c)$ of μ c-handles, and the set $\mathbf{Wr}(\kappa)$ of handle modes with write permission, will be defered to §5.4.2.

[var$_l$] Since identifiers x evaluate to $\eta(x)$, they are assigned the type $\Gamma(x)$ assumed for them in the given type assignment Γ (with $\eta \models \Gamma$).

[var$_f$] Field expressions `this`.x evaluate to the location of field x of the object referenced by `this`. Hence they are assigned the type τ which instance record type Γ_c specifies for x, where c is the target class in the range type `co` c of `this`'s assumed type $\Gamma($`this`$)$.

[rd$_{cp}$], [rd$_{dst}$] Expressions of read access to a variable named ν are normally assigned the type τ of the value range in the variables' type ref τ determined for ν. In case of mode-preserving non-destructive read, this is always legal. Destructive read expressions `destval`(ν) are legal only if the variable ν is not `this`, and legal if ν is a field expression only in methods of mutator kind. As an extra explained in paragraph *6*, we can permit in observers the non-destructive read of `free` local variables, which weakens the handle's mode to `read` (cf. reduction rule {rd$_{cp}$} in paragraph *7*). Correspondingly, the type infered for non-destructive read expressions is the substitution τ[`read`/`free`] of `read` for `free` in the read variable's range type.

[null] Since `null` evaluates to a `free` nil-handle, it can be assigned `free` handle types with any valid class name c.

[new] Since object creation expression `new` c`()` evaluates to `free` handles targeting new c-objects, it gets type `free` c. It is legal if c is a valid class name.[7]

[call] Operation call expressions $e_0 \Leftarrow f(e_1, \ldots, e_n)$ are assigned the result type of the operation f in the signature $\Sigma(\mu\ c)$ of the type $\mu\ c$ inferred for receiver expression e_0. To be legal, the argument expressions' types must be mode-compatible to the corresponding parameter types of f in $\Sigma(\mu\ c)$. And if the signature marks f as a mutator, then the operation call expression is only legal in a method whose kind κ permits mutator invocations (see §5.4.2).

[7] In the presence of Java `interfaces` or `abstract` classes, it would be necessary to check that the instantiated class is a concrete classes, i.e., fully implemented.

$$[\text{var}_l]\ \frac{(x:\tau)\in\Gamma}{\Gamma,\kappa\vdash x:\tau}\qquad [\text{var}_f]\ \frac{(\texttt{this}:\mathsf{ref}\,\mu\ c)\in\Gamma\quad \vdash \mathit{FldsMths}(c)=\langle\{\ldots,x{:}\tau,\ldots\},F\rangle}{\Gamma,\kappa\vdash \texttt{this}.x:\tau}$$

$$[\text{rd}_{cp}]\ \frac{\Gamma,\kappa\vdash\nu:\mathsf{ref}\,\tau\quad \tau'=\tau[\texttt{read/free}]\quad \tau=\texttt{free<...>}\ c\Rightarrow\kappa=\mathsf{obs}\wedge\nu\in \mathit{Id}}{\Gamma,\kappa\vdash\texttt{val}(\nu):\tau'}$$

$$[\text{rd}_{dst}]\ \frac{\Gamma,\kappa\vdash\nu:\mathsf{ref}\,\tau\quad \nu\neq\texttt{this}\quad \nu=\texttt{this}.y\Rightarrow\kappa=\texttt{mut}}{\Gamma,\kappa\vdash\texttt{destval}(\nu):\tau}$$

$$[\text{null}]\ \frac{\vdash c\ \mathbf{ok}}{\Gamma,\kappa\vdash\texttt{null}:\texttt{free}\ c}\qquad [\text{new}]\ \frac{\vdash c\ \mathbf{ok}}{\Gamma,\kappa\vdash\texttt{new}\ c\texttt{()}:\texttt{free}\ c}$$

$$[\text{call}]\ \frac{\begin{array}{c}\Gamma,\kappa\vdash e:\mu\ c\quad \vdash(f:\overline{\tau_i}\xrightarrow{\kappa^\star}\tau)\in\Sigma(\mu\ c)\\ \kappa^\star=\texttt{mut}\Rightarrow\mu\in\mathbf{Wr}(\kappa)\quad \overline{\Gamma,\kappa\vdash e_i:\tau_i'}\quad \overline{\vdash\tau_i'\leq_{\mathrm{m}}\tau_i}\end{array}}{\Gamma,\kappa\vdash e{\Leftarrow}f\,(\overline{e_i}):\tau}$$

$$[\text{upd}]\ \frac{\Gamma,\kappa\vdash\nu:\mathsf{ref}\,\tau\quad \Gamma,\kappa\vdash e:\tau'\quad \vdash\tau'\leq_{\mathrm{m}}\tau\quad \nu\neq\texttt{this}\quad \nu=\texttt{this}.y\Rightarrow\kappa=\texttt{mut}}{\Gamma,\kappa\vdash\nu\texttt{=}e\texttt{;}:\mathsf{Cmd}}$$

$$[\text{ret}]\ \frac{\Gamma,\kappa\vdash e:\tau}{\Gamma,\kappa\vdash\texttt{return}\ e\texttt{;}:\tau}\qquad [\text{seq}]\ \frac{\Gamma,\kappa\vdash s_1:\mathsf{Cmd}\quad \Gamma,\kappa\vdash s_2:\tau}{\Gamma,\kappa\vdash s_1\ s_2:\tau}$$

$$[\text{if}]\ \frac{\Gamma,\kappa\vdash e_1:\mu_1\ c_1\quad \Gamma,\kappa\vdash e_2:\mu_2\ c_2\quad \Gamma,\kappa\vdash s:\mathsf{Cmd}}{\Gamma,\kappa\vdash\texttt{if}\,(e_1\,\psi\,e_2)\ \texttt{\{}s\texttt{\}}:\mathsf{Cmd}}$$

$$[\text{wh}]\ \frac{\Gamma,\kappa\vdash e_1:\mu_1\ c_1\quad \Gamma,\kappa\vdash e_2:\mu_2\ c_2\quad \Gamma,\kappa\vdash s:\mathsf{Cmd}}{\Gamma,\kappa\vdash\texttt{while}\,(e_1\,\psi\,e_2)\ \texttt{\{}s\texttt{\}}:\mathsf{Cmd}}$$

Figure 5.13: Typing rules for program terms

[upd] Assignment statements reduce to ϵ and are therefore given the special type Cmd. They are legal under the following conditions: The left-hand side is an l-value expression. The right-hand side is an expression whose type is mode-compatible to the range τ of the left-hand side. The left-hand side must not be `this`, and a field expression only inside a method of mutator kind.

[ret] The (return) type of a `return` statement is the type of its return expression. The typing rules for sequences and `if` and `while` will imply that `return` statements can only occur as the last statement of a method body s.

[seq] The (return) type of a sequence of statements is the second statement's (return) type τ. To be legal, the first statement must be of the type Cmd of continuing statements, i.e., a statement not returning from the current method.

[if] `If` statements reduce to ϵ or to their then-branch. They are given the type Cmd and it is checked that the then-branch is continuing. Moreover, to be legal, the compared expressions need to be object reference-valued expressions, i.e.,

$$\text{[val]}\ \frac{v \in [\![\tau]\!]}{\Gamma, \kappa \vdash_X\ v : \tau} \qquad\qquad \text{[nest]}\ \frac{\Gamma, \kappa \vdash_X\ s : \mu\ c}{\Gamma', \kappa' \vdash_{\tilde{\mu}',\Gamma,\kappa,X}\ \ll s \gg : \tilde{\mu}' \circ \mu\ c}$$

$$\overline{\eta_{i}{}^{\kappa_i}_{h_i}} \models \overline{\tilde{\mu}_i, \Gamma_i, \tilde{\kappa}_i} \ \Leftrightarrow_{\text{df}}\ \overline{\eta_i \models \Gamma_i} \ \wedge\ \overline{\kappa_i = \tilde{\kappa}_i} \ \wedge\ \overline{\exists o_i, \omega_i \bullet\ h_i = \langle o_i, \tilde{\mu}_i, \omega_i \rangle}$$

Figure 5.14: Typing rules for runtime terms and consistency with extended context

typeable with handle types $\mu\ c$,

[wh] `While` loops reduce to `if` statements and therefore have type Cmd. They are legal if the loop body is continuing (so that is can be prefixed to a repetition of the `while` loop) and if the compared expressions are object reference-valued expressions (to ensure validity of the produced `if` statement).

3. TYPING RUNTIME TERMS. For reasoning about the evaluation of well-formed program terms *in a small-step semantics*, it is standard to assign types also to all intermediate runtime terms. (This is unrelated to judging the validity of program p.) To this end, the typing rules are extended in a natural way to cover runtime terms. Figure 5.14 shows the two rules for the runtime-specific terms:

[val] Irreducible terms v that are values in a type's extension $[\![\tau]\!]$ must obviously be assigned the type τ. These are the locations $\mathcal{L}oc \in [\![\mathsf{ref}\ \tau]\!]$, handles $\mathcal{V} \in [\![\mu\ c]\!]$, and $\epsilon \in [\![\mathsf{Cmd}]\!]$ as the "value" to which continuing statements reduce.[8]

[nest] The type of value to which an inlined, currently executing method $\ll s \gg$ will reduce on return is predicted by determining the (return) type $\mu\ c$ of the statement s to which its body has reduced so far, and by mode-adapting this type like an eventually calculated result handle's mode μ will be adapted on return. In order to define this, we need not only the type assignment Γ' and method kind κ' of the calling method, but also of the called method, and we need the mode $\mu_{\mathbf{r}}$ of the call-link through which the call was made and relative to which the returned handle will be adapted.

For the typing of terms containing arbitrary nesting levels of inlined method bodies more contextual information is required than for the type checking of the terms in the program: The general scheme of typing rules has to be extended to include the type assignments Γ_i, method kinds κ_i, and call-link modes $\tilde{\mu}_i$ for all method nesting levels $i > 1$ in the term. This is done by annotating the turnstile symbol of typing judgments with a sequence $X = \mu_2, \Gamma_2, \kappa_2, \ldots, \mu_n, \Gamma_n, \kappa_n$. It will usually be written $X = \mu_2, \Gamma_2, \kappa_2, X_2$ (with $X_2 = \mu_3, \Gamma_3, \kappa_3, \ldots, \mu_n, \Gamma_n, \kappa_n$). All typing judgments in the program typing rule in figure 5.13 have to be annotated this way to obtain the corresponding runtime term typing rules. It shall suffice here to show this on the example of `return` terms:

[8]Note however that irreducible "term" ϵ is not an element of R.

$$[\text{ret}]\ \frac{\Gamma, \kappa \vdash_X \ e : \tau}{\Gamma, \kappa \vdash_X \ \texttt{return}\ e;\ : \tau}$$

The original rules with '$\vdash$' can then be seen as the special case '$\vdash_\epsilon$' with empty X because there are no inlined method bodies in the terms of the program.

Naturally one expects a correspondence between this annotation and the environment stack in the execution (see figure 5.13 again). An environment stack $\vec{\eta} = \overline{\eta_{i}{}_{h_i}^{\kappa_i}}$ is *type consistent* with a sequence $X = \overline{\tilde{\mu}_i, \Gamma_i, \tilde{\kappa}_i}$ of type assignments, method kinds, and call-link modes, written $\vec{\eta} \models X$, under the following conditions: Each environment is type consistent with its corresponding type assignment. The sequences of method kinds in $\vec{\eta}$ and X are the same. And the modes of the call-links in $\vec{\eta}$ are the same as the corresponding modes in X.

5.4.2 The Mode System

The ***mode system*** comprises the mode-specific checks and definitions on top of the type system which ensure that program execution is orderly *in the higher-level view* and respects the structural integrity and state encapsulation of composite objects (§5.3). Two mode-operations from the mode system also show up in reduction semantics—substitution $\mu[\texttt{read}/\texttt{free}]$ in non-destructive read and mode import $\mu_{\mathbf{r}} \circ \mu$ in return—but they are, like all mode annotations in the runtime model, only included for reasoning about the success of enforcing structural integrity and state encapsulation, and would not normally be included in an implementation of JaM.

4. STATE ENCAPSULATION: CONTROLLING THE MUTATION OF OBJECTS. Enforcing that objects change state only through their declared mutators requires JaM to control field updates and mutator method invocations. An object's fields can change through assignments and destructive reads. The syntax of base-JaM allows only access to the fields of `this`. Consequently, for shallow state encapsulation, typing rules [upd] and [rd_{dst}] (fig. 5.13) permit assignment to fields and destructive read of fields only within methods declared mutator ($\kappa = \texttt{mut}$). The invocation of methods declared mutator is limited in rule [call] through **Wr** defined in figure 5.15 to enforce shallow and composite state encapsulation:

- Calling mutators through `free` handles is always permitted ($\texttt{free} \in \mathbf{Wr}(\kappa)$) since they are expected never to belong to any sanctuary. This follows from the Unique Head invariant, that excludes `rep` ownership paths to them.
- A mutator sent through a `rep` handle, if it indeed changes the target's state, is a change in the caller's sanctuary, and thus a mutation of the composite object with the caller as representative. Hence, in order to ensure that composite objects change state only through their declared mutators, a `rep` handle can permit its source to call mutators only from within mutators. It is permitted ($\texttt{rep} \in \mathbf{Wr}(\texttt{mut})$) since the Unique Owner invariant guarantees that the target does not belong also to any other object's sanctuary.

- Since co-objects have the same owner, if it was safe for the caller to be executing a mutator ($\kappa = \texttt{mut}$) then it is for its co-objects to do the same. Hence a `co`-handle permits its source to call mutators from within mutators ($\texttt{co} \in \mathbf{Wr}(\texttt{mut})$). However, the same permission in observers would enable objects to modify themselves in observers by calls through the `co`-handle `this`.
- `read` handles provide no information about the sanctuaries to which the target might or might not belong. Hence invoking mutators through them cannot in general be guaranteed to be safe.

5. MODE COMPATIBILITY. In typing rules [upd] and [call], the type τ' of the right-hand side expression or argument expression, respectively, does not need to match exactly the, respectively, left-hand side's range type τ, or operation's parameter type τ. Normally, subclassing polymorphism would allow to weaken handles' target class to a superclass. In JaM, also the handles' modes can be adapted if a certain compatibility between modes is respected: Type $\tau' \equiv \mu'\ c'$ is ***mode-compatible*** to $\tau \equiv \mu\ c$, written $\tau' \leq_{\mathrm{m}} \tau$, if $c' = c$ and μ' is mode-compatible to μ, written $\mu' \leq_{\mathrm{m}} \mu$, as defined in figure 5.15:

- Every mode μ is trivially compatible with itself (reflexivity).
- Any mode is compatible to `read` since `read` handles give their source no mutation right on the target and make no statement about ownership and sanctuaries.
- Mode `free` is compatible with any other mode since a `free` handle is the unique initial segment of ownership paths to all co-objects reachable through it (the Unique Head property). Converting it to a non-`free` handle may create new ownership paths, but at the same time destroys all the old ownership paths with which they could be in Unique Head- or Unique Owner-conflict.

It is easy to convince oneself that treating other combinations of modes as compatible in assignments and calls would not generally be safe:[9]

6. NON-DESTRUCTIVE READ ACCESS to a variable containing a `free` handle must not create an exact copy of it since that would immediately violate the uniqueness of free paths' heads (the Unique Head property). Simply prohibiting the non-destructive read access to `free` variables would be too restrictive: The client of a `free` iterator object needs a way to call observers like `current()` on the iterator, and obtain a result, without losing the `free` reference to the iterator required to advance the iterator to the next element. For observer calls, a `read` call-link suffices, so that they

[9] `rep`, `co` or `read` $\leq_{\mathrm{m}}$ `free` would allow an object o to convert a non-`free` handle h to `free` h_f and then convert a copy of h also to h_f, thus violating Unique Head. `read` $\leq_{\mathrm{m}}$ `rep` or `co` would allow o to convert a `read` handle h to `rep` or `co`, thus making, respectively, itself or its own owner q to the owner of h's target ω. However, ω may already have an owner, and this owner is not guaranteed to be o or q, respectively, so that Unique Owner could be violated. `rep` $\leq_{\mathrm{m}}$ `co` and `co` $\leq_{\mathrm{m}}$ `rep` would allow o to convert a `co` handle h to a `rep` handle h' or vice versa. h' and and old copy of h make o an owner as well as a co-object of h's target ω. Owning ω, o owns ω's co-object o. However, o may already have an owner that is not o, so that Unique Owner could be violated.

$$
\begin{array}{lll}
\mu \leq_{\mathrm{m}} \mu & \mu_{\mathbf{r}} \circ \mathtt{read} =_{\mathrm{df}} \mathtt{read} & \\
\mu \leq_{\mathrm{m}} \mathtt{read} & \mu_{\mathbf{r}} \circ \mathtt{free} =_{\mathrm{df}} \mathtt{free} & \mathbf{Wr}(\mathtt{obs}) =_{\mathrm{df}} \{\mathtt{free}\} \\
\mathtt{free} \leq_{\mathrm{m}} \mu & \mu_{\mathbf{r}} \circ \mathtt{rep} =_{\mathrm{df}} \mathtt{read} & \mathbf{Wr}(\mathtt{mut}) =_{\mathrm{df}} \{\mathtt{free}, \mathtt{rep}, \mathtt{co}\} \\
 & \mu_{\mathbf{r}} \circ \mathtt{co} =_{\mathrm{df}} \mu_{\mathbf{r}} &
\end{array}
$$

$$
\frac{\vdash \mathit{FldsMths}(c) = \langle \Gamma, F \rangle \quad F(f) = \kappa\ \mu\ d\ f\,\mathtt{(}\overline{\mu_i\ d_i\ y_i}\mathtt{)\{\ldots\}} \qquad \overline{\mu_i \neq \mathtt{rep}} \quad \overline{\mu_i = \mathtt{co} \Rightarrow \mu_{\mathbf{r}} \neq \mathtt{read}}}{\vdash (\, f : \overline{\mu_{\mathbf{r}} \circ \mu_i\ d_i} \xrightarrow{\kappa} \mu_{\mathbf{r}} \circ \mu\ d\,) \in \Sigma(\mu_{\mathbf{r}}\ c)}
$$

Figure 5.15: Mode-specific definitions for base-JaM

can be supported by allowing to create `read` copies of `free` handles in `free` variables through non-destructive read access. Calling mutators like `Step()` is possible only if the `free` handle is taken out of the variable and used as call-link. Mutators whose purpose is only the side-effect and not the calculation of a value (`void` mutators) can return `this` to the sender. By this convention, the sender gets back the `free` handle to the receiver and can use it for further calls. Hence there is a solution for both observer and mutators calls to `free` objects.

Observe that, while `free` is compatible to `read`, mode compatibility alone is not a sufficient reason: A copy weakened only to `rep` or `co` would, respectively, still violate Unique Head or risk violating Unique Owner if the converting object has an owner. While `free` and `read` handles between the same objects can coexist (true inclusion polymorphism, "submoding"), a conversion of a `free` handle to `rep` and `co` is only safe because no `free` handle remains with the same target.

7. Import of Returned Handles. When the receiver returns a handle to the sender in reduction rule {ret} (fig. 5.8), then its mode μ may have to be adapted from the perspective of the receiver to the perspective of the sender. For defining a deterministic adaption, there should be a unique, "best" adaption $\mu_{\mathbf{r}} \circ \mu$ that is calculated from μ relative to the call-link's mode $\mu_{\mathbf{r}}$ and is mode-compatible to all other adaptions that might be desirable. This adaption, called the ***import*** of μ through $\mu_{\mathbf{r}}$ and written $\mu_{\mathbf{r}} \circ \mu$, is defined in figure 5.15:

- A returned `read` handle can only remain `read` since it provides no information that would make another mode a safe choice.
- The sender can safely import a `free` handle from the receiver as `free`, since it was the unique initial segment of ownership paths to all co-objects reachable through it, and all these old ownership paths are destroyed by the removal of the receiver's `free` handle from the graph.
- If the receiver returns a `rep` handle, however, the receiver may still possess further `rep` handles with the same target, and thus remain the target's owner. Hence the sender cannot import the handle as `free` or `rep` without risking a violation

of unique ownership (unless sender and receiver are the same). Importing it as `co` would make the sender a co-object of the target, and thus also owned by the receiver (if the receiver still owns the target). This might raise a uniqueness conflict with any old owner of the sender (unless the receiver is the old owner of the sender). Only `read` is always safe as the mode of the returned handle in the sender.

- If the returned handle is `co`, i.e., points to the receiver's co-object, the sender best imports it with the mode $\mu_{\mathbf{r}}$ of the call-link: If $\mu_{\mathbf{r}}$ is `rep` or `free`, then the sender already had an ownership path to the target by concatenation of the call-link and the receiver's `co` handle. Hence it is reasonable to shorten it to a direct $\mu_{\mathbf{r}}$ handle. In case of `free`, the imported handle will replace the unstored `free` call-link as the unique initial edge of `free` ownership paths to the receiver and all its co-objects. If $\mu_{\mathbf{r}}$ is `co` then sender and target were already co-objects through the call-link and the handle of the receiver, so that a direct `co`-handle is safe. And if $\mu_{\mathbf{r}}$ is `read` then the imported handle can only be `read`, since in a `read` call-link gives the sender no information about the receiver's owner and sanctuary memberships, and thus about a target with the same owner and sanctuary memberships as the receiver.

8. Signature of Handles. Typing rule [call] checks operation call expressions $e \Leftarrow f(e_1, \ldots, e_n)$ against the type $\overline{\tau_i} \xrightarrow{\kappa} \tau$ of f in the signature $\Sigma(\mu_{\mathbf{r}}\ c)$ of handles of the receiver expression's type $\mu_{\mathbf{r}}\ c$. The operations which can be called through a handle of type $\mu_{\mathbf{r}}\ c$ are those of objects of class c. But class c expresses the parameters' and results' modes from the perspective of the c-object, i.e., the receiver, and not from the perspective of the object using the handle for a call, i.e., the sender. If class c defines method f with *result* type $\mu\ d$ then the result type for operation f on $\mu_{\mathbf{r}}\ c$ handles must have the mode $\mu_{\mathbf{r}} \circ \mu$ to which the mode of returned μ-handles is adapted in a return step (see above). The *parameters'* modes are imported the same way from the receiver's to the sender's perspective (see figure 5.15). However, we have to reconsider the validity of this import for the modes of formal parameters since parameter values flow in the opposite direction as compared to results:

- A parameter of mode $\mu = \texttt{read}$ means that the receiver makes no assumptions about the target's place in the object graph. Hence the sender can supply handles of any mode, and any mode is mode-compatible to $\mu_{\mathbf{r}} \circ \texttt{read} = \texttt{read}$.
- If the c-object expects $\mu = \texttt{free}$ parameter values then only $\mu_{\mathbf{r}} \circ \texttt{free} = \texttt{free}$ handles of the sender (which are destroyed in the call step) can guarantee the necessary uniqueness of the initial ownership path segments.
- If the parameter has mode $\mu = \texttt{rep}$ then the receiver expects a handle to an object in its sanctuary. However, no mode on a handle *of the sender* can guarantee that the target is in the sanctuary *of the receiver*. Hence methods with `rep` parameters are not included in the signature of handles. (It would be safe to permit to call them with `null` as argument, or to call them on `this` with a `rep` argument.)
- A parameter of mode $\mu = \texttt{co}$ means that the receiver expects a handle to an object with the same owner and in the same sanctuaries as itself. If the call-link is of

mode $\mu_{\mathtt{r}} = \mathtt{read}$ then the sender has no information about the receiver's owner and sanctuary status, and thus cannot know which handle's target would have the same status. If the call-link is of mode $\mu_{\mathtt{r}} = \mathtt{co}$, a $\mu_{\mathtt{r}} \circ \mathtt{co} = \mathtt{co}$ handle of the sender is just right, since the $\mu_{\mathtt{r}} = \mathtt{co}$ means that *sender and receiver* have the same owner and are in the same sanctuaries, and $\mu_{\mathtt{r}} \circ \mathtt{co} = \mathtt{co}$ means that *sender and target* have the same owner and are in the same sanctuaries. And if the call-link is of mode $\mu_{\mathtt{r}} = \mathtt{rep}$ or $\mathtt{free}$ then only a, respectively, $\mathtt{rep}$ or $\mathtt{free}$ handle of the sender guarantees that receiver and target have the same owner, namely the sender, and are in the same sanctuaries, namely the sender's sanctuary and those enclosing it.

5.4.3 Type Correctness and Consistency

A type system's main purpose is to accept only those programs as legal whose execution never causes certain, forbidden kinds of execution errors to occur, in other words, to make the programming language "*safe*" [Car97]. The main error to prevent in object-oriented programming is the *message-not-understood* error, i.e., the attempt to invoke an operation on a receiver object that does not implement it. (Not normally forbidden is the *null-pointer* error, i.e., the attempt to make a call although the receiver expression evaluated to a nil-handle.) It has been shown repeatedly in the literature that smaller and larger subsets of *Java* are safe in this sense [IPW99, Sym97, DE97, Ohe01], including the subset on which (base-)JaM is based. It would not be difficult to extend these results to base-JaM since the addition of modes introduces no new cases where execution runs into an error. (The only operation used on modes, $\circ$, is a total operation.) But to do so would be very tedious and a distraction from our new safety property of composite state encapsulation.

In a formal setting, the mentioned execution errors mean that there is no continuation for the reduction process. Hence safety properties at the composite object level can be treated independently from the traditional, lower-level safety. What is needed as basis for composite state encapsulation is not type safety but *type consistency*: The execution of legal base-JaM programs p produces only stores and object-maps that are type consistent ($\models \mathfrak{s}, om$), and runtime terms typeable in a context type consistent with the corresponding environment stack. The latter implies in particular that the next reduction step's redex is a well-formed term.

Observe that type consistency is independent from the details of the mode system defined in §5.4.2, so that the proofs will be nearly identical for full JaM. The only necessary assumption is that the signature $\Sigma(\mu\ c)$ of handles is calculated from $\mathit{FldsMths}(c)$ by adapting the modes μ_i in it to $\mu \circ \mu_i$.

9. TYPE PRESERVATION. The standard basis for proofs about the type system is the property of *type preservation* (or its generalization to the *subject reduction* property in the presence of subtype-polymorphism): Each legal reduction step preserves type consistency and the term's type (relative to a perhaps changed annotation X' for the higher call-levels).

Lemma 1 (Type preservation) If $e,\ \vec{\eta}, \mathfrak{s}, om, \mathfrak{g} \Longrightarrow e',\ \vec{\eta}', \mathfrak{s}', om', \mathfrak{g}'$ is a reduction step defined relative to a program p that is legal, i.e., $\vdash p$ **start** e_0, then

$$\begin{array}{rl} & \Gamma, \kappa \vdash_X\ e : \tau \quad \wedge\ \vec{\eta} \models \tilde{\mu}, \Gamma, \kappa, X \quad \wedge \models \mathfrak{s}, om \\ \Rightarrow & \exists X'\text{.}\ \Gamma, \kappa \vdash_{X'}\ e' : \tau \ \wedge\ \vec{\eta}' \models \tilde{\mu}, \Gamma, \kappa, X' \ \wedge \models \mathfrak{s}', om' \end{array}$$

The proof of this lemma and other theorems uses a small technical lemma to relate mode $\hat{\mu}'$ and method suite F' of receivers $\mathbf{r}$ in the type system with their actual mode $\hat{\mu}$ and method suite F:

Lemma 2 $\quad \Gamma, \kappa \vdash_X\ \langle \mathbf{s}, \hat{\mu}, \mathbf{r} \rangle : \hat{\mu}'\ c\ \wedge\ om(\mathbf{r}) \doteq \langle \varrho_{\mathbf{r}}, F \rangle\ \wedge \models om$
$\Rightarrow\ \hat{\mu} = \hat{\mu}'\ \wedge\ \mathbf{r} \in \mathbb{O}_c\ \wedge\ \exists \Gamma_c\text{.}\ \mathit{FldsMths}(c) = \langle \Gamma_c, F \rangle \wedge \varrho_{\mathbf{r}} \models \Gamma_c$

Proof: First, $\Gamma, \kappa \vdash_X\ \langle \mathbf{s}, \hat{\mu}, \mathbf{r} \rangle : \hat{\mu}'\ c\ \Rightarrow\ \langle \mathbf{s}, \hat{\mu}, \mathbf{r} \rangle \in [\![\hat{\mu}'\ c]\!]\ \Rightarrow\ \hat{\mu} = \hat{\mu}' \wedge \mathbf{r} \in \mathbb{O}_c \cup \{\mathsf{nil}\}$. Second, if $om(\mathbf{r})$ is defined, $\mathbf{r}$ cannot be nil. This leaves $\mathbf{r} \in \mathbb{O}_c \overset{\models om}{\Longrightarrow} om(\mathbf{r}) \in [\![\mathsf{obj}\ c]\!] \overset{om(\mathbf{r}) = \langle \varrho_{\mathbf{r}}, F \rangle}{\Longrightarrow} \mathit{FldsMths}(c) = \langle \Gamma_c, F \rangle \wedge \varrho_{\mathbf{r}} \models \Gamma_c$. ■

Proof of the main lemma: $e,\ \vec{\eta}, \mathfrak{s}, om, \mathfrak{g} \Longrightarrow e',\ \vec{\eta}', \mathfrak{s}', om', \mathfrak{g}'$ means there is a multi-level context $\mathcal{E}^\star$, a redex $\hat{e}$ and a term $\hat{e}'$ such that $e = \mathcal{E}^\star[\hat{e}]$ and $e' = \mathcal{E}^\star[\hat{e}']$, and postfixes $\vec{\hat{\eta}}$ and $\vec{\hat{\eta}}'$ of $\vec{\eta}$ and $\vec{\eta}'$ such that $\hat{e},\ \vec{\hat{\eta}}, \mathfrak{s}, om, \mathfrak{g} \longrightarrow \hat{e}',\ \vec{\hat{\eta}}', \mathfrak{s}', om', \mathfrak{g}'$. Proceed by induction on the height N of the derivation tree for the reduction step, which is the same as the method nesting level of the hole in $\mathcal{E}^\star$. In the base case, $\mathcal{E}^\star$ contains no inlined method, i.e., $\mathcal{E}^\star = \mathcal{E} \in R_1^{\square}$, $\vec{\hat{\eta}} = \vec{\eta}$ and $\vec{\hat{\eta}}' = \vec{\eta}'$, and $X = \epsilon$. In the simplest case, $\mathcal{E} = \square$ and $\hat{e} = e$. Proceed by case analysis of the rule by which redex e is reduced. It determines what kind of term e and e' are, and thus how they are typed.

Let us start with the easy cases, where the environment stack is unchanged and consists only of the top-level environment: $\vec{\eta} = \vec{\eta}' = \eta_h^\kappa$. Then $\vec{\eta} \models \tilde{\mu}, \Gamma, \kappa, X$ means $\eta \models \Gamma$. First, we derive that the new term e' that it can be typed as τ or, if it is an irreducible value, that it belongs to τ's extension $[\![\tau]\!]$, from which $\Gamma, \kappa \vdash_\epsilon\ e' : \tau$ follows immediately for annotation $X' = \epsilon = X$.

$\{\mathrm{var}_l\}$: $e = x \overset{e:\tau}{\Longrightarrow} \Gamma(x) = \tau \overset{\eta \models \Gamma}{\Longrightarrow} \eta(x) = e' \in [\![\tau]\!]$

$\{\mathrm{var}_f\}$: $e = \mathtt{this}.x \overset{e:\tau}{\Longrightarrow} \left\{ \begin{array}{l} \Gamma(\mathtt{this}) = \mathsf{ref\ co}\ c \overset{\eta \models \Gamma}{\Longrightarrow} \eta(\mathtt{this}) = \ell \in \mathcal{L}oc_{\mathsf{co}\ c} \\ \overset{\models \mathfrak{s}}{\Longrightarrow} \mathfrak{s}(\ell) \in [\![\mathsf{co}\ c]\!] \overset{\mathfrak{s}(\ell) = \langle o, \mu, o \rangle}{\Longrightarrow} o \in \mathbb{O}_c \cup \{\mathsf{nil}\} \overset{om(o)\ \mathrm{defd.}}{\Longrightarrow} o \in \mathbb{O}_c \\ \left. \begin{array}{l} \overset{\models om}{\Longrightarrow} om(o) = \langle \varrho', F' \rangle \in [\![\mathsf{obj}\ c]\!] \\ \mathit{FldsMths}(c) = \langle \Gamma_c, F \rangle \wedge \Gamma_c(x) = \tau \end{array} \right\} \Rightarrow \varrho' \models \Gamma_c \wedge \varrho'(x) \in [\![\tau]\!] \end{array} \right.$

$\{\mathrm{rd}_{dst}\}$: $e = \mathtt{destval}(\ell) \overset{e:\tau}{\Longrightarrow} \Gamma, \kappa \vdash_X\ \ell : \mathsf{ref}\,\tau \Rightarrow \ell \in \mathcal{L}oc_\tau \overset{\models \mathfrak{s}}{\Longrightarrow} \mathfrak{s}(\ell) = e' \in [\![\tau]\!]$
$\{\mathrm{rd}_{cp}\}$: $e = \mathtt{val}(\ell) \overset{e:\tau}{\Longrightarrow} \exists \tilde{\tau}\text{.}\ \Gamma, \kappa \vdash_X\ \ell : \mathsf{ref}\,\tilde{\tau} \wedge \tau = \tilde{\tau}[\mathtt{read}/\mathtt{free}] \ \Rightarrow\ \ell \in \mathcal{L}oc_{\tilde{\tau}}$
$\qquad \overset{\models \mathfrak{s}}{\Longrightarrow} \mathfrak{s}(\ell) \in [\![\tilde{\tau}]\!] \ \Rightarrow\ \mathfrak{s}(\ell)[\mathtt{read}/\mathtt{free}] = e' \in [\![\tau]\!]$
$\{\mathrm{null}\}$: $e = \mathtt{null} \overset{e:\tau}{\Longrightarrow} \tau = \mathtt{free}\ c\ \Rightarrow\ e' = \langle \mathbf{r}, \mathtt{free}, \mathsf{nil} \rangle \in [\![\tau]\!]$
$\{\mathrm{new}\}$: $e = \mathtt{new}\ c()\ \Rightarrow\ \mathbf{o} \in \mathbb{O}_c\ \wedge\ \tau = \mathtt{free}\ c\ \Rightarrow\ e' = \langle \mathbf{r}, \mathtt{free}, \mathbf{o} \rangle \in [\![\tau]\!]$

$\{\text{upd}\}$: $e = \ell = \langle \mathbf{r}, \mu, \omega \rangle\,\texttt{;}\ \stackrel{e:\tau}{\Longrightarrow}\ \tau = \mathsf{Cmd}\ \Rightarrow\ e' = \epsilon \in [\![\tau]\!]$
$\{\text{if}_f\}$: $e = \texttt{if}(h_1 \psi h_2)\texttt{\{}s\texttt{\}}\ \stackrel{e:\tau}{\Longrightarrow}\ \tau = \mathsf{Cmd}\ \Rightarrow\ e' = \epsilon \in [\![\tau]\!]$
$\{\text{if}_t\}$: $e = \texttt{if}(h_1 \psi h_2)\texttt{\{}s\texttt{\}}\ \stackrel{e:\tau}{\Longrightarrow}\ \tau = \mathsf{Cmd}\ \wedge\ \Gamma, \kappa \vdash_X s : \mathsf{Cmd}\ \stackrel{e'=s}{\Longrightarrow}\ \Gamma, \kappa \vdash_X e' : \tau$
$\{\text{wh}\}$: $e = \texttt{while}(h_1 \psi h_2)\texttt{\{}s\texttt{\}} \stackrel{e:\tau}{\Longrightarrow} \tau = \mathsf{Cmd} \wedge \Gamma, \kappa \vdash_X s : \mathsf{Cmd} \wedge \Gamma, \kappa \vdash_X h_1 : \mu_1\ c_1$
$\qquad \wedge\ \Gamma, \kappa \vdash_X h_2 : \mu_2\ c_2\ \Rightarrow\ \Gamma, \kappa \vdash_X \texttt{if}(h_1 \psi h_2)\texttt{\{}s\ e\texttt{\}} : \mathsf{Cmd} \stackrel{e'=\dots,\tau=\dots}{\Longrightarrow} \Gamma, \kappa \vdash_X e' : \tau$

Since $\vec{\eta}'$ is the unchanged top-level environment, trivially $\vec{\eta}' \models \tilde{\mu}, \Gamma, \kappa, \epsilon$. And $\models \mathfrak{s}', om'$ is trivial by assumption in cases where store and object-map are unchanged.

In case of $\{\text{new}\}$, nil-handles $\langle \mathbf{o}, \mu_i, \mathsf{nil} \rangle$ are filled into the store at locations in $[\![\mathsf{ref}\ \mu_i\ c_i]\!] = \mathcal{L}oc_{\mu_i\ c_i}$ reserved for handles of these modes. Hence $\models \mathfrak{s}'$. And the new $\mathbf{o} \in \mathbb{O}_c$ is mapped to an object value with the right field locations and the right method suite for a c-object. Hence $\models om'$.

And in case of $\{\text{rd}_{dst}\}$ and $\{\text{upd}\}$, $om' = om$, so that $\models om'$. By $\models \mathfrak{s}$, $\mathfrak{s}(\ell) = \langle o, \mu, \omega \rangle$ means that for some class c, $\ell \in \mathcal{L}oc_{\mu\ c}$ and $\langle o, \mu, \omega \rangle \in [\![\mu\ c]\!]$. But then the new store value $\mathfrak{s}'(\ell) = \langle o, \mu, \mathsf{nil} \rangle$ of the $\{\text{rd}_{dst}\}$-case is in $[\![\mu\ c]\!]$, so that $\models \mathfrak{s}'$. In the $\{\text{upd}\}$-case, we have to consider what class the target $\omega' \neq \mathsf{nil}$ of the new store value $\mathfrak{s}'(\ell)$ is. $\ell \in \mathcal{L}oc_{\mu\ c}$ (see above) means that $\Gamma, \kappa \vdash_X \ell : \mathsf{ref}\ \mu\ c$. And having a typing for $e \equiv \ell = \langle \mathbf{r}, \mu', \omega' \rangle$ means $\Gamma, \kappa \vdash_X \langle \mathbf{r}, \mu', \omega' \rangle : \mu'\ c'$ with $\mu'\ c' \leq_{\mathrm{m}} \mu\ c$. Hence $\langle \mathbf{r}, \mu', \omega' \rangle \in [\![\mu'\ c']\!]$ with $\mu' \leq_{\mathrm{m}} \mu$ and $c' = c$, and thus $\omega' \in \mathbb{O}_c \cup \{\mathsf{nil}\}$. But then also $\mathfrak{s}'(\ell) = \langle o, \mu, \omega' \rangle \in [\![\mu\ c]\!] = \mathcal{L}oc_{\mu\ c}$, so that $\models \mathfrak{s}'$.

$\boxed{\{\text{ret}\}}$ Return is the case where the environment shrinks from $\vec{\eta} = \eta_h^{\kappa} \bullet \eta^{\star\kappa^\star}_{\langle \mathbf{s}, \mu^\star, \mathbf{r} \rangle}$ to η_h^{κ}.

$$\left.\begin{array}{l} \bullet\ e = \ll\!\texttt{return}\ \langle \mathbf{r}, \mu, \omega \rangle\,\texttt{;}\!\gg \\ \bullet\ \vec{\eta} \models \ldots \Rightarrow X = \mu^\star, \Gamma^\star, \kappa^\star, \epsilon \end{array}\right\} \begin{array}{l} \stackrel{e:\tau}{\Longrightarrow} \Gamma^\star, \kappa^\star \vdash_\epsilon \texttt{return}\ \langle \mathbf{r}, \mu, \omega \rangle\,\texttt{;} : \mu'\ c \wedge \tau = \mu^\star \circ \mu'\ c \\ \Rightarrow \Gamma^\star, \kappa^\star \vdash_\epsilon \langle \mathbf{r}, \mu, \omega \rangle : \mu'\ c \\ \Rightarrow \mu = \mu'\ \wedge\ \omega \in \mathbb{O}_c \cup \{\mathsf{nil}\} \\ \Rightarrow \langle \mathbf{s}, \mu^\star \circ \mu, \omega \rangle \in [\![\mu^\star \circ \mu'\ c]\!] \\ \stackrel{e'=\dots,\tau=\dots}{\Longrightarrow} e' \in [\![\tau]\!]\ \Rightarrow\ \Gamma, \kappa \vdash_{X'} e' : \tau \end{array}$$

$\models \mathfrak{s}', om'$ follows trivially from the assumption since nothing is added to $\mathfrak{s}$ nor om. The new stack $\vec{\eta}'$ is η_h^{κ}, and X' is ϵ. Since $\vec{\eta} = \eta_h^{\kappa} \bullet \eta^{\star\kappa^\star}_{\langle \mathbf{s}, \mu_{\mathbf{r}}, \mathbf{r} \rangle} \models \tilde{\mu}, \Gamma, \kappa, X$, also $\eta_h^{\kappa} \bullet \epsilon \models \tilde{\mu}, \Gamma, \kappa, \epsilon$. That is, $\vec{\eta}' \models \tilde{\mu}, \Gamma, \kappa, X'$.

$\boxed{\{\text{call}\}}$ Calls are the case where the environment stack grows.

$$\left.\begin{array}{l} \left.\begin{array}{l} \bullet\ e = \langle \mathbf{s}, \hat{\mu}, \mathbf{r} \rangle \Leftarrow f\,(\ldots) \\ \quad \stackrel{e:\tau}{\Longrightarrow} \Gamma, \kappa \vdash_X \langle \mathbf{s}, \hat{\mu}, \mathbf{r} \rangle : \hat{\mu}\ c \\ \bullet\ om(\mathbf{r}) \doteq \langle \varrho_{\mathbf{r}}, F \rangle \end{array}\right\} \begin{array}{l} \Rightarrow\ (\, f : \overline{\tau_i} \stackrel{\kappa^\star}{\to} \tau\,) \in \Sigma(\hat{\mu}\ c) \\ \stackrel{\text{Lemma 2}}{\stackrel{\models om}{\Longrightarrow}}\ FldsMths(c) \\ \qquad = \langle \Gamma_c, F \rangle \end{array} \\ \bullet\ F(f) = \kappa^\star\ \mu\ d\ f\,(\ldots)\texttt{\{}\ldots\ s\texttt{\}} \end{array}\right\} \Rightarrow \begin{array}{l} \tau = \\ \hat{\mu} \circ \mu\ d \end{array}$$
$$\left.\begin{array}{l} \ldots \\ \quad \stackrel{\models om,\ \vdash p}{\Longrightarrow} \Gamma^\star, \kappa^\star \vdash s : \mu\ d \end{array}\right\} \Rightarrow \begin{array}{l} \Gamma, \kappa \vdash_{\hat{\mu}, \Gamma^\star, \kappa^\star, \epsilon} \\ \quad \ll\! s \!\gg : \tau \end{array}$$

Since the new term e' is $\ll\! s\!\gg$, this gives us the desired $\Gamma, \kappa \vdash_{X'} e' : \tau$ with $X' = \hat{\mu}, \Gamma^\star, \kappa^\star, \epsilon$. The new stack $\vec{\eta}'$ is $\eta_h^{\kappa} \bullet \eta^{\star\kappa^\star}_{\langle \mathbf{s}, \hat{\mu}, \mathbf{r} \rangle}$, where the new top-level environment $\eta^\star$ maps identifiers x_i to locations in $[\![\mathsf{ref}\ \tau_i]\!]$. The types τ_i are determined from the declarations in method $F(f)$ or τ_i is, in case of $x_i = \mathtt{this}$, the type $\mathsf{co}\ c$ since $\mathbf{r} \in \mathbb{O}_c$. The type assignment $\Gamma^\star$ maps the same identifiers to corresponding type terms $\mathsf{ref}\,\tau_i$. Hence $\eta^\star \models \Gamma^\star$. Combined with $\vec{\eta} = \eta_h^\kappa \models \tilde{\mu}, \Gamma, \kappa, X$, therefore $\vec{\eta}' = \eta_h^\kappa \bullet \eta^{\star\kappa^\star}_{\langle \mathbf{s}, \hat{\mu}, \mathbf{r}\rangle} \models \tilde{\mu}, \Gamma, \kappa, \hat{\mu}, \Gamma^\star, \kappa^\star, \epsilon = \tilde{\mu}, \Gamma, \kappa, X'$.

Since om is unchanged, $\models om'$ by assumption. Consider the extensions of the store: $\mathtt{this}$'s location $\ell \in [\![\mathsf{ref}\ \mathsf{co}\ c]\!]$ is mapped to a corresponding handle $\langle \mathbf{r}, \mathsf{co}, \mathbf{r}\rangle \in [\![\mathsf{co}\ c]\!]$. The local variables' locations $\ell_j^z \in [\![\mathsf{ref}\,\mu_j'\ c_j']\!]$ are mapped to corresponding nil-handles $\langle \mathbf{r}, \mu_j', \mathsf{nil}\rangle \in [\![\mu_j'\ c_j']\!]$. The parameters' locations $\ell_i^y \in [\![\mathsf{ref}\,\mu_i\ c_i]\!]$ are mapped to handles $\langle \mathbf{r}, \mu_j, \omega_i\rangle$. The target classes match because of the typing of $e = \langle \mathbf{s}, \hat{\mu}, \mathbf{r}\rangle \Leftarrow f(\langle \mathbf{s}, \mu_i, \omega_i\rangle)$: Derived from the types $\mu_i\ c_i$ declared in $F(f)$, the handle signature is $(\,f : \mu \circ \mu_i\ c_i \xrightarrow{\kappa^\star} \tau\,) \in \Sigma(\hat{\mu}\ c)$. Hence the argument expressions $\langle \mathbf{s}, \mu_i, \omega_i\rangle$ must be typed with a subtype of $\hat{\mu} \circ \mu_i\ c_i$, i.e., some $\mu_i'\ c_i$. This means $\langle \mathbf{s}, \mu_i, \omega_i\rangle \in [\![\mu_i'\ c_i]\!]$, and thus $\omega_i \in c_i$. Hence $\langle \mathbf{r}, \mu_i, \omega_i\rangle \in [\![\mu_i\ c_i]\!]$. This shows that $\models \mathfrak{s}'$.

If the case that the *single-level* context $\mathcal{E}^\star = \mathcal{E}$ is not empty ($\mathcal{E} \neq \Box$), consider that typing $e = \mathcal{E}[\hat{e}]$ required to have a typing for all of its subterms, in particular redex $\hat{e}$. Since $\mathcal{E}$ contains no inlined method, the typing of $\hat{e}$ must have been in the same context. That is, $\Gamma, \kappa \vdash_X \hat{e} : \hat{\tau}$ for some $\hat{\tau}$. In conjunction with $\hat{e}, \vec{\eta}, \mathfrak{s}, om, \mathfrak{g} \longrightarrow \hat{e}', \vec{\eta}', \mathfrak{s}', om', \mathfrak{g}'$, the case of $\mathcal{E} = \Box$ above allows one to conclude that there is an X' such that $\Gamma, \kappa \vdash_{X'} \hat{e}' : \tau$ and $\vec{\eta}' \models \tilde{\mu}, \Gamma, \kappa, X'$ and $\models \mathfrak{s}', om'$. Since $\hat{e}$ and $\hat{e}'$ have the same type in the same context, *if* a type is inferred for $\mathcal{E}[\hat{e}']$ it must be the type τ of $\mathcal{E}[\hat{e}]$. The only way how the typing might fail, since it depends not only on subterms' type, is the condition on the l-value expression in an assignment or destructive read. But the result $\hat{e}'$ of a redex substitution can neither be, nor be contained in, '$\mathtt{this}$' nor '$\mathtt{this}.x$'. Therefore $\Gamma, \kappa \vdash_{X'} e' : \tau$.

In the induction step, $\mathcal{E}[\ll e'' \gg], \eta_h^\kappa \bullet \vec{\eta}, \mathfrak{s}, om, \mathfrak{g} \Longrightarrow \mathcal{E}[\ll e''' \gg], \eta_h^\kappa \bullet \vec{\eta}', \mathfrak{s}', om', \mathfrak{g}'$ is derived with hypothesis $e'', \vec{\eta}, \mathfrak{s}, om, \mathfrak{g} \Longrightarrow e''', \vec{\eta}', \mathfrak{s}', om', \mathfrak{g}'$. The typing of $e = \mathcal{E}[\ll e'' \gg]$ required to have a typing for all of its subterms, in particular $\ll e'' \gg$. Since $\mathcal{E}$ contains no inlined method, the typing of this subterm must have been in the same context, i.e., $\Gamma, \kappa \vdash_X \ll e'' \gg : \hat{\tau}$ for some $\hat{\tau}$. This typing requires that $\Gamma^\star, \kappa^\star \vdash_{X^\star} e'' : \mu\ c$ with $\hat{\tau} = \mu^\star \circ \mu\ c$ and $X = \mu^\star, \Gamma^\star, \kappa^\star, X^\star$. And $\eta_h^\kappa \bullet \vec{\eta} \models \tilde{\mu}, \Gamma, \kappa, X$ means $\vec{\eta} \models X$. With the induction hypothesis it follows that $\Gamma^\star, \kappa^\star \vdash_{X^{\star\prime}} e''' : \mu\ c$ and $\vec{\eta}' \models \mu^\star, \Gamma^\star, \kappa^\star, X^{\star\prime}$ and $\models \mathfrak{s}', om'$. Since e'' and e''' have the same type in the same context, $\Gamma, \kappa \vdash_X \mathcal{E}[\ll e'' \gg] : \tau$ with $X = \mu^\star, \Gamma^\star, \kappa^\star, X^\star$ implies the desired $\Gamma, \kappa \vdash_{\hat{X}} \mathcal{E}[\ll e''' \gg] : \tau$ with $\hat{X} = \mu^\star, \Gamma^\star, \kappa^\star, X^{\star\prime}$. Finally, $\eta_h^\kappa \bullet \vec{\eta}' \models \tilde{\mu}, \Gamma, \kappa, \mu^\star, \Gamma^\star, \kappa^\star, X^{\star\prime}$ since $\vec{\eta}' \models \mu^\star, \Gamma^\star, \kappa^\star, X^{\star\prime}$ and $\eta_h^\kappa \bullet \vec{\eta} \models \tilde{\mu}, \Gamma, \kappa, X$. ∎

10. Type System Correctness. As corollary from Lemma 1 we get a standard property of typed programming languages: The type system, by assigning types τ to the program's terms e (*static types*), correctly predicts the types of the values v to which these terms will evaluate (*dynamic types*) in environments $\vec{\eta}$ consistent with the assumptions Γ made in the typing rules:

Corollary 1 If $\vdash p$ **start** e_0 and $\models \mathfrak{s}, om$ and $\vec{\eta} \models \tilde{\mu}, \Gamma, \kappa, X$ then

$$\Gamma, \kappa \vdash_X e : \tau \wedge (e, \vec{\eta}, \mathfrak{s}, om, \mathfrak{g} \Longrightarrow^* v, \vec{\eta}', \mathfrak{s}', om', \mathfrak{g}') \wedge v \in \mathcal{L}oc \cup \mathcal{V} \cup \{\epsilon\} \Rightarrow v \in [\![\tau]\!]$$

Proof: By induction on the number of reduction steps from e to v, we get $\Gamma, \kappa \vdash_{X'} v : \tau$ with Lemma 1. Since $v \in \mathcal{L}oc \cup \mathcal{V} \cup \{\epsilon\}$, this typing is only possible by $v \in [\![\tau]\!]$. ■

Type system correctness is a *partial* notion of correctness, correctness under the condition of successful reduction to a value. A typing for a term neither says that the reduction process will ever reach an end, i.e., a configuration where no further reduction is defined, nor that, if an end is reached, it is because the term was reduced to a value $v \in \mathcal{L}oc \cup \mathcal{V} \cup \{\epsilon\}$ and not because of an execution error.

11. TYPE CONSISTENCY. With the powerful type preservation lemma, type consistency requires only to establish type consistency and typeability in the initial configuration $e_0, \eta_0, \mathfrak{s}_0, om_0, \mathfrak{g}_0$.

Theorem 1 If $e_0, \eta_0, \mathfrak{s}_0, om_0, \mathfrak{g}_0 \Longrightarrow^* e, \vec{\eta}, \mathfrak{s}, om, \mathfrak{g}$ is a reduction defined relative to a program p with $\vdash p$ **start** e_0 then there is a τ and an X such that

$$\models \mathfrak{s}, om \;\wedge\; \emptyset, \mathsf{obs} \vdash_X e : \tau \;\wedge\; \vec{\eta} \models \mathtt{read}, \emptyset, \mathtt{obs}, X$$

Proof by induction on the number N of reduction steps from e_0 to e: In the base case $N = 0$, we have $e, \vec{\eta}, \mathfrak{s}, om, \mathfrak{g} = e_0, \eta_0, \mathfrak{s}_0, om_0, \mathfrak{g}_0$. Empty store $\mathfrak{s}_0 = \emptyset$ and object-map $om_0 = \emptyset$ are trivially type-consistent. The type assignment matching the empty environment $\eta_0 = \emptyset^{\mathsf{obs}}_{\langle \mathsf{nil}, \mathsf{read}, \mathsf{nil} \rangle}$ is the empty set $\emptyset$ of type assumptions. And the empty annotation $X = \epsilon$ at the turnstile symbol matches the lack of further environments in the environment stack. Hence $\eta_0 \models \mathtt{read}, \emptyset, \mathtt{obs}, X$.

Now consider the typing of the initial term $e_0 \equiv \mathtt{new}\; c\mathtt{()}\mathtt{.main()}$: It is an operation call expression, which is typed by [call]: Receiver expression $\mathtt{new}\; c\mathtt{()}$ is typed by [new] as $\mathtt{free}\; c$ under condition $\vdash c$ **ok**, which is satisfied since p's legality, i.e., $\vdash p$ **start** e_0, guarantees $\vdash D_n$ **defs** c for $p \equiv D_1 \ldots D_n$. Since $\vdash p$ **start** e_0 ensures that class c defines a method suite F containing some $F(\mathtt{main}) \doteq \mathtt{mut}\; \tau'\; \mathtt{main()\{\ldots\}}$ without parameters, $\Sigma(\mathtt{free}\; c)$ contains $\mathtt{main} : \epsilon \xrightarrow{\kappa} \tau$ with defined adaption $\tau = \mathtt{free} \circ \tau'$. $\mathtt{main}$'s kind κ is irrelevant since the receiver expression's mode is $\mathtt{free}$. The lack of argument expressions in e_0 matches the lack of parameters in $F(\mathtt{main})$. Hence $\epsilon, \mathsf{obs} \vdash_X e_0 : \tau$.

In the induction step $N \rightarrow N+1$, reduction $e_0, \eta_0, \mathfrak{s}_0, om_0, \mathfrak{g}_0 \Longrightarrow^* e', \vec{\eta}', \mathfrak{s}', om', \mathfrak{g}'$ is continued $e', \vec{\eta}', \mathfrak{s}', om', \mathfrak{g}' \Longrightarrow e, \vec{\eta}, \mathfrak{s}, om, \mathfrak{g}$. From the induction hypothesis's $\models \mathfrak{s}', om'$ and $\epsilon, \mathsf{obs}' \vdash_{X'} e' : \tau'$ with $\vec{\eta}' \models \mathtt{read}', \epsilon, \mathtt{obs}, X'$, the theorem follows by type preservation (Lemma 1). ■

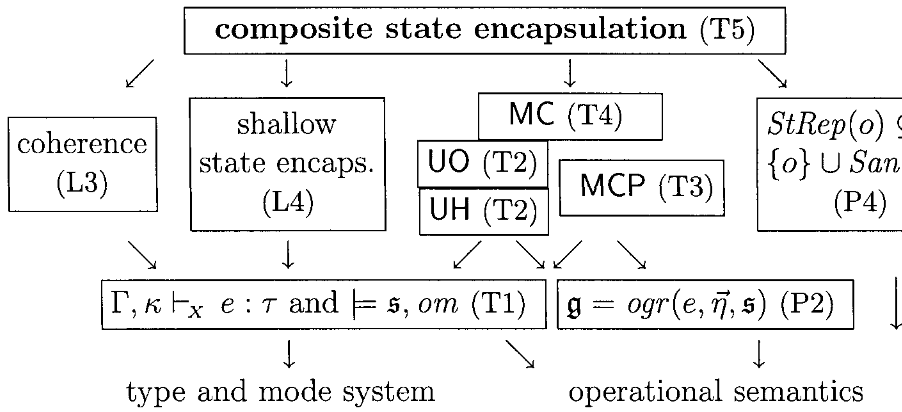


Figure 5.16: Dependency of proven properties

5.5 Integrity of the Higher-Level View

This section constructs bottom-up proofs for more and more complex properties, with composite state encapsulation in base-JaM as the ultimate goal. Figure 5.16 shows on which more basic properties which more complex properties depend.[10]

1. The ownership paths in all object graphs reachable in the execution of legal base-JaM programs, share targets so that they satisfy the Unique Owner and Unique Head integrity invariants (Theorem 2).
2. The structure of mutator access as recorded in the environment stack during the execution of legal base-JaM programs is always consistent with ownership paths and sanctuaries as captured in the integrity invariants Mutator Control and Mutator Control Path (Theorems 3 and 4).
3. Each change of objects' state during a step in the execution of legal base-JaM programs respects the state encapsulation of composite objects (Theorem 5).

5.5.1 Structural Integrity of Object Ownership

Theorem 2 If $e_0,\ \eta_0, \mathfrak{s}_0, om_0, \mathfrak{g}_0 \Longrightarrow^* e',\ \vec{\eta}', \mathfrak{s}', om', \mathfrak{g}'$ is a reduction defined relative to a program p with $\vdash p\ \ \mathbf{start}\ e_0$ then

$$\mathfrak{g}' \models \mathsf{UH}, \mathsf{UO}$$

Proof by induction on the number N of reduction steps from e_0 to e': In the base case $N = 0$, $\mathfrak{g}'$ is the empty graph $\mathfrak{g}_0 = \emptyset$, which trivially satisfies UO and UH. In the induction step $N \to N + 1$, reduction $e_0,\ \eta_0, \mathfrak{s}_0, om_0, \mathfrak{g}_0 \Longrightarrow^* e,\ \vec{\eta}, \mathfrak{s}, om, \mathfrak{g}$ is continued $e,\ \vec{\eta}, \mathfrak{s}, om, \mathfrak{g} \Longrightarrow e',\ \vec{\eta}', \mathfrak{s}', om', \mathfrak{g}'$. By induction hypothesis, $\mathfrak{g} \models \mathsf{UO}, \mathsf{UH}$. The question is whether the step to $\mathfrak{g}'$ preserves UO and UH.

[10]Note that dependency arrows are opposite to the order in the bottom-up proof.

Properties UH and UO are stated over potential access paths of mode free and rep (§5.3.2). From the definition of potential access paths it is obvious that such paths are made of free, rep, and co-edges in the extended graph $\mathfrak{g}^\star$. A violation of UH or UO could at most be introduced in reduction steps that *increase* the multiplicity of such edges in $\mathfrak{g}$, i.e., if $\mathfrak{g}' = \mathfrak{g} \oplus o \xrightarrow{\mu} \omega \ldots$ with $\mu \in \{\texttt{free}, \texttt{rep}, \texttt{co}\}$ and $o, \omega \in \mathbb{O}$. The addition of read edges and nil-handles, and the removal of edges can be ignored.

A look at the context rules shows that the changes to the object graph are absolutely independent of the term context $\mathcal{E}^\star$ surrounding the redex $\hat{e}$ in $e = \mathcal{E}^\star[\hat{e}]$. Hence we can move directly to a case analysis of the rule by which redex $\hat{e}$ is reduced. In case of $\{\text{var}_l\}$, $\{\text{var}_f\}$, $\{\text{rd}_{dst}\}$, $\{\text{null}\}$, $\{\text{if}_t\}/\{\text{if}_f\}$, and $\{\text{wh}\}$, the object graph is unchanged or edges are removed, so that the preservation of UH and UO is trivial. In the other cases, we may need the existence of *some* typing $\hat{\Gamma}, \hat{\kappa} \vdash_\epsilon \hat{e} : \hat{\tau}$ for the redex. It follows from the typing $\Gamma, \kappa \vdash_X e : \tau$ guaranteed for the whole term (Theorem 1).

$\boxed{\{\text{new}\}}$ In case of object creation, the added edge $\mathbf{r} \xrightarrow{\texttt{free}} \mathbf{o}$ targets a *fresh* object $\mathbf{o}$. By definition, $\mathbf{o}$ therefore neither is targeted by old handles, nor is the source of old (co) handles in the object graph. $\mathbf{r} \xrightarrow{\texttt{free}} \mathbf{o}$ is the only new potential access path in $\mathfrak{g}'$, and there is no old free or rep path with which it could be in UH- or UO-conflict.

$\boxed{\{\text{rd}_{cp}\}}$ In case of non-destructive read with $\hat{e} = \texttt{val}(\ell)$ and $\mathfrak{s}(\ell) = \langle o, \mu, \omega \rangle$, the multiplicity of edge $o \xrightarrow{\mu[\texttt{read}/\texttt{free}]} \omega$ is increased. If $\mu = \texttt{free}$ or read, the edge has the harmless mode read. If $\mu = \texttt{rep}$ or co, the edge is the same as the handle $\mathfrak{s}(\ell)$ and thus existed already in $\mathfrak{g} = ogr(e, \vec{\eta}, \mathfrak{s})$ (Proposition 2), so that further increasing its multiplicity cannot introduce violations of UH nor UO.

$\boxed{\{\text{upd}\}}$ In case of assignment with $\hat{e} = \ell = \langle o, \hat{\mu}, \hat{\omega} \rangle$ and $\ell \in \mathcal{L}oc_{\mu\, c}$, the multiplicity of $o \xrightarrow{\mu} \hat{\omega}$ is increased while that of $o \xrightarrow{\hat{\mu}} \hat{\omega}$ is decreased (if $\hat{\omega} \neq \textsf{nil}$). If $\mu = \hat{\mu}$, this means the only change from $\mathfrak{g}$ to $\mathfrak{g}'$ is the decrease of ℓ's old value's multiplicity. Handle $\langle o, \hat{\mu}, \hat{\omega} \rangle$ in term $\hat{e}$ means that the multiplicity-decreased edge indeed existed in $\mathfrak{g} = ogr(e, \vec{\eta}, \mathfrak{s})$ (Proposition 2). Typing $\hat{\Gamma}, \hat{\kappa} \vdash_\epsilon \hat{e} : \hat{\tau}$ of the redex presupposes $\hat{\Gamma}, \hat{\kappa} \vdash_\epsilon \ell : \textsf{ref}\, \mu\, c$ and $\hat{\Gamma}, \hat{\kappa} \vdash_\epsilon \langle o, \hat{\mu}, \hat{\omega} \rangle : \hat{\mu}\, c$ with $\hat{\mu} \leq_{\text{m}} \mu$. By the definition of $\leq_{\text{m}}$ (§5.4.2) then $\mu = \texttt{free}$ implies $\hat{\mu} = \texttt{free}$, $\mu = \texttt{rep}$ implies $\hat{\mu} = \texttt{rep}$ or free, and $\mu = \texttt{co}$ implies $\hat{\mu} = \texttt{co}$ or free. That is, in all cases of where an edge's multiplicity is increased because of $\mu \neq \hat{\mu}$, $\hat{\mu} = \texttt{free}$. But then the outer induction hypothesis $\mathfrak{g} \models \textsf{UH}$ guarantees that edge $o \xrightarrow{\hat{\mu}} \hat{\omega} \in \mathfrak{g}$ is the head of *all* ownership paths to $\hat{\omega}$ and its co-objects, and its multiplicity is 1. Consequently, in the intermediate graph $\mathfrak{g}'' = \mathfrak{g} \ominus o \xrightarrow{\hat{\mu}} \hat{\omega}$ there is *no* ownership path to $\hat{\omega}$ and its co-objects. Now consider the addition in $\mathfrak{g}' = \mathfrak{g}'' \oplus o \xrightarrow{\mu} \hat{\omega}$:

- In case of $\mu = \texttt{rep}$, where there are no new co edges, *all* new ownership paths start with $o \xrightarrow{\texttt{rep}} \hat{\omega}$ and go to $\hat{\omega}$ and its co-objects. This cannot cause any UH- or UO-conflicts in $\mathfrak{g}'$ since $\hat{\omega}$ and co-objects are unowned in $\mathfrak{g}''$.
- In case of $\mu = \texttt{co}$, the addition of $o \xrightarrow{\texttt{co}} \hat{\omega}$ entails the appearence also of its inverse $o \xleftarrow{\texttt{co}} \hat{\omega}$ in $\mathfrak{g}'^\star$. These two may give raise to new free or rep paths if they extend

old ones. $o \xleftarrow{\texttt{co}} \hat{\omega}$ cannot extend old ownership paths since there were no ownership paths to $\hat{\omega}$ in $\mathfrak{g}''$. Old ownership paths to o might be extended by $o \xrightarrow{\texttt{co}} \hat{\omega}$ and further co-paths to new ownership paths of the same mode that target $\hat{\omega}$, its old co-objects or, if further extended by inverse $\hat{\omega} \xrightarrow{\texttt{co}} o$, o and its co-objects.

$\mathfrak{g} \models \mathsf{UO}$ guarantees that o has a unique owner q. That is, the source of all old ownership paths to o is q. Since extensions does not change the path's source, besides the old also all the new ownership paths to $\hat{\omega}$, to o, and to their co-objects have the source q. There is no UO-conflict.

$\mathfrak{g} \models \mathsf{UH}$ guarantees that if there is a `free` path among the old ownership paths to o, then they all have the same head and its multiplicity is one. Since all extensions of such paths have the same head h, and since there are no other ownership paths to $\hat{\omega}$, to o, and to their co-objects, all ownership paths to them have head h of multiplicity one. There is no UH-conflict.

$\boxed{\{\text{ret}\}}$ In case of a return redex with $\hat{e} = \ll\!\texttt{return}\ \langle \mathbf{r}, \mu, \mathbf{o}\rangle\,;\!\gg$ and top-environment η_h^κ with $h = \langle \mathbf{s}, \mu_\mathbf{r}, \mathbf{r}\rangle$, the multiplicity of $\mathbf{s} \xrightarrow{\mu_\mathbf{r}\circ\mu} \mathbf{o}$ is increased, while those of $\mathbf{r} \xrightarrow{\mu} \mathbf{o}$ and $\mathbf{s} \xrightarrow{\mu_\mathbf{r}} \mathbf{r}$ are decreased (if $\mathbf{o} \neq \mathsf{nil}$). Note that the decreased edges indeed exist in $\mathfrak{g} = ogr(e, \vec{\eta}, \mathfrak{s})$ (Proposition 2) because of handles $\langle \mathbf{r}, \mu, \mathbf{o}\rangle$ in term e and $\langle \mathbf{s}, \mu_\mathbf{r}, \mathbf{r}\rangle$ in the top-environment. Consider the mode μ: In case of $\mu = \texttt{rep}$ or `read`, the new edge has the harmless mode `read`. In case of $\mu = \texttt{free}$, the new edge has mode `free` and thus establishes `free` paths in $\mathfrak{g}'$ from $\mathbf{s}$ to $\mathbf{o}$ and its co-objects. On the other hand, the receiver's old `free` handle $\mathbf{r} \xrightarrow{\mu} \mathbf{o}$ was by $\mathfrak{g} \models \mathsf{UH}$ the initial edge in *all* ownership paths to $\mathbf{o}$ and its co-objects. All these are destroyed in $\mathfrak{g}'$. Hence there can be no new UH- nor UO-conflict between new and unchanged potential access paths in $\mathfrak{g}'$. Most complicated is the case of $\mu = \texttt{co}$. Here, the new edge has mode $\mu_\mathbf{r}\circ\texttt{co} = \mu_\mathbf{r}$ and may, depending on this mode, be used to build new potential access paths $\pi \in PAP_{\mathfrak{g}'}(o', \mu', \omega')$. However, for each of them there is a precursor $\pi' \in PAP_{\mathfrak{g}}(o', \mu', \omega')$:

- If $\mu_\mathbf{r} = \texttt{free}$, new μ'-paths can only be extensions $\pi = \mathbf{s} \xrightarrow{\mu_\mathbf{r}} \mathbf{o} \xrightarrow{\texttt{co}}{}^* \omega'$ of the new edge by co-edges in $\mathfrak{g}'$ (actually, in $\mathfrak{g}'^*$). The co-edges must be old, since the only new edge in $\mathfrak{g}'$ has mode $\mu_\mathbf{r} \neq \texttt{co}$. Hence a `free` path $\pi' = \mathbf{s} \xrightarrow{\mu_\mathbf{r}} \mathbf{r} \xrightarrow{\texttt{co}} \mathbf{o} \xrightarrow{\texttt{co}}{}^* \omega'$ existed already in $\mathfrak{g}$. By $\mathfrak{g} \models \mathsf{UH}$ it ensured that the `free` call-link $\mathbf{s} \xrightarrow{\mu_\mathbf{r}} \mathbf{r}$ is the initial edge of all ownership paths to ω' and has multiplicity one. Consequently, decreasing the call-link's multiplicity in $\mathfrak{g}'$ destroys all old ownership paths to $\mathbf{o}$ and ω'. Hence the multiplicity of the new `free` edge $\mathbf{s} \xrightarrow{\mu_\mathbf{r}\circ\texttt{free}} \mathbf{o}$ must be 1, and must be the start of all new ownership paths to ω'. There is neither a UH- nor UO-conflict.
- If $\mu_\mathbf{r} = \texttt{rep}$ then, analogously, the new μ'-paths can only be extensions $\pi = \mathbf{s} \xrightarrow{\mu_\mathbf{r}} \mathbf{o} \xrightarrow{\texttt{co}}{}^* \omega'$ of the new edge by *old* co-edges in $\mathfrak{g}'$, and there was an old `rep` path $\pi' = \mathbf{s} \xrightarrow{\mu_\mathbf{r}} \mathbf{r} \xrightarrow{\texttt{co}} \mathbf{o} \xrightarrow{\texttt{co}}{}^* \omega'$ in $\mathfrak{g}$. This path by $\mathfrak{g} \models \mathsf{UH}$ excludes any *old* `free` path π'' to ω'. Since the only *new* potential access paths in $\mathfrak{g}'$ have mode $\mu_\mathbf{r} \neq \texttt{free}$, there is no new UH-conflict. And by $\mathfrak{g} \models \mathsf{UO}$, `rep` path π' ensured that all *old*

ownership paths to ω' have source $\mathbf{s}$. Since also all *new* ownership paths π have source have source $\mathbf{s}$, there is no UO-conflict.

- If $\mu_{\mathbf{r}} = \mathtt{co}$ then, besides the $h = \mathbf{s} \xrightarrow{\mathtt{co}} \mathbf{o}$, also the multiplicity of its implicit inverse $h^{-1} = \mathbf{s} \xleftarrow{\mathtt{co}} \mathbf{o}$ in $\mathfrak{g}'^{\star}$ is increased. These edges have precursors $\pi_h = \mathbf{s} \xrightarrow{\mathtt{co}} \mathbf{r} \xrightarrow{\mathtt{co}} \mathbf{o}$ and $\pi_h^{-1} = \mathbf{s} \xleftarrow{\mathtt{co}} \mathbf{r} \xleftarrow{\mathtt{co}} \mathbf{o}$ in $\mathfrak{g}^{\star}$. All new ownership paths π in $\mathfrak{g}'^{\star}$ must contain h or h^{-1} as non-head edge. But they all have a precursor in $\mathfrak{g}^{\star}$ with π_h and π_h^{-1} in place of h and h^{-1}. There can be no new UH- nor UO-conflict.

$\boxed{\{\text{call}\}}$ Operation calls are the most tedious case. If $\hat{e} = \langle \mathbf{s}, \hat{\mu}, \mathbf{r}\rangle \Leftarrow f(\langle \mathbf{s}, \hat{\mu}_i, \mathbf{o}_i\rangle)$ and $om(\mathbf{r}) = \langle \varrho_{\mathbf{r}}, F\rangle$ and $F(f) = \kappa^{\star}\ \mu\ d\, f\,(x_i\ \mu_i\ d_i)\{\ldots\}$, then the multiplicity of self-link $\mathbf{r} \xrightarrow{\mathtt{co}} \mathbf{r}$ and received handles $\mathbf{r} \xrightarrow{\mu_i} \mathbf{o}_i$ is increased while that of sent handles $\mathbf{s} \xrightarrow{\hat{\mu}_i} \mathbf{o}_i$ is decreased (unless $\mathbf{o}_i = \mathsf{nil}$). Note that handles $\langle \mathbf{r}, \hat{\mu}_i, \mathbf{o}_i\rangle$ in term e mean that the removed edges indeed existed in $\mathfrak{g} = ogr(e, \vec{\eta}, \mathfrak{s})$ (Proposition 2). If there are n parameters, the new graph $\mathfrak{g}'$ is the final graph $\mathfrak{g}_n$ in the sequence $\mathfrak{g}, \mathfrak{g}_0, \ldots, \mathfrak{g}_n$ of graphs with $\mathfrak{g}_0 = \mathfrak{g} \oplus \mathbf{r} \xrightarrow{\mathtt{co}} \mathbf{r}$ and $\mathfrak{g}_i = \mathfrak{g}_{i-1} \ominus \mathbf{s} \xrightarrow{\hat{\mu}_i} \mathbf{o}_i \oplus \mathbf{r} \xrightarrow{\mu_i} \mathbf{o}_i$ for $i > 0$. Show $\mathfrak{g}_i \models \mathsf{UH}, \mathsf{UO}$ for $i = 1, \ldots, n$ by induction on the number k of non-`null` arguments. Let i be the index of the last, the k^{th} non-`null` argument, so that all graphs following $\mathfrak{g}_i$ are not actually changed: $\mathfrak{g}_i = \mathfrak{g}_{i+1} = \ldots = \mathfrak{g}_n = \mathfrak{g}'$. In the base case, the self-link is the only added edge. It cannot introduce a UH- or UO-conflict since for every new `free` or `rep` path $\pi \in PAP_{\mathfrak{g}'}(o', \mu', \omega')$ containing it, there was already a potential access path $\pi' \in PAP_{\mathfrak{g}'}(o', \mu', \omega')$ with the self-link cut out and the same head in $\mathfrak{g}'$. In the induction step $k-1 \to k$, the graph $\mathfrak{g}_{i-1}$ with $k-1$ transfered handles still satisfies UH and UO by induction hypothesis. The question is, if $\mathfrak{g}_i = \mathfrak{g}_{i-1} \ominus \mathbf{s} \xrightarrow{\hat{\mu}_i} \mathbf{o}_i \oplus \mathbf{r} \xrightarrow{\mu_i} \mathbf{o}_i$ preserves them.

Let us derive how the received handle's mode μ_i must relate to the sent handle's modes $\hat{\mu}_i$. Typing $\hat{\Gamma}, \hat{\kappa} \vdash_{\epsilon} \hat{e} : \hat{\tau}$ of the redex means three things:

$$
\left.\begin{array}{l}
\left.\begin{array}{l}
\bullet\ \hat{\Gamma}, \hat{\kappa} \vdash_{\epsilon} \langle \mathbf{s}, \hat{\mu}, \mathbf{r}\rangle : \hat{\mu}'\ c \wedge om(\mathbf{r}) \doteq \langle \varrho_{\mathbf{r}}, F\rangle \\
\quad \overset{\text{Lemma 2}}{\underset{\models om}{\Longrightarrow}}\ \hat{\mu} = \hat{\mu}' \wedge FldsMths(c) = \langle \Gamma_c, F\rangle \\
\bullet\ (\, f : \overline{\tau_i} \xrightarrow{\kappa^{\star}} \tau\,) \in \Sigma(\hat{\mu}'\ c)
\end{array}\right\} \Rightarrow \begin{array}{l} \tau_i = \hat{\mu}' \circ \mu_i\ d_i \\ \wedge\ \mu_i \neq \mathtt{rep} \end{array} \\
\bullet\ \hat{\Gamma}, \hat{\kappa} \vdash_{\epsilon} \langle \mathbf{s}, \hat{\mu}_i, \mathbf{o}_i\rangle : \hat{\mu}_i\ d_i \ \wedge\ \hat{\mu}_i\ d_i \leq_{\mathrm{m}} \tau_i
\end{array}\right\} \Rightarrow \begin{array}{l} \langle \mathbf{s}, \hat{\mu}_i, \mathbf{o}_i\rangle \in [\![\hat{\mu}_i\ d_i]\!] \\ \wedge\ \hat{\mu}_i \leq_{\mathrm{m}} \hat{\mu}' \circ \mu_i \end{array}
$$

Observe above that $\mu_i \neq \mathtt{rep}$. This leaves $\mu_i = \mathtt{free}$ and $\mathtt{co}$ as relevant cases.

If $\mu_i = \mathtt{free}$ then $\hat{\mu}' \circ \mu_i = \mathtt{free}$, so that $\hat{\mu}_i \leq_{\mathrm{m}} \hat{\mu}' \circ \mu_i$ must be `free`. But if sent handle $\mathbf{s} \xrightarrow{\hat{\mu}_i} \mathbf{o}_i$ in $\mathfrak{g}_{i-1}$ is `free`, then by induction hypothesis $\mathfrak{g}_{i-1} \models \mathsf{UH}$ *all* ownership paths to $\mathbf{o}_i$ and its co-objects started with this handle. All these paths will be destroyed in $\mathfrak{g}_i$ by the argument links' removal. And the only new ownership paths in $\mathfrak{g}_i$ are those through `free` handle $\mathbf{r} \xrightarrow{\mu_i} \mathbf{o}_i$. Hence there can be no new UH- nor UO-conflict in $\mathfrak{g}_i$.

Similar to return steps, the case of $\mu_i = \mathtt{co}$ is the most complicated, but this time even more so since the subcases are less uniform to deal with. Beside parameter link $h = \mathbf{r} \xrightarrow{\mathtt{co}} \mathbf{o}_i$, we also find its inverse $h^{-1} = \mathbf{r} \xleftarrow{\mathtt{co}} \mathbf{o}_i$ as the new edges in $\mathfrak{g}_i^{\star}$. All new potential access paths $\pi \in PAP_{\mathfrak{g}_i}(o', \mu', \omega')$ in $\mathfrak{g}_i$ must contain h or h^{-1}. Hence new potential access paths π of mode $\mathtt{co}$ can only exist between $\mathbf{r}$ and $\mathbf{o}_i$ and their old co-objects. And all potential access paths π of mode $\mathtt{free}$ or $\mathtt{rep}$ must be extensions $\pi = o' \xrightarrow{\mu'} q' \bullet \pi' \bullet h \bullet \pi''$ or $o' \xrightarrow{\mu'} q' \bullet \pi' \bullet h^{-1} \bullet \pi''$ of an unchanged $\mathtt{rep}$ or $\mathtt{free}$ edge $o' \xrightarrow{\mu'} q'$ and some unchanged $\mathtt{co}$-edges π' by parameter link h or its inverse h^{-1} and by further $\mathtt{co}$-edges π'' (which constitute a $\mathtt{co}$-path). For UO, this means that old owners of $\mathbf{r}$ (and its old co-objects) become also owners of $\mathbf{o}_i$ and its old co-objects, and old owners of $\mathbf{o}_i$ (and its old co-objects) become also owners of $\mathbf{r}$ and its old co-objects.

First, consider what the sent handle tells us about ownership paths to $\mathbf{o}_i$. On one side, the rule for handle-signature $\Sigma(\hat{\mu}'\ c)$ with parameter mode $\mu_i = \mathtt{co}$ ensures that neither the call-link's mode $\hat{\mu} = \hat{\mu}'$ nor the sent handle's mode $\hat{\mu}_i \leq_{\mathrm{m}} \hat{\mu}' \circ \mu_i = \hat{\mu}' \circ \mathtt{co} = \hat{\mu}'$ are $\mathtt{read}$. On the other side, argument link $\mathbf{s} \xrightarrow{\hat{\mu}_i} \mathbf{o}_i$ in $\mathfrak{g}_{i-1}$ allows the following conclusions:

- If $\hat{\mu}_i = \mathtt{free}$, then $\mathbf{s} \xrightarrow{\hat{\mu}_i} \mathbf{o}_i$ was by induction hypothesis $\mathfrak{g}_{i-1} \models \mathsf{UH}$ the initial edge in all ownership paths to $\mathbf{o}_i$ and its co-objects. All these disappear in $\mathfrak{g}_i$ by the decrease of its multiplicity from one to zero. Hence there can be no old ownership path $o' \xrightarrow{\mu'} q' \bullet \pi'$ to $\mathbf{o}_i$ which $h \bullet \pi''$ could extend.
- If $\hat{\mu}_i = \mathtt{rep}$, then through $\mathbf{s} \xrightarrow{\hat{\mu}_i} \mathbf{o}_i$ there were $\mathtt{rep}$ paths $\mathbf{s} \xrightarrow{\hat{\mu}_i} \mathbf{o}_i \xrightarrow{\mathtt{co}}{}^* \omega'$ to $\mathbf{o}_i$ and all its old co-objects ω' in $\mathfrak{g}_{i-1}$. They ensure by induction hypothesis $\mathfrak{g}_{i-1} \models \mathsf{UO}$, that the source of all old ownership paths $o' \xrightarrow{\mu'} q' \bullet \pi'$ to these objects is $\mathbf{s}$. And they exclude by induction hypothesis $\mathfrak{g}_{i-1} \models \mathsf{UH}$ that any of them is a $\mathtt{free}$ path. But then all extensions of unchanged ownership paths $o' \xrightarrow{\mu'} q' \bullet \pi'$ to $\mathbf{o}_i$ by $h \bullet \pi''$ must have mode $\mu' = \mathtt{rep}$ (hence no UH-conflict here), and their source o' is $\mathbf{s}$ (hence no UO-conflict here).
- If $\hat{\mu}_i = \mathtt{co}$, then argument link $\mathbf{s} \xrightarrow{\mathtt{co}} \mathbf{o}_i$ has an inverse $\mathbf{s} \xleftarrow{\mathtt{co}} \mathbf{o}_i$ in $\mathfrak{g}^{\star}_{i-1}$.

Second, consider the receiver expression $\langle \mathbf{s}, \hat{\mu}, \mathbf{r} \rangle$ in e. tells us about ownership paths to $\mathbf{r}$: By $\mathfrak{g} = ogr(e, \vec{\eta}, \mathfrak{s})$ (Proposition 2), there must be a corresponding edge $\mathbf{s} \xrightarrow{\hat{\mu}} \mathbf{r}$ in $\mathfrak{g}$, the call-link. Since it is not removed, it still exists in $\mathfrak{g}_{i-1}$:

- If $\hat{\mu} = \hat{\mu}' = \mathtt{free}$ then the $\mathtt{free}$ call-link $\mathbf{s} \xrightarrow{\hat{\mu}} \mathbf{r}$ in $\mathfrak{g}_{i-1}$ means by induction hypothesis $\mathfrak{g}_{i-1} \models \mathsf{UH}$ that it was the head of all old ownership paths to $\mathbf{r}$ and its co-objects. Hence all extensions $\pi = \pi' \bullet \mathbf{r} \xrightarrow{\mathtt{co}} \mathbf{o}_i \bullet \pi''$ of unchanged ownership paths π' with target $\mathbf{r}$ must start with call-link $\mathbf{s} \xrightarrow{\hat{\mu}} \mathbf{r}$. On the other side, $\hat{\mu}_i \leq_{\mathrm{m}} \hat{\mu}' \circ \mu_i = \mathtt{free}$ means that $\hat{\mu}_i$ is $\mathtt{free}$. Hence, as shown above, there are no other new ownership paths, and the old ownership paths to $\mathbf{o}_i$ and its old co-objects have disappeared in $\mathfrak{g}_i$. The only new ownership paths are $\mathtt{free}$ with initial edge $\mathbf{s} \xrightarrow{\hat{\mu}} \mathbf{r}$, and the only unchanged ownership paths with the same targets ($\mathbf{o}_i$ and $\mathbf{r}$ and their co-objects) are $\mathtt{free}$ paths with initial edge $\mathbf{s} \xrightarrow{\hat{\mu}} \mathbf{r}$ (targeting $\mathbf{r}$ and its co-objects). There is no new UH- nor UO-conflict.
- If $\hat{\mu} = \hat{\mu}' = \mathtt{rep}$ then the $\mathtt{rep}$ call-link $\mathbf{s} \xrightarrow{\hat{\mu}} \mathbf{r}$ in $\mathfrak{g}_{i-1}$ means by induction hypothesis $\mathfrak{g}_{i-1} \models \mathsf{UH}$ that all unchanged ownership paths to $\mathbf{r}$ and its co-objects are $\mathtt{rep}$ paths. And by induction hypothesis $\mathfrak{g}_{i-1} \models \mathsf{UO}$, all these ownership paths must

have source $\mathbf{s}$. Consequently, all extensions $\pi = \pi' \bullet \mathbf{r} \xrightarrow{\texttt{co}} \mathbf{o}_i \bullet \pi''$ of unchanged $\mathbf{r}$-targeting ownership paths π' are `rep` paths (a), and $\mathbf{s}$ is their source (b). On the other side, $\hat{\mu}_i \leq_{\mathrm{m}} \hat{\mu}' \circ \mu_i = \texttt{rep}$ means that $\hat{\mu}_i$ is `rep` or `free`. In case of `rep`, as shown above, all unchanged ownership paths to $\mathbf{o}_i$ and its co-objects are `rep` paths (a), with source $\mathbf{s}$ (b), and also all new ownership paths by extending them are `rep` (a), and have source $\mathbf{s}$ (b). In case of `free`, as shown above, there are no other new ownership paths, and the old ownership paths to $\mathbf{o}_i$ and its old co-objects have disappeared. That is, in both cases, all new ownership paths are `rep` (a) with source $\mathbf{s}$ (b), and the unchanged ownership paths to their targets ($\mathbf{o}_i$ and $\mathbf{r}$ and their co-objects) are also `rep` (a) with source $\mathbf{s}$ (b). There is no new UH-conflict and no new UO-conflict.

- If $\hat{\mu} = \texttt{co}$ then $\hat{\mu}_i \leq_{\mathrm{m}} \hat{\mu}' \circ \mu_i = \texttt{co}$ means that $\hat{\mu}_i$ is `co` or `free`. If `free` then, as shown above, the old ownership paths to $\mathbf{o}_i$ and its old co-objects have disappeared in $\mathfrak{g}_i$ and cannot give raise to new ownership paths: All new ownership paths π are extensions of unchanged ownership paths $\pi' \in P =_{\mathrm{df}} \bigcup_{o'} PAP_{\mathfrak{g}_i}(o', \texttt{free}, \mathbf{r}) \cup PAP_{\mathfrak{g}_i}(o', \texttt{rep}, \mathbf{r})$ to $\mathbf{r}$, and thus have the same initial edges $h \in H =_{\mathrm{df}} \mathit{first}(P)$ (there are no new `free` or `rep` edges in $\mathfrak{g}_i^\star$). And all unchanged ownership paths π'' with the same targets, namely $\mathbf{r}$ and its old co-objects, also have initial edges $h \in H$ since they are themselves $\mathbf{r}$-targeting paths in P, or since they can be extended by the old `co`-links between $\mathbf{r}$ and this co-object to an $\mathbf{r}$-targeting path in P with the same initial edge. But if all new ownership paths—and the unchanged ownership paths sharing targets with them—have an initial edge in H then the new ownership paths cannot introduce new UH- nor UO-conflicts: Induction hypothesis $\mathfrak{g}_{i-1} \models \mathsf{UH}$ ensures that if one $h \in H$ is a `free` handle, then there is no other handle in H, and h's multiplicity is one. And induction hypothesis $\mathfrak{g}_{i-1} \models \mathsf{UO}$ ensures that paths $\pi' \in P$ have a unique source o'. Hence so have all initial handles $h \in H$, and thus all new and old ownership paths with the same target. There is no UH- and no UO-conflict.

  If $\hat{\mu}_i = \texttt{co}$, then consider that the `co`-call-link $\mathbf{s} \xrightarrow{\texttt{co}} \mathbf{r}$ has an inverse $\mathbf{r} \xrightarrow{\texttt{co}} \mathbf{s}$ in $\mathfrak{g}_{i-1}^\star$. Hence every new potential access path $\pi \in PAP_{\mathfrak{g}_i}(o', \mu', \omega')$ in $\mathfrak{g}_i$ has a precursor $\hat{\pi} \in PAP_{\mathfrak{g}_{i-1}}(o', \mu', \omega')$ in $\mathfrak{g}_{i-1}$ where for the new parameter link $\mathbf{r} \xrightarrow{\texttt{co}} \mathbf{o}_i$ one substitutes the pair $\mathbf{r} \xrightarrow{\texttt{co}} \mathbf{s} \xrightarrow{\texttt{co}} \mathbf{o}_i$ of the inverse call-link and the argument link, and for the new inverse parameter link $\mathbf{o}_i \xrightarrow{\texttt{co}} \mathbf{r}$ one substitutes the pair $\mathbf{o}_i \xrightarrow{\texttt{co}} \mathbf{s} \xrightarrow{\texttt{co}} \mathbf{r}$ of the inverse argument link and the call-link. The precursor $\hat{\pi}$ of a new `free` or `rep` path π moreover must have the same initial edge since the only new edges in $\mathfrak{g}_i^\star$ have mode `co`. Hence $\mathfrak{g}_i \models \mathsf{UH}$ and $\mathfrak{g}_i \models \mathsf{UO}$ follow directly from induction hypothesis $\mathfrak{g}_{i-1} \models \mathsf{UH}$ and $\mathfrak{g}_{i-1} \models \mathsf{UO}$. ■

5.5.2 Structural Integrity of Mutator Access

Theorem 3 If e_0, $\eta_0, \mathfrak{s}_0, om_0, \mathfrak{g}_0 \Longrightarrow^* e, \vec{\eta}, \mathfrak{s}, om, \mathfrak{g}$ is a reduction defined relative to a program p with $\vdash p$ **start** e_0 then

$$\mathfrak{g}, \vec{\eta} \models \mathsf{MCP}$$

Proof by induction on the number N of reduction steps from e_0 to e: In the base case $N = 0$, $\mathfrak{g}$ is the empty graph $\mathfrak{g}_0 = \emptyset$ which trivially satisfies MCP with any environment stack. In the induction step $N \to N+1$, execution e_0, $\eta_0, \mathfrak{s}_0, om_0, \mathfrak{g}_0 \Longrightarrow^* e_N, \vec{\eta}_N, \mathfrak{s}_N, om_N, \mathfrak{g}_N$ is continued $e_N, \vec{\eta}_N, \mathfrak{s}_N, om_N, \mathfrak{g}_N \Longrightarrow e, \vec{\eta}, \mathfrak{s}, om, \mathfrak{g}$. Let $\vec{\eta} = {\eta_1}_{h_1}^{\kappa_1} \bullet \ldots \bullet {\eta_n}_{h_n}^{\kappa_n}$ with $h_i = \langle \omega_{i-1}, \mu_i, \omega_i \rangle$.

The situation is simple for all those call-levels in $\vec{\eta}$ which existed already in $\vec{\eta}_N$. Let k be the depth of stack $\vec{\eta}_N$. For mutators at any level $i \leq k$ ($\kappa_i = \mathtt{mut}$), the induction hypothesis's $\mathfrak{g}_N, \vec{\eta}_N \models \mathsf{MCP}$ guarantees for some j a path $\pi = h_j, \ldots, h_i$ of call-links in $\vec{\eta}_N$ that form an ownership path $\omega_{j_i-1} \xrightarrow{\mu_{j_i}} \omega_{j_i}, \ldots, \omega_{i-1} \xrightarrow{\mu_i} \omega_i$. Since $\vec{\eta}$ still contains the call-links $h_j, \ldots, h_i$ of levels $i \leq k$ and below, these call-links still exist in $\mathfrak{g} = ogr(e, \vec{\eta}, \mathfrak{s})$ (Proposition 2) and still form the ownership path π.

Consequently, in all reduction steps where $n = k$ or $n = k-1$, the induction hypothesis guarantees $\mathfrak{g}, \vec{\eta} \models \mathsf{MCP}$. And case of {call}-steps with $n = k+1$, levels 1 to $n-1 = k$ are covered by the induction hypothesis. The new level n is a mutator, i.e., $\kappa_n = \mathtt{mut}$, if $\kappa^\star$ in the called method $F(f) = \kappa^\star\, t\, f(\ldots)\{\ldots\}$ for $\overline{om(\mathbf{r}) \doteq \langle \varrho_{\mathbf{r}}, F \rangle}$ is $\mathtt{mut}$. The term's typing $\Gamma, \kappa \vdash_X e : \tau$ (Theorem 1) with $\tilde{\mu}, \Gamma, \kappa, X = \overline{\mu_i, \Gamma_i, \kappa_i}$. implies a typing $\Gamma_n, \kappa_n \vdash_\epsilon \hat{e} : \hat{\tau}$ for the redex $\hat{e}$ in the context of the type assignment and method kind for the most deeply nested inlined method. The last element is $\Gamma_n, \kappa_n, \epsilon$ since $\vec{\eta}_N \models \tilde{\mu}, \Gamma, \kappa, X$ (Theorem 1). Typing the call expression $\hat{e}$ required a typing $\Gamma_n, \kappa_n \vdash_\epsilon \langle \mathbf{s}, \mu_{\mathbf{r}}, \mathbf{r} \rangle : \tilde{\mu}_{\mathbf{r}}\ c$ for the receiver expression. Hence $om(\mathbf{r}) \doteq \langle \varrho_{\mathbf{r}}, F \rangle$ with $\models om$ (Theorem 1) means by Lemma 2 that $FldsMths(c) = \langle \Gamma_c, F \rangle$ and $\tilde{\mu}_{\mathbf{r}} = \mu_{\mathbf{r}}$. But then $(\, f : (\ldots) \xrightarrow{\kappa^\star} \tau' \,) \in \Sigma(\tilde{\mu}_{\mathbf{r}}\ c)$ with the same kind $\kappa^\star$ as $F(f)$. Therefore typing $\hat{e}$ ensured in case of $\kappa^\star = \mathtt{mut}$ that $\mu_{\mathbf{r}} = \tilde{\mu}_{\mathbf{r}} \in \mathbf{Wr}(\kappa_n)$. This leaves two cases:

- If $\mu_{\mathbf{r}}$ is $\mathtt{free}$ or $\mathtt{rep}$, then the call-link $h = \mathbf{s} \xrightarrow{\mu_{\mathbf{r}}} \mathbf{r}$ in $\mathfrak{g}$ *is* the necessary ownership path for $\mathbf{r}$: $h \in PAP_{\mathfrak{g}}(\mathbf{s}, \mu_{\mathbf{r}}, \mathbf{r})$. ($h \in \mathfrak{g}$ follows with $\mathfrak{g} = ogr(e, \vec{\eta}, \mathfrak{s})$ from h as call-link in the new top-level environment in $\vec{\eta}$.)
- $\mu_{\mathbf{r}}$ can be $\mathtt{co}$ only if κ_n is $\mathtt{mut}$. But then induction hypothesis $\mathfrak{g}_N, \vec{\eta}_N \models \mathsf{MCP}$ ensures an ownership path $\pi = h_j \bullet \ldots \bullet h_n \in PAP_{\mathfrak{g}_N}(\omega_j, \mu', \mathbf{s})$. As explained above, π still exists in $\mathfrak{g}$ since it consists of call-links in $\vec{\eta}_N$. The $\mathtt{co}$ call-link in $\mathfrak{g}$ extends π to the necessary ownership path for $\mathbf{r}$: $\pi \bullet h \in PAP_{\mathfrak{g}}(\omega_j, \mu', \mathbf{r})$. ■

Theorem 4 If e_0, $\eta_0, \mathfrak{s}_0, om_0, \mathfrak{g}_0 \Longrightarrow^* e, \vec{\eta}, \mathfrak{s}, om, \mathfrak{g}$ is a reduction defined relative to a program p with $\vdash p$ **start** e_0 then

$$\mathfrak{g}, \vec{\eta} \models \mathsf{MC}$$

Proof by induction on the number N of reduction steps from e_0 to e: In the base case $N = 0$, $\mathfrak{g}$ is the empty graph $\mathfrak{g}_0 = \emptyset$, which trivially satisfies MC with any environment stack. In the induction step $N \to N+1$, execution $e_0, \eta_0, \mathfrak{s}_0, om_0, \mathfrak{g}_0 \Longrightarrow^* e_N, \vec{\eta}_N, \mathfrak{s}_N, om_N, \mathfrak{g}_N$ is continued $e_N, \vec{\eta}_N, \mathfrak{s}_N, om_N, \mathfrak{g}_N \Longrightarrow e, \vec{\eta}, \mathfrak{s}, om, \mathfrak{g}$. Let $\vec{\eta} = \eta_1{}^{\kappa_1}_{h_1} \bullet \ldots \bullet \eta_n{}^{\kappa_n}_{h_n}$ with $h_i = \langle \omega_{i-1}, \mu_i, \omega_i \rangle$.

Proceed by *reductio ad absurdum*. Assume a violation of MC by $\mathfrak{g}$ and $\vec{\eta}$. Then there must be a call-level i in $\vec{\eta}$ at which the receiver ω_i is executing a mutator ($\kappa_i = \mathtt{mut}$), and there is a "non-controlling" representative o_1 which has ω_i in its sanctuary $Sanc_{\mathfrak{g}}(o_1)$ but does not execute a mutator at a lower call-level ($o_1 \notin \{\omega_1, \ldots, \omega_i\}$). The inductive definition of $\omega_i \in Sanc_{\mathfrak{g}}(o_1)$ based on $\mathtt{rep}$ paths obviously requires a non-empty sequence $\pi_1 \bullet \ldots \bullet \pi_k$ of $\mathtt{rep}$ paths $\pi_j \in PAP_{\mathfrak{g}}(o_j, \mathtt{rep}, o_{j+1})$ connecting o_1 with $\omega_i = o_{k+1}$ via objects $o_2, \ldots, o_k$: $o_1 \xrightarrow{\mathtt{rep}} o_2 \xrightarrow{\mathtt{rep}} \ldots \xrightarrow{\mathtt{rep}} o_{k+1} = \omega$.

Show by induction on the length k of the shortest connecting $\mathtt{rep}$ path sequence that the representative is executing a mutator at a level $j \leq i$. In the base case $k = 0$, $j = i$ and $\omega_i = o_1$. But ω_i cannot be the *non-controlling* representative o_1 since $\kappa_i = \mathtt{mut}$.

In the induction step $k-1 \to k$, sequence $o_1 \xrightarrow{\mathtt{rep}} o_2 \ldots o_{k-1} \xrightarrow{\mathtt{rep}} o_k$ is extended by $\pi_k = o_k \xrightarrow{\mathtt{rep}} o_{k+1} \in PAP_{\mathfrak{g}}(o_k, \mathtt{rep}, \omega_i)$

1. $\mathfrak{g}_N, \vec{\eta}_N \models \mathsf{MCP}$ (Theorem 3) guarantees for $\kappa_i = \mathtt{mut}$ some j such that $\pi = h_j \bullet \ldots \bullet h_i$ is an ownership path from ω_{j-1} to ω_i.
2. $\pi_k \in PAP_{\mathfrak{g}}(o_k, \mathtt{rep}, \omega_i)$ means by $\mathfrak{g} \models \mathsf{UO}$ (Theorem 3) that *all* ownership paths to ω_i start with o_k. Hence $\omega_{j-1} = o_k$.
3. By $\mathfrak{g} \models \mathsf{UH}$ (Theorem 3), $\mathtt{rep}$ path π_k to ω_i guarantees that there is no $\mathtt{free}$ path to ω_i. Consequently, $\pi \in PAP_{\mathfrak{g}}(o_k, \mathtt{rep}, \omega_i)$ and $\omega_i \in Sanc_{\mathfrak{g}}(o_k)$.
4. If, on one hand, call-level i existed already in $\vec{\eta}_N$, i.e., $i \leq n$, then $\mathtt{rep}$ path $\pi = h_j \bullet \ldots \bullet h_i$ already existed in $\vec{\eta}_N$, and thus in $\mathfrak{g}_N$ by virtue of $\mathfrak{g}_N = ogr(e_N, \vec{\eta}_N, \mathfrak{s}_N)$ (Proposition 2). But then $\omega_i \in Sanc_{\mathfrak{g}_N}(o_k)$, so that the outer induction hypothesis $\mathfrak{g}_N, \vec{\eta}_N \models \mathsf{MC}$ guarantees a mutator-execution by o_k at a call-level $j \leq i$ in $\vec{\eta}_N$, and thus in $\vec{\eta}$.
5. If, on the other hand, call-level i is new in $\vec{\eta}$, then it must be the new top-level $i = n = n_N + 1$ in a $\{$call$\}$-step: $\vec{\eta}$ is $\vec{\eta}_N \bullet \eta^{\star}{}^{\kappa^\star}_{\langle \mathbf{s}, \mu_{\mathbf{r}}, \mathbf{r} \rangle}$ for redex $\hat{e} = \langle \mathbf{s}, \mu_{\mathbf{r}}, \mathbf{r} \rangle \Leftarrow f(\ldots)$, with $\kappa_i = \kappa_n = \kappa^\star$ and $\omega_i = \omega_n = \mathbf{r}$ and $\omega_{i-1} = \omega_n = \mathbf{s}$. As elaborated in the proof for Theorem 3, the typing $\Gamma, \kappa \vdash_X e : \tau$ of term e with redex $\hat{e}$ required that $\mu_{\mathbf{r}} \in \mathbf{Wr}(\kappa_n)$ if $\kappa_i = \kappa_n = \kappa^\star$ is $\mathtt{mut}$. $\mu_{\mathbf{r}}$ cannot be $\mathtt{free}$, since call-link $\langle \mathbf{s}, \mu_{\mathbf{r}}, \mathbf{r} \rangle$ in $\vec{\eta}$ would by $\mathfrak{g} = ogr(e, \vec{\eta}, \mathfrak{s})$ (Proposition 2) mean a $\mathtt{free}$ path $\mathbf{s} \xrightarrow{\mu_{\mathbf{r}}} \mathbf{r} \in PAP_{\mathfrak{g}}(\mathbf{s}, \mathtt{free}, \omega_i)$ to ω_i, in contradiction to step 3. If $\kappa_n = \mathtt{mut}$ and $\mu_{\mathbf{r}} = \mathtt{rep}$, then the call-link $\mathbf{s} \xrightarrow{\mathtt{rep}} \mathbf{r}$ in $\mathfrak{g}$ *is* the path π_k, i.e., $o_k = \mathbf{s} \xrightarrow{\mathtt{rep}} \mathbf{r} = \omega_i$. This means that $o_k = \mathbf{s} = \omega_n$ is executing a mutator at level $j = n \leq n$. And if $\kappa_n = \mathtt{mut}$ and $\mu_{\mathbf{r}} = \mathtt{co}$, then π_k is extended by the call-link's inverse $\mathbf{r} \xrightarrow{\mathtt{co}} \mathbf{s}$ to a $\mathtt{rep}$ path from o_k to $\mathbf{s}$ in $\mathfrak{g}^\star$, so that $\mathbf{s} \in Sanc_{\mathfrak{g}}(o_k)$. But then, as shown in the previous step, $\mathbf{s} = \omega_n$ with $\kappa_n = \mathtt{mut}$ means a mutator-execution by o_k at a level $j \leq n$.

6. Either way, o_k is executing a mutator at some call-level $j \leq i \leq n$ in $\vec{\eta}$. Since $o_k \in Sanc_{\mathfrak{g}}(o_1)$ through **rep** paths $\pi_1 \bullet \ldots \bullet \pi_{k-1}$, the induction hypothesis therefore guarantees that o_1 is executing a mutator at some call-level $j' \leq j \leq n$. This violates the assumption of o_1 as a "non-controlling" representative. ■

5.5.3 Composite State Encapsulation

Theorem 5 If $e_0, \eta_0, \mathfrak{s}_0, om_0, \mathfrak{g}_0 \Longrightarrow^* e, \vec{\eta}, \mathfrak{s}, om, \mathfrak{g} \Longrightarrow e', \vec{\eta}', \mathfrak{s}', om', \mathfrak{g}'$ is a reduction defined relative to a program p with $\vdash p$ **start** e_0 then for all $o \in \mathrm{dom}(om)$,

$$CState_{\mathfrak{s},om}(o) \neq CState_{\mathfrak{s}',om'}(o) \;\Rightarrow\; \exists i \leq n \bullet\ \mathbf{r}_i = o \;\wedge\; \kappa_i = \mathtt{mut}$$

where $\vec{\eta} = \eta_1{}^{\kappa_1}_{h_1}, \ldots, \eta_n{}^{\kappa_n}_{h_n}$ with $h_i = \langle \mathbf{s}_i, \mu_i, \mathbf{r}_i \rangle$.

Proof: The proof goes straight-forward with the lemmas on coherence and shallow state encapsulation developed below (Lemmas 3 and 4).

$$\begin{array}{cll}
 & CState_{\mathfrak{s},om}(o) \neq CState_{\mathfrak{s}',om'}(o) & \\
\overset{\text{Lemma 3}}{\Longrightarrow} & \exists \omega \in StRep_{\mathfrak{s},om}(o) \bullet\ \mathfrak{s}\,|_{flds_{om}(\omega)} \neq \mathfrak{s}'\,|_{flds_{om}(\omega)} & \\
\overset{\text{Lemma 4}}{\Longrightarrow} & \exists \omega \in StRep_{\mathfrak{s},om}(o) \bullet\ \mathbf{r}_n = \omega & \wedge\ \kappa_n = \mathtt{mut} \\
\Rightarrow & \mathbf{r}_n \in StRep_{\mathfrak{s},om}(o) & \wedge\ \kappa_n = \mathtt{mut} \\
\overset{\text{Proposition 4}}{\Longrightarrow} & \mathbf{r}_n \in \{o\} \cup Sanc_{ogr(e,\vec{\eta},\mathfrak{s})}(o) & \wedge\ \kappa_n = \mathtt{mut} \\
\overset{\text{Proposition 2}}{\Longrightarrow} & \mathbf{r}_n \in \{o\} \cup Sanc_{\mathfrak{g}}(o) & \wedge\ \kappa_n = \mathtt{mut} \\
\Rightarrow & (\mathbf{r}_n = o \wedge \kappa_n = \mathtt{mut}) \;\vee\; (\mathbf{r}_n \in Sanc_{\mathfrak{g}}(o) \;\wedge\; \kappa_n = \mathtt{mut}) & \\
\overset{\text{Theorem 4}}{\Longrightarrow} & (\mathbf{r}_n = o \wedge \kappa_n = \mathtt{mut}) \;\vee\; (\exists i \leq n \bullet\ \mathbf{r}_i = o \;\wedge\; \kappa_i = \mathtt{mut}) & \\
\Rightarrow & \exists i \leq n \bullet\ \mathbf{r}_i = o \;\wedge\; \kappa_i = \mathtt{mut} &
\end{array}$$

■

One naturally expects that all changes of a composite object's state *CState*(o) are represented by updates of fields of some implementation objects in its state representation *StRep*(o). We call this property "***coherence***" (of composite objects, of composite state, or of state representations, as you please).

Lemma 3 If $e_0, \eta_0, \mathfrak{s}_0, om_0, \mathfrak{g}_0 \Longrightarrow^* e, \vec{\eta}, \mathfrak{s}, om, \mathfrak{g} \Longrightarrow e', \vec{\eta}', \mathfrak{s}', om', \mathfrak{g}'$ is a reduction defined relative to a program p with $\vdash p$ **start** e_0 then

$$CState_{\mathfrak{s},om}(o) \neq CState_{\mathfrak{s}',om'}(o) \;\Rightarrow\; \exists \omega \in StRep_{\mathfrak{s},om}(o) \bullet\ \mathfrak{s}\,|_{flds_{om}(\omega)} \neq \mathfrak{s}'\,|_{flds_{om}(\omega)}$$

Proof: A change of the composite state $CState_{\mathfrak{s},om}(o)$ means, if we expand it by Definition 5, a change of a restriction of the store, namely

$$\mathfrak{s}\,|_{\bigcup_{\omega \in StRep_{\mathfrak{s},om}(o)} flds_{om}(\omega)} \;\neq\; \mathfrak{s}'\,|_{\bigcup_{\omega \in StRep_{\mathfrak{s}',om'}(o)} flds_{om'}(\omega)}$$

In the simple case, the domain of the restriction is unchanged: $L = \bigcup_{\omega \in StRep_{\mathfrak{s},om}(o)} flds_{om}(\omega) = \bigcup_{\omega \in StRep_{\mathfrak{s}',om'}(o)} flds_{om'}(\omega) = L'$. Then the composite state change $\mathfrak{s}\,|_L \neq \mathfrak{s}'\,|_{L'}$ means that the store changed at some location $\ell \in L = L'$: $\mathfrak{s}(\ell) \neq \mathfrak{s}'(\ell)$. It must be the field location $\ell \in flds_{om}(\omega)$ of some object $\omega \in StRep_{\mathfrak{s},om}(o)$. Since $\mathfrak{s}(\ell) \neq \mathfrak{s}'(\ell)$, $\mathfrak{s}\,|_{flds_{om}(\omega)} \neq \mathfrak{s}'\,|_{flds_{om}(\omega)}$ for this ω.

Next consider a change in the set of store locations representing the composite state: $L = \bigcup_{\omega \in StRep_{\mathfrak{s},om}(o)} flds_{om}(\omega) \neq \bigcup_{\omega \in StRep_{\mathfrak{s}',om'}(o)} flds_{om'}(\omega) = L'$. It is obvious from the reduction rules that object-map om changes only by extension, and for fresh object identifiers. For the "old" objects $\omega \in StRep_{\mathfrak{s},om}(o)$, the set of field locations is unchanged: $flds_{om}(\omega) = flds_{om'}(\omega)$. Hence the change from L to L' presupposes a change of the set of state-representing implementation objects: $StRep_{\mathfrak{s},om}(o) \neq StRep_{\mathfrak{s}',om'}(o)$. Expanded with Definition 4, this means

$$\{o\} \cup \bigcup_{PAP_{fgr_{om}(\mathfrak{s})}(o,\mathtt{rep},\omega) \neq \emptyset} StRep_{\mathfrak{s},om}(\omega) \;\neq\; \{o\} \cup \bigcup_{PAP_{fgr_{om'}(\mathfrak{s}')}(o,\mathtt{rep},\omega) \neq \emptyset} StRep_{\mathfrak{s}',om'}(\omega)$$

That is, there must be an object q that is reachable from o by a non-empty sequence $o = o_0 \overset{\mathtt{rep}}{\dashrightarrow} o_1 \overset{\mathtt{rep}}{\dashrightarrow} \ldots \overset{\mathtt{rep}}{\dashrightarrow} o_n = q$ of `rep` paths in field subgraph $fgr_{om}(\mathfrak{s})$ but no such sequence in $fgr_{om'}(\mathfrak{s}')$, or vice versa. Each of the `rep` paths $o_i \overset{\mathtt{rep}}{\dashrightarrow} o_{i+1}$ is in base-JaM a `rep` edge followed by `co` edges: $o_i = o_{i,0} \xrightarrow{\mathtt{rep}} o_{i,1} \xrightarrow{\mathtt{co}} o_{i,2} \ldots o_{i,k_i-1} \xrightarrow{\mathtt{co}} o_{i,k_i} = o_{i+1}$. In order for the path sequence to exist in $fgr_{om}(\mathfrak{s})$ but not in $fgr_{om'}(\mathfrak{s}')$, or vice versa, there must be a left-most `rep` or `co` edge $o_{i,j} \xrightarrow{\mu_{i,j}} o_{i,j+1}$ or $o_{i,j} \xrightarrow{\mu_{i,j}} o_{i+1,0}$ that appears in, or disappears from, the field subgraph. That is, a handle $\langle o_{i,j}, \mu, o_{i,j+1} \rangle$ or $\langle o_{i,j}, \mu, o_{i+1,0} \rangle$ is captured in, or removed from, a field location ℓ. By source consistency $\models_{\mathfrak{s}} om$ (Proposition 1), this field must belong to the handle's source $o_{i,j}$: $\ell \in flds_{om}(o_{i,j})$. Since non-empty prefixes of `rep` paths are also `rep` paths, there is an unchanged `rep` path sequence from o up to $o_{i,j}$. This means that $o_{i,j}$ is in o's state representation before and after the change: $o_{i,j} \in StRep_{\mathfrak{s},om}(o) \cap StRep_{\mathfrak{s}',om'}(o)$. Since $o_{i,j}$ has a changed field, $\mathfrak{s}\,|_{flds_{om}(o_{i,j})} \neq \mathfrak{s}'\,|_{flds_{om}(o_{i,j})}$, it is the desired object ω. ■

Shallow state encapsulation means that o's fields change only by assignments and destructive reads executed by o itself. This may seem obvious from the typing rules, but proving it is surprisingly tedious since two obvious invariants about variables have to be verified.

Lemma 4 If $e_0, \eta_0, \mathfrak{s}_0, om_0, \mathfrak{g}_0 \Longrightarrow^* e, \vec{\eta}, \mathfrak{s}, om, \mathfrak{g} \Longrightarrow e', \vec{\eta}', \mathfrak{s}', om', \mathfrak{g}'$ is a reduction defined relative to a program p with $\vdash p$ **start** e_0 then for $\vec{\eta} = {\eta_1}_{h_1}^{\kappa_1}, \ldots, {\eta_n}_{h_n}^{\kappa_n}$ with $h_n = \langle \mathbf{s}_n, \mu_n, \mathbf{r}_n \rangle$, and for all $\omega \in \mathrm{dom}(om)$,

$$\mathfrak{s}\,|_{flds_{om}(\omega)} \neq \mathfrak{s}'\,|_{flds_{om}(\omega)} \;\Rightarrow\; \mathbf{r}_n = \omega \;\wedge\; \kappa_n = \mathtt{mut}$$

Proof: The proof is based on two invariants holding in configuration $e, \vec{\eta}, \mathfrak{s}, om, \mathfrak{g}$. Let $\vec{\eta} = {\eta_1}_{h_1}^{\kappa_1}, \ldots, {\eta_n}_{h_n}^{\kappa_n}$ with $h_i = \langle \mathbf{s}_i, \mu_i, \mathbf{r}_i \rangle$.

I1 *Fields are not aliased by local identifiers*: $\mathit{locals}(\vec{\eta}) \cap \mathit{flds}(\mathit{om}) = \emptyset$
where $\mathit{locals}(\vec{\eta})$ is the set $\bigcup_{\eta_h^\kappa \in \vec{\eta}} \mathrm{im}(\eta)$ of locations of all local variables.

I2 *Field locations of object o occur at "mutable positions" only in mutators of o*:
$\mathit{mutlocs}(\mathcal{E}_i) \cap \mathit{flds}_{\mathit{om}}(o) \neq \emptyset \;\Rightarrow\; \mathbf{r}_i = o \wedge \kappa_i = \mathtt{mut}$
where $\mathit{mutlocs}(\mathcal{E}_i)$ are is the set of locations that are left-hand sides of assignments and the l-values in destructive read accesses in the term context $\mathcal{E}_i$ at nesting level i. A precise definition will be given below.

If the environment stack $\vec{\eta}$ has height n, a runtime term e for which e, $\vec{\eta}, \mathfrak{s}, \mathit{om}, \mathfrak{g}$ $\Longrightarrow$ e', $\vec{\eta}', \mathfrak{s}', \mathit{om}', \mathfrak{g}'$ is defined must contain n nesting levels of inlined methods. Hence it can be decomposed by a series of reduction context $\mathcal{E}_1, \ldots, \mathcal{E}_{n-1} \in R_1^\Box$ and an innermost runtime term e_n containing no inlined method body such that $e = \mathcal{E}_1[\ll\mathcal{E}_2[\ll\ldots[\ll\mathcal{E}_{n-1}[\ll e_n\gg]\gg]\ldots\gg]\gg]$. For uniformity, let us write $\mathcal{E}_n$ for e_n.

The set $\mathit{mutlocs}(e)$ of locations identifying the updated variables in assignments or the destructively read variables in read accesses in a runtime term *or* reduction context e is determined inductively as follows.

$\mathit{mutlocs}(x)$	$=_{\mathrm{df}} \emptyset$	$\mathit{mutlocs}(e\,\texttt{=}\,\tilde{e}\texttt{;})$	$=_{\mathrm{df}} \{e \in \mathcal{L}oc\} \cup \mathit{mutlocs}(\tilde{e})$
$\mathit{mutlocs}(\texttt{this}.x)$	$=_{\mathrm{df}} \emptyset$	$\mathit{mutlocs}(\texttt{destval}(e))$	$=_{\mathrm{df}} \{e \in \mathcal{L}oc\}$
$\mathit{mutlocs}(\ell)$	$=_{\mathrm{df}} \emptyset$	$\mathit{mutlocs}(\texttt{val}(e))$	$=_{\mathrm{df}} \emptyset$
$\mathit{mutlocs}(h)$	$=_{\mathrm{df}} \emptyset$	$\mathit{mutlocs}(\ll s\gg)$	$=_{\mathrm{df}} \mathit{mutlocs}(s)$
$\mathit{mutlocs}(\texttt{null})$	$=_{\mathrm{df}} \emptyset$	$\mathit{mutlocs}(\texttt{return}\ e\texttt{;})$	$=_{\mathrm{df}} \mathit{mutlocs}(e)$
$\mathit{mutlocs}(\texttt{new}\ c\texttt{()})$	$=_{\mathrm{df}} \emptyset$	$\mathit{mutlocs}(s_1\ s_2)$	$=_{\mathrm{df}} \mathit{mutlocs}(s_1)$
$\mathit{mutlocs}(\Box)$	$=_{\mathrm{df}} \emptyset$	$\mathit{mutlocs}(\texttt{if}\,(e_1\psi e_2)\,\{s\})$	$=_{\mathrm{df}} \mathit{mutlocs}(e_1) \cup \mathit{mutlocs}(e_2)$
$\mathit{mutlocs}(\texttt{while}(e)\{s\})$	$=_{\mathrm{df}} \emptyset$	$\mathit{mutlocs}(e_0 \Leftarrow f\,(e_1, \ldots, e_n))$	$=_{\mathrm{df}} \bigcup_{i=0}^n \mathit{mutlocs}(e_i)$

We are able to ignore the subterms of `while` statements, the then-branch of `if` statements, and the second statement in a sequence since these are never partially evaluated, and thus always free of locations. The cases of runtime terms $e\,\texttt{=}\,\tilde{e}$, $\texttt{val}(e)$, and $\texttt{destval}(e)$ are simplified based on the assumption that their subterm e can never contain destructive reads nor assignments.

Next, show by induction on the number N of reduction steps from e_0 to e that the auxiliary invariants I1 and I2 hold in e, $\vec{\eta}, \mathfrak{s}, \mathit{om}, \mathfrak{g}$. In the base case $N = 0$, there can be no $\ell \in \mathit{flds}(\mathit{om})$ and no $o \in \mathrm{dom}(\mathit{om})$ since $\mathit{om} = \mathit{om}_0 = \emptyset$. Hence invariants I1 and I2 are trivial. In the induction step $N \to N+1$, execution e_0, $\eta_0, \mathfrak{s}_0, \mathit{om}_0, \mathfrak{g}_0$ $\Longrightarrow^*$ e_N, $\vec{\eta}_N, \mathfrak{s}_N, \mathit{om}_N, \mathfrak{g}_N$ is continued e_N, $\vec{\eta}_N, \mathfrak{s}_N, \mathit{om}_N, \mathfrak{g}_N \Longrightarrow e$, $\vec{\eta}, \mathfrak{s}, \mathit{om}, \mathfrak{g}$.

Ad I1. Since $\mathit{locals}(\vec{\eta}_N) \cap \mathit{flds}(\mathit{om}_N) = \emptyset$ by induction hypothesis, condition $\mathit{locals}(\vec{\eta}) \cap \mathit{flds}(\mathit{om}) = \emptyset$ could only be violated by the addition of locations to $\vec{\eta}_N$, i.e., in a {call}-step, or to om_N, i.e., in a {new}-step. In both cases, the added locations are fresh—so that they do not overlap with old locations—and they are added to only one of $\vec{\eta}_N$ or om_N. Hence $\mathit{locals}(\vec{\eta}) \cap \mathit{flds}(\mathit{om}) = \emptyset$.

Ad I2. The induction hypothesis means that
$e_N = \mathcal{E}_1[\ll\mathcal{E}_2[\ll\ldots[\ll\mathcal{E}_{n_N-1}[\ll\mathcal{E}_{n_N}\gg]\gg]\ldots\gg]\gg]$
with $\mathit{mutlocs}(\mathcal{E}_i) \cap \mathit{flds}_{\mathit{om}_N}(o) \neq \emptyset \;\Rightarrow\; \mathbf{r}_i = o \wedge \kappa_i = \mathtt{mut}$ for $o \in \mathrm{dom}(\mathit{om}_N)$. The

invariant is unaffected by all steps in which there are no new locations in the term e, and in which om and $\vec{\eta}$ are unchanged. In case of {new}, there is a new object in dom(om), but the locations of its fields are fresh, and thus cannot occur in e. In case of {call}, nothing is removed from $\vec{\eta}_N$. It is only extended to height $n = n_N + 1$. The term at the new nesting level in e is the method body s. Since it originated from the program p, it cannot contain any locations. In case of {ret}, top-level receiver $\mathbf{r}_n$ and method kind κ_n are removed from the stack. But they are not needed any more since the method nesting depth of e is $n = n_N - 1$, i.e., one less than that of e_N. In case of {var$_l$}, the only change is that e contains the additional location $\eta_n(x)$. By induction hypothesis I1, $\mathit{locals}(\vec{\eta}_N) \cap \mathit{flds}(om_N) = \emptyset$, so that this location cannot be the location of any object's field.

The really interesting case is {var$_f$}. Here the redex $\hat{e} = \mathtt{this}.x$ in e_N reduces to the location $\ell \in \mathit{flds}_{om_N}(o)$ of o's x-field, where o is the target of the top-level this-handle: $\mathfrak{s}(\eta_n(\mathtt{this})) = \langle o, \mu, o\rangle$. By source consistency $\models_{\mathfrak{s}} \vec{\eta}$ (Proposition 1), the source o of a handle at a location in top-level environment $\eta_{n}{}_{h_n}^{\kappa_n}$ implies that o is the top-level receiver $\mathbf{r}_n$. If the redex $\hat{e}$ in e_N is a right-hand side or destructively read expression, i.e., $\mathcal{E}_n = \mathcal{E}[\mathtt{destval}(\hat{e})]$ or $\mathcal{E}_n = \mathcal{E}[\hat{e} = e_2]$, then $\mathcal{E}'_n = \mathcal{E}[\mathtt{destval}(\ell)]$ or $\mathcal{E}'_n = \mathcal{E}[\ell = e_2]$, respectively means there is a new field location ℓ in $\mathit{mutlocs}(\mathcal{E}'_n)$. The typeability of e_N guaranteed by Theorem 1 required the typeability, in particular, of redex $\mathtt{destval}(\hat{e})$ or $\hat{e} = e_2$, respectively in the context of Γ_n, κ_n. But then $\hat{e} = \mathtt{this}.x$ implied $\kappa_n = \mathtt{mut}$. Since $\mathbf{r}_n = \omega$ and $\kappa_n = \mathtt{mut}$, and since the other levels of term and environment stack are unchanged, invariant I2 is preserved.

With invariants I1 and I2 holding in $e, \vec{\eta}, \mathfrak{s}, om, \mathfrak{g}$, it is now easy to show shallow state encapsulation: Consider the cases of step $e, \vec{\eta}, \mathfrak{s}, om, \mathfrak{g} \implies e', \vec{\eta}', \mathfrak{s}', om', \mathfrak{g}'$ which reduces redex $\hat{e}$ in $e = \mathcal{E}[\hat{e}]$ to $\hat{e}'$ in $e' = \mathcal{E}[\hat{e}']$. Shallow state encapsulation is trivial for all reductions which change neither $\mathfrak{s}$ nor om. In case of {call} and {new}, om is unchanged or changes only for a *fresh* object identifier, and $\mathfrak{s}$ changes only at locations that are *fresh*. Thus neither do old objects get new fields, nor does the value of their old fields change, so that shallow state encapsulation is trivial. In case of {ret}, the store is reset at the locations in top-level environment $\eta_n \in \vec{\eta}$. This changes no object's fields since $\mathit{locals}(\vec{\eta}) \cap \mathit{flds}(om) = \emptyset$ by I1. Hence shallow state encapsulation is trivial.

The two steps where $\mathfrak{s}\,|_{\mathit{flds}_{om}(\omega)} \neq \mathfrak{s}'\,|_{\mathit{flds}_{om'}(\omega)}$ are {rd$_{dst}$}-steps with $\hat{e} = \mathtt{destval}(\ell)$ and {upd}-steps with $\hat{e} = \ell = e_2$ such that $\ell \in \mathit{flds}_{om}(\omega) = \mathit{flds}_{om'}(\omega)$ for some ω. In this case, the redex's reduction is $\hat{e}, \vec{\hat{\eta}}, \mathfrak{s}, om, \mathfrak{g} \longrightarrow \hat{e}', \vec{\hat{\eta}}', \mathfrak{s}', om', \mathfrak{g}'$ with $\hat{e}$ at nesting level n in e and with $\vec{\hat{\eta}} = \eta_n$ and $\vec{\hat{\eta}}' = \eta'_n$. The I2 guarantees the desired $\mathbf{r}_n = \omega \wedge \kappa_n = \mathtt{mut}$. ∎

Chapter 6

JaM with the Full Mode System

There is a theory which states that if ever anybody discovers
exactly what the Universe is for and why it is here,
it will instantly disappear and be replaced
by something even more bizarre and inexplicable.
Douglas Adams (1952-2001)

This section extends base-JaM's reduced system of modes to the full system presented in chapter 1: Association modes $\alpha \in \mathbb{A}$ are added to the set of modes, and these base-modes are parameterized by correlations δ to specify the mode of μ-paths' extensions by association paths. A full mode now has the form $m{<}\delta{>}$.

While the formal treatment of many JaM properties is a simple forward adaption from the previous chapter, the proofs of the unique owner and unique head invariants, and of coherence will have to be redone completely: Potential access paths in JaM have a much more complicated structure than in base-JaM through the possibility of extending μ-paths to non-μ paths. This lets the complexity of reasoning explode. In order to make the formal treatment feasible within the space of a dissertation, some simplifications are made:

- We will consider neither the extensions $o \overset{\mathtt{co}}{\dashrightarrow} q \overset{\alpha}{\dashrightarrow} \omega$ and $o \overset{\alpha'}{\dashrightarrow} q \overset{\alpha}{\dashrightarrow} \omega$ *of* `co`- and association paths by association paths, nor the extension $o \overset{\mu}{\dashrightarrow} q \overset{\alpha}{\dashrightarrow} \omega$ of potential access paths by association paths *to* `co`- and `free` paths $o \overset{\mathtt{co}}{\dashrightarrow} \omega$ and $o \overset{\mathtt{free}}{\dashrightarrow} \omega$. This simplification is reflected in constraints on the nesting structure of mode-terms: Only modes `free`, `rep` and `read` are parameterized by correlations ($\mathtt{free}{<}\delta{>}$, $\mathtt{rep}{<}\delta{>}$ and $\mathtt{read}{<}\delta{>}$, but $\mathtt{co}{<}{>}$ and $\beta{<}{>}$), and only by correlations to `rep`, `read` and association modes ($m{<}\ldots,\alpha{=}\mathtt{rep}{<}\delta'{>},\ldots{>}$, $m{<}\ldots,\alpha{=}\mathtt{read}{<}\delta'{>},\ldots{>}$ and $m{<}\ldots,\alpha{=}\gamma{<}{>},\ldots{>}$).
- Implicit mode-conversions from $\mathtt{free}{<}{>}$ to $\mathtt{co}{<}{>}$ or $\alpha{<}{>}$ caused by assignment or parameter supply will not be considered. (Tedious invariants about all sequences $\omega \overset{\vec{\alpha}}{\dashrightarrow} q$ of association paths starting from targets of $\mathtt{free}{<}{>}$ paths $o \overset{\mathtt{free}{<}{>}}{\dashrightarrow} \omega$ would be needed in order to show that such conversions preserve the uniqueness of ownership.) This simplification will be reflected in the definition of the mode compatibility relation $\leq_{\mathrm{m}}$.

- `null` and `new` expressions are annotated with a correlation-set δ to make explicit which correlations should parameterize their value.
- Base-JaM's simplifications of the Java subset for the formal treatment remain: Explicit read access, no values other than object references, no subclassing, etc.

After the definition of JaM as an extension of base-JaM in the first half of this chapter, the second half will prove the higher-level properties of the execution states and steps possible in JaM, namely the structural integrity of ownership and mutator access, and state encapsulation.

6.1 Introducing the New Modes

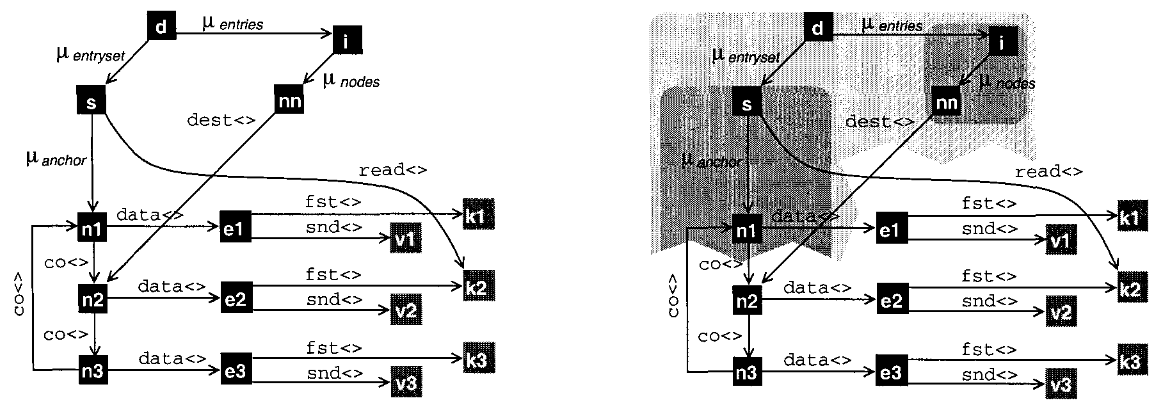
1. FULL MODES. In JaM as in base-JaM, class names c are annotated with modes μ in Java's declaration of object reference variables in order to classify the contained object references and their concatenations to potential access paths. JaM extends and refines base-JaM's set of modes, the ***base-modes***: The set of modes is extended by a set $\mathbb{A}$ of identifiers α called ***association roles*** to the set $\mathcal{B} = \{\texttt{free}, \texttt{rep}, \texttt{co}, \texttt{read}\} \cup \mathbb{A}$ of base-modes. Each base-mode $m \in \mathcal{B}$ in then parameterized by the annotation of a (possible) empty set δ of ***correlations*** α_i=μ_i to a full mode $\mu = m$<α_1=$\mu_1, \ldots, \alpha_n$=μ_n>.

A correlation α=μ' on a reference or path $\pi = q \overset{m<\ldots,\alpha=\mu',\ldots>}{\dashrightarrow} o$ correlates o's α-paths with q's μ'-paths, so that consequently π is extended by o's association references and paths $o \overset{\alpha<>}{\dashrightarrow} \omega$ to paths $q \overset{\mu'}{\dashrightarrow} \omega$. By itself, i.e., from the perspective of source object o, classifying a reference or path $o \dashrightarrow \omega$ by an association role $\alpha \in \mathbb{A}$ says nothing about target object ω. It does not fix ω's owners or sanctuary memberships. It only allows *third objects* q with a path $q \overset{m<\ldots,\alpha=\mu',\ldots>}{\dashrightarrow} o$ to o to fix ω's owner and sanctuary membership by correlating α with an appropriate mode.

The meaning of old base-modes $m \in \{\texttt{free}, \texttt{rep}, \texttt{co}, \texttt{read}\}$ for the classification of references $h = o \xrightarrow{m} \omega$ and paths $\pi = o \overset{m}{\dashrightarrow} \omega$ remains unchanged:

- Base-mode $m = \texttt{rep}$ means that o is ω's owner, which is expected to be unique, and ω belongs to o's sanctuary *Sanc*(o). If π is represented in fields, then ω is a state-representing component of o in *StRep*(o).
- Base-mode $m = \texttt{free}$ means that o is ω's owner, which is expected to be unique, and ω is expected not to belong to any sanctuary (and to no composite state representation except *StRep*(ω)).
- Base-mode $m = \texttt{co}$ means that ω and o have the same owners, and belong to the same sanctuaries *Sanc*(q). o and ω are called *co-objects.*
- Base-mode $m = \texttt{read}$ means that nothing is said about ω's owners and sanctuary memberships.

2. EXAMPLE: MAPS. Let `d` be an instance of the `MapImp` class with three entries represented as `Pair` components `e1`, `e2` and `e3`, and a entry-set component `s` of class



where

$\mu_{anchor} \equiv \texttt{rep<data=elem<>>}$

$\mu_{entryset} \equiv \texttt{rep<elem=rep<fst=key<>, snd=value<>>>}$

$\mu_{nodes} \equiv \texttt{rep<dest=read<data=dest<>>>}$

$\mu_{entries} \equiv \texttt{free<dest=rep<fst=key<>, snd=value<>>>}$

Figure 6.1: Moded object graph during lookup

`PSetImp`. Composite object `s` is a composite object like C_2 in chapter 1 that reifies not a set S_1 of pairs but a set S_2 of `Pair` *objects*. Consider the situation while `d` is in the middle of an iteration over its entries during a lookup for `k2`. Figure 6.1 shows this situation as an object graph. The edges are labeled with modes in a way that expresses the object composition and state representation relationships:

`Rep` and `free` modes suffice to express object composition relationships represented by directed object references (see the right hand side of fig. 6.1): Node `n1` is a state-representing component of set `s`, which is a state-representing component of map `d`; and iterator `nn` is a state-representing component of iterator `i`, which is a behavioral component of map `d`. That is, the abstract state of the map object is, in part, represented in the objects `d`, `s`, and `n1`. And the state of the map's iteration over the set is represented in `i` and `nn`. However, `rep` and `free` are unable to properly capture that the other two nodes also belong to `s`'s state representation, and that the `Pair` objects `e1`, `e2`, and `e3` belong to `d`'s state representation. Declaring the `next`-links of `PNode` objects to be `rep` would turn single-linked node `n1` into a composite object, that subsumes the next object `n2` as a part of its state, contrary to the meaning of "node." Since the nodes are linked to a ring structure, `next`-links of mode `rep` would mean *cyclic* composition, something which is non-sensical for any aggregation or part-whole relation [OMG00, Sim87, Var96].

The `next`-links between `PNode` objects link the node objects to an *object structure*, and an object structure is not expected to be shared among composite objects. Either it completely belongs to a composite object, or not. It cannot be that one composite uses the first half of a `PNode`-list (or the odd `PNodes`), while another composite uses the second half (or the even `PNodes`), for their respective state representation. Therefore

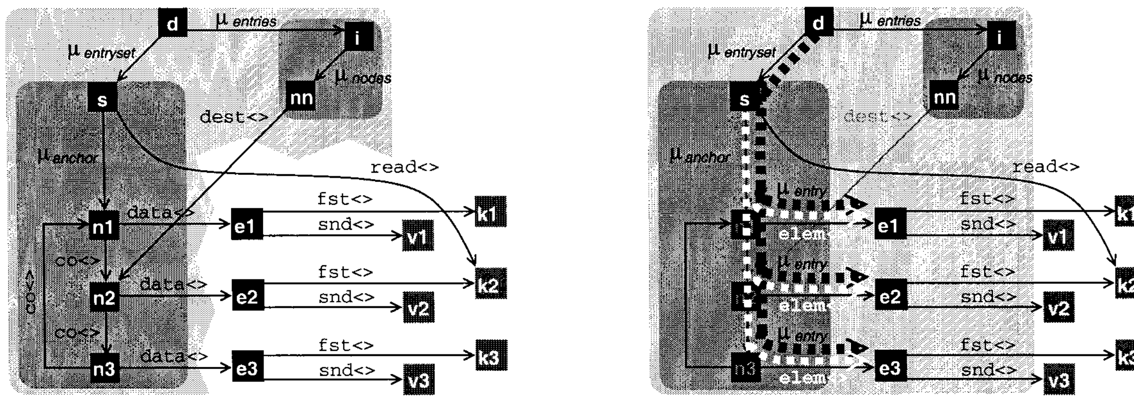


Figure 6.2: Structural interpretation of the moded object graph

the correct classification for the `next`-links is `co`: This tells us that `n2` and `n3` are, like `n1`, state-representing components of `s` in the state representation of the set (and thus of the map). The result of this interpretation of base-JaM's `rep`, `free` and `co` labels on the edges is depicted in the right hand side of figure 6.2. However, what worked for the nodes' `next`-links does not work for their `data` references: First, the value stored in a node, if it is an object reference, does not mean that the target is a constituent of the object structure formed by the linked nodes. Second, the element objects of the set $S = \{\mathtt{e1}, \mathtt{e2}, \mathtt{e3}\}$ of `Pair` objects reified by `s` are not parts of the set's `PSetImp` implementation (but of the map's implementation).

The correct classification of the `data` references in the `Nodes`, and more generally, of object references stored in data structures or container objects, requires the additional flexibility provided in JaM only by the new association modes and correlations. Giving the references stored in the `Nodes` the mode `data<>` lifts from a field name to the type level the information that the targets are the `Nodes`' *data*. The set representative's `anchor` reference $\mathtt{s} \xrightarrow{\mathtt{rep<data=elem<>>}} \mathtt{n1}$ specifies that the *data* of `n1` and its co-objects are `s`'s *elements*. And the map representative's `entryset` reference $\mathtt{d} \xrightarrow{\mathtt{rep<elem=rep<fst=key<>,snd=value<>>>}} \mathtt{s}$ specifies that `s`'s *elements* are state-representing components of `d` and that the *first* and *second* elements in these elements are, respectively, `d`'s *keys* and *values.*

3. Example Path Derivation. More systematically we can determine the objects' relationships by deriving, step by step, the modes of the paths in the object graph (see the left hand side of fig. 6.2): `Anchor` reference $\mathtt{s} \xrightarrow{\mathtt{rep<data=elem<>>}} \mathtt{n1}$ combines with $\mathtt{n1} \xrightarrow{\mathtt{data<>}} \mathtt{e1}$ to a path $\mathtt{s} \dashrightarrow \mathtt{e1}$ of mode `elem<>`. With this path, the `entryset` reference $\mathtt{d} \xrightarrow{\mathtt{rep<elem=rep<fst=key<>,snd=value<>>>}} \mathtt{s}$ combines to a path $\mathtt{d} \dashrightarrow \mathtt{e1}$ of mode `rep<fst=key<>,snd=value<>>`. This means that `e1` is an component of `d` whose `fst` and `snd` paths extend $\mathtt{d} \dashrightarrow \mathtt{e1}$ to `key` and `value` paths of `d`, i.e., that `e1` is an entry object. The same works for paths through the other nodes `n2` and `n3`: `Anchor` reference $\mathtt{s} \xrightarrow{\mathtt{rep<data=elem<>>}} \mathtt{n1}$ is extended by the next-links $\mathtt{n1} \xrightarrow{\mathtt{co<>}} \mathtt{n2}$ and

The class of e1, e2, e3:

```
class Pair {
   fst<> Object  fst;
   snd<> Object  snd;
   ...
}
```

The class of n1, n2, n3:

```
class PNode {
   co<> PNode   next;
   data<> Pair  data;
   ...
}
```

The class of nn:

```
class PNodeIt {
   dest<> PNode curdest;
   ...
}
```

The class of s:

```
class PSetImp {
   rep<data=elem<>> PNode  anchor;
   ...
}
```

The class of i:

```
class PDataIt {
   rep<dest=read<data=dest<>>>
                       PNodeIt  nodes;
   ...
}
```

The class of d:

```
class MapImp {
   rep<elem=rep<fst=key<>, snd=value<>>> PSetImp  entryset;
   ...
   obs value<> Object  lookup(read<> Object wanted)
   {  free<dest=rep<fst=key<>, snd=value<>>> PDataIt  entries;
      ...
   }
}
```

Figure 6.3: Mode declarations in the map example

n2 $\xrightarrow{\texttt{co<>}}$ n3 to paths s $\dashrightarrow$ n2 and s $\dashrightarrow$ n3 of the same mode. They combine with n2 $\xrightarrow{\texttt{data<>}}$ e2 and n3 $\xrightarrow{\texttt{data<>}}$ e3 to `elem` paths s $\dashrightarrow$ e2 and s $\dashrightarrow$ e3. With these paths, the `entryset` reference combines to paths d $\dashrightarrow$ e2 and d $\dashrightarrow$ e3 of mode `rep<fst=key<>,snd=value<>>`, which specify also for e2 and e3 that they are entry components of d.

On the iterator's side, nn's `dest<>` edge to n2 entails `dest`-paths to all nodes in the ring (not shown). Intuitively these paths mean that the nodes are the *destination* objects in the iteration which nn reifies. And i's `rep<dest=read<data=dest<>>>` link to nn specifies that the `data` objects of nn's destination objects are the *destination* objects in the iterative navigation along s's `elem`-links which i reifies: i's link to nn is extended to `dest`-paths i $\dashrightarrow$ e1, e2, and e3 by concatenation with nn's `dest`-paths and with the `data`-edges in their targets. Map d has a `free<dest=rep<fst=key<>,snd=value<>>>` reference to i (in its lookup method), whose extensions by i's `dest` paths are alternative paths to e1, e2 and e3 of the same mode `rep<fst=key<>, snd=value<>>`.

4. EXAMPLE DECLARATIONS. The mode-classification of the references in the object graph, from which the mode-classification of paths is derived, is specified in the program by the mode qualification of object reference types in declarations. The code fragments in figure 6.3 show the declarations of the fields and local variables which hold the object graph's handles in the map example. The complete program code can be found in appendix B.

6.2 Adapted Definitions

6.2.1 Syntax, Semantics, Typing

1. THE SYNTAX OF JAM PROGRAMS adapts the base-JaM syntax by association roles "$\mathbb{A}$" as additional alternative for base-JaM's mode-terms ("base-modes") and by the annotation of correlations to base-modes, to `null` and to `new`. The syntax rules which changed compared to the base-JaM-grammar (fig. 5.1 on page 71) are shown in figure 6.4. The complete JaM-grammar can be found in appendix A.

2. SEMANTICS AND TYPE SYSTEM. Most definitions remain literally the same as in base-JaM, changing only implicitly through the change of the definition of the set $\mathcal{M}$ of valid modes (which is a restriction of the mode-terms derived from nonterminal M that will be defined in paragraph *6*): Object graph edges $\langle o, \mu, \omega \rangle$, handles $\langle o, \mu, \omega \rangle$ in store, runtime-term and environments; source consistencies $\models_{\mathfrak{s}} om$, $\models_{\mathfrak{s}} \vec{\eta}$, and $\models_{\mathfrak{s},\vec{\eta}} e$; term reduction contexts; handle types $\mu\ c$, location-partitions $\mathcal{L}oc_{\mu\ c}$, type extensions $[\![\tau]\!]$, and type consistencies $\eta \models \Gamma$, $\models \mathfrak{s}$, $\models om$, and $\vec{\eta} \models \Gamma, \kappa, \mu_{\mathbf{r}}, X$. Their definitions will not be repeated here.

Explicit adaptions are required only for definitions in which modes occur verbatim. In particular, the mode system definitions will have to be reconsidered carefully. This is the subject of §6.2.3 further below. The other adaptions are straight-forward additions of empty or explicitly annotated correlation-sets:

Initial configuration. The mode of the call-link in the initial configuration is adapted from `read` to `read<>`. That is, the execution of a JaM program is the sequence of reduction steps starting

$$\texttt{new<>}\ c_n\texttt{().main()},\ \emptyset^{\mathsf{obs}}_{\langle \mathsf{nil}, \mathsf{read<>}, \mathsf{nil} \rangle}, \emptyset, \emptyset, \emptyset \implies e_1, \vec{\eta}_1, \mathfrak{s}_1, om_1, \mathfrak{g}_1 \implies \ldots$$

The changed reduction and typing rules are shown in figure 6.4 (for the complete set of definitions, see appendix A):

- Value and type of expression `this` are adapted by adding an empty correlation-set: Reduction rule {call} reduces an operation call expression to an inlined method body executed in an environment mapping `this` to a fresh location $\ell \in \mathcal{L}oc_{\mathsf{co<>}\ c}$ (instead of a base-JaM-location $\ell \in \mathcal{L}oc_{\mathsf{co}\ c}$), and a store with a handle $\langle \mathbf{r}, \mathsf{co<>}, \mathbf{r} \rangle$ (instead of $\langle \mathbf{r}, \mathsf{co}, \mathbf{r} \rangle$) at location ℓ, and edge $\mathbf{r} \xrightarrow{\mathsf{co<>}} \mathbf{r}$ added to the object graph (instead of $\mathbf{r} \xrightarrow{\mathsf{co}} \mathbf{r}$). Correspondingly, the typing rule [meth] types a method body in the context of type assumption `this`$: \mathsf{ref}\ \mathsf{co<>}\ c$ (instead of $\mathsf{ref}\ \mathsf{co}\ c$).

mode-term $\mu \in \mathrm{M} ::= (\mathtt{free} \mid \mathtt{rep} \mid \mathtt{co} \mid \underline{\mathbb{A}} \mid \mathtt{read})\underline{\mathtt{<}\Delta\mathtt{>}}$
correlations $\delta \in \Delta ::= (\mathbb{A}\text{=}\mathrm{M})^*$
expression $e \in \mathrm{E} ::= \mathtt{val}(\mathrm{N}) \mid \mathtt{destval}(\mathrm{N}) \mid \mathtt{null}\underline{\mathtt{<}\Delta\mathtt{>}} \mid \mathtt{new}\underline{\mathtt{<}\Delta\mathtt{>}}\ \mathbb{C}\mathtt{()} \mid \mathrm{E} \Leftarrow Id(\mathrm{E}^*)$

$$\{\text{call}\}\ \frac{\begin{array}{l} \mathbf{r} \in \mathbb{O}_c, \quad om(\mathbf{r}) \doteq \langle \ldots, F \rangle, \quad F(f) \doteq \kappa^\star\, \tau\, f\,(\overline{\mu_i\ c_i\ y_i})\{\overline{\mu'_j\ c'_j\ z_j}\,;\ s\} \\ \text{fresh } \ell \in [\![\mathsf{ref}\ \mathtt{co<>}\ c]\!],\ \overline{\text{fresh } \ell^y_i \in [\![\mathsf{ref}\ \mu_i\ c_i]\!]},\ \overline{\text{fresh } \ell^z_j \in [\![\mathsf{ref}\ \mu'_j\ c'_j]\!]} \\ \eta^\star = \{\mathtt{this} \mapsto \ell, \overline{y_i \mapsto \ell^y_i}, \overline{z_j \mapsto \ell^z_j}\} \\ \mathfrak{s}' = \mathfrak{s}[\ell \mapsto \langle \mathbf{r}, \mathtt{co<>}, \mathbf{r}\rangle, \overline{\ell^y_i \mapsto \langle \mathbf{r}, \mu_i, \mathbf{o}_i\rangle}, \overline{\ell^z_j \mapsto \langle \mathbf{r}, \mu'_j, \mathsf{nil}\rangle}] \\ \mathfrak{g}' = \mathfrak{g} \ominus \overline{\mathbf{s} \xrightarrow{\mu''_i} \mathbf{o}_i} \oplus \mathbf{r} \xrightarrow{\mathtt{co<>}} \mathbf{r} \oplus \overline{\mathbf{r} \xrightarrow{\mu_i} \mathbf{o}_i} \end{array}}{\langle \mathbf{s}, \mu_\mathbf{r}, \mathbf{r}\rangle \Leftarrow f\,(\overline{\langle \mathbf{s}, \mu''_i, \mathbf{o}_i\rangle}),\ \eta^\kappa_h, \mathfrak{s}, om, \mathfrak{g} \longrightarrow \ll s \gg,\ \eta^\kappa_h \bullet \eta^{\star\kappa^\star}_{\langle \mathbf{s}, \mu_\mathbf{r}, \mathbf{r}\rangle}, \mathfrak{s}', om, \mathfrak{g}'}$$

$$\{\text{null}\}\ \frac{h \doteq \langle \mathbf{s}, \mu_\mathbf{r}, \mathbf{r}\rangle}{\mathtt{null<}\delta\mathtt{>},\ \eta^\kappa_h, \mathfrak{s}, om, \mathfrak{g} \longrightarrow \langle \mathbf{r}, \mathtt{free<}\delta\mathtt{>}, \mathsf{nil}\rangle,\ \eta^\kappa_h, \mathfrak{s}, om, \mathfrak{g}}$$

$$\{\text{new}\}\ \frac{\begin{array}{llll} h \doteq \langle \mathbf{s}, \mu_\mathbf{r}, \mathbf{r}\rangle & \vdash \mathit{FldsMths}(c) \doteq \langle \{\overline{x_i : \mathsf{ref}\ \mu_i\ c_i}\}, F\rangle & h' = \langle \mathbf{r}, \mathtt{free<}\delta\mathtt{>}, \mathbf{o}\rangle \\ \text{fresh } \mathbf{o} \in \mathbb{O}_c & \overline{\text{fresh } \ell_i \in [\![\mathsf{ref}\ \mu_i\ c_i]\!]} \quad \varrho = \{\overline{x_i \mapsto \ell_i}\} & \overline{h_i = \langle \mathbf{o}, \mu_i, \mathsf{nil}\rangle} \end{array}}{\mathtt{new<}\delta\mathtt{>}\ c\mathtt{()},\ \eta^\kappa_h, \mathfrak{s}, om, \mathfrak{g} \longrightarrow h',\ \eta^\kappa_h, \mathfrak{s}[\ell_i \mapsto h_i], om[\mathbf{o} \mapsto \langle \varrho, F\rangle], \mathfrak{g} \oplus h'}$$

$$[\text{meth}]\ \frac{\begin{array}{l} \vdash t\ \mathbf{ok} \quad \overline{\vdash t_i\ \mathbf{ok}} \quad \overline{\vdash t'_j\ \mathbf{ok}} \\ \Gamma = \mathtt{this} : \mathsf{ref}\ \mathtt{co<>}\ c,\ \overline{x_i : \mathsf{ref}\ t_i},\ \overline{z_j : \mathsf{ref}\ t'_j} \quad \vdash \Gamma\ \mathbf{ok} \quad \Gamma, \kappa \vdash s : t \end{array}}{\vdash \kappa\ t\ f\,(\overline{t_i\ x_i})\{\overline{t'_j\ z_j}\,;\ s\}\ \mathbf{defs}\ f}$$

$$[\text{null}]\ \frac{\vdash c\ \mathbf{ok}}{\Gamma, \kappa \vdash \mathtt{null<}\delta\mathtt{>} : \mathtt{free<}\delta\mathtt{>}\ c} \qquad [\text{new}]\ \frac{\vdash c\ \mathbf{ok}}{\Gamma, \kappa \vdash \mathtt{new<}\delta\mathtt{>}\ c\mathtt{()} : \mathtt{free<}\delta\mathtt{>}\ c}$$

Figure 6.4: Changed syntax, reduction, and typing rules

- The syntax of the `null`-expression is extended so that it specifies the correlation-set δ to be added to the `free` mode of its value, the nil-handle. Reduction rule {null} reduces $e = \mathtt{null<}\delta\mathtt{>}$ to $\langle \mathbf{r}, \mathtt{free<}\delta\mathtt{>}, \mathsf{nil}\rangle$ (instead of $\langle \mathbf{r}, \mathtt{free}, \mathsf{nil}\rangle$), and typing rule [null] assigns to $e = \mathtt{null<}\delta\mathtt{>}$ the JaM-type $\mathtt{free<}\delta\mathtt{>}\ c$ (instead of $\mathtt{free}\ c$).
- The syntax of the creation expressions is extended to specify the correlation-set δ to be added to the `free` mode of its value, the handle to the new object. Reduction rule {new} reduces $e = \mathtt{new<}\delta\mathtt{>}\ c\mathtt{()}$ to $\langle \mathbf{r}, \mathtt{free<}\delta\mathtt{>}, \omega\rangle$ (instead of $\langle \mathbf{r}, \mathtt{free}, \omega\rangle$), and adds to the object-graph the edge $\mathbf{r} \xrightarrow{\mathtt{free<}\delta\mathtt{>}} \omega$ (instead $\mathbf{r} \xrightarrow{\mathtt{free}} \omega$). Correspondingly, typing rule [new] assigns to e the type $\mathtt{free<}\delta\mathtt{>}\ c$ (instead of $\mathtt{free}\ c$).

The annotation of correlations to `new` and `null` is the easiest way to make initial handles and nil-handles available that are compatible to any mode desired while ensuring that the mode in the computation and in the type inference are the same. Alternatively, one could introduce a new mode `free<*>` that is mode-compatible to any mode `free<δ>` and use it as the mode of initial and nil-handles. This solution would require us to extend the set of modes and the mode compatibility relation $\leq_{\mathrm{m}}$.

In the rule [rtype] for valid range types $\tau = \mu\ c$ of variables, parameters and results the condition "$\mu \in \mathcal{M}$" refers to the set of *valid* modes. That is, μ is required to be a valid mode, $\vdash \mu$ **ok**. This restriction of the mode-terms derived from nonterminal M will be defined in paragraph *6*. For emphasis, the rule could be rewritten

$$\text{[rtype]}\ \frac{\vdash \mu\ \mathbf{ok} \qquad \vdash c\ \mathbf{ok}}{\vdash \mu\ c\ \mathbf{ok}}$$

With the extension to full modes it should be clarified that substitutions $\mu' = \mu[\mathtt{read}/\mathtt{free}]$ in reduction rule $\{\mathrm{rd}_{cp}\}$ and $\tau' = \tau[\mathtt{read}/\mathtt{free}]$ in typing rule $[\mathrm{rd}_{cp}]$ should mean to change not just μ's and τ's *base*-modes but *all* occurrences of '`free`' in μ and τ. However, since nested `free` modes are excluded from valid modes (paragraph *6*), it is actually sufficient to consider just the base-mode for replacement:

$$\begin{aligned}(m\texttt{<}\delta\texttt{>}\ c)[\mathtt{read}/\mathtt{free}] &=_{\mathrm{df}} m[\mathtt{read}/\mathtt{free}]\texttt{<}\delta\texttt{>}\ c\\ (m\texttt{<}\delta\texttt{>})[\mathtt{read}/\mathtt{free}] &=_{\mathrm{df}} m[\mathtt{read}/\mathtt{free}]\texttt{<}\delta\texttt{>}\end{aligned} \qquad m[\mathtt{read}/\mathtt{free}] =_{\mathrm{df}} \begin{cases}\mathtt{read} & \text{if } m = \mathtt{free}\\ m & \text{otherwise}\end{cases}$$

3. CONSISTENCY PROPERTIES below the level of potential access paths are independent from the mode system, as long as the signature $\Sigma(\mu\ c)$ of handles is still calculated from $\mathit{FldsMths}(c)$ by adapting the modes μ_i in it to $\mu\circ\mu_i$.

Proposition 5 If $e_0, \eta_0, \mathfrak{s}_0, om_0, \mathfrak{g}_0 \Longrightarrow^* e', \vec{\eta}', \mathfrak{s}', om', \mathfrak{g}'$ is a reduction defined relative to a program p with $\vdash p\ \ \mathbf{start}\ e_0$ then

$$\mathfrak{g}' = ogr(e', \vec{\eta}', \mathfrak{s}')$$

Proof: The proof is nearly identical to the base-JaM-version of the theorem (Proposition 2). Deviations are only necessary where the reduction rules changed (see paragraph *2*). It is easy to see that the changes to the modes of handles $\langle o, \mu, \omega\rangle$ added and removed in runtime model $e', \vec{\eta}', \mathfrak{s}'$ are the same as the changes to the modes of edges $\langle o, \mu, \omega\rangle$ added and removed edges in the object graph. Hence execution in JaM still preserves compatibility. ■

Lemma 5 (Type preservation) If $e, \vec{\eta}, \mathfrak{s}, om, \mathfrak{g} \Longrightarrow e', \vec{\eta}', \mathfrak{s}', om', \mathfrak{g}'$ is a reduction defined relative to a program p with $\vdash p\ \ \mathbf{start}\ e_0$ then

$$\begin{aligned} &\Gamma, \kappa \vdash_X e : \tau \quad \wedge\ \vec{\eta} \models \tilde{\mu}, \Gamma, \kappa, X \quad \wedge \models \mathfrak{s}, om\\ \Rightarrow\ \exists X'.\ &\Gamma, \kappa \vdash_{X'} e' : \tau \ \ \wedge\ \vec{\eta}' \models \tilde{\mu}, \Gamma, \kappa, X' \ \ \wedge \models \mathfrak{s}', om'\end{aligned}$$

Proof: The proof from the base-JaM-version of this theorem (Lemma 1) can be copied here, except that the mode of `this`, `null`, and `new` has to be adapted. But in all cases (reduction rules {null}, {new}, and {call}), this change is the same in reduction as it is in typing. Hence the proof still goes through. ■

Theorem 6 (Type consistency) If $e_0, \eta_0, \mathfrak{s}_0, om_0, \mathfrak{g}_0 \Longrightarrow^* e, \vec{\eta}, \mathfrak{s}, om, \mathfrak{g}$ is a reduction defined relative to a program p with $\vdash p$ **start** e_0 then $\exists X, \tau$.

$$\models \mathfrak{s}, om \;\wedge\; \emptyset, \texttt{obs} \vdash_X e : \tau \;\wedge\; \vec{\eta} \models \texttt{read<>}, \emptyset, \texttt{obs}, X$$

Proof: The proof goes the same as that for its base-JaM version (Lemma 4). The difference is that the mode is `read<>` instead of just `read`, that the receiver expression `new<`δ`>` c`()` in operation call expression e_0 has mode `free<`δ`>` instead of just `free`, and that one has to use JaM's Lemma 5 instead of base-JaM's Lemma 1. ■

6.2.2 The Higher-Level View

4. Potential Access Paths, Sanctuaries, State. The meaning of the modes is captured formally in the rules for the derivation of potential access paths $\pi \in PAP(o, \mu, \omega)$ and in the definition of objects' sanctuaries $Sanc(o)$. There are three rules for potential access paths in an object graph $\mathfrak{g}$ labeled with full modes which are listed in figure 6.5: The base-case is an edge $o \xrightarrow{\mu} \omega \in \mathfrak{g}$. As in base-JaM (cf. fig. 5.9 on page 88), the extension $\pi_1 \bullet \pi_2$ of a μ-path $\pi_1 \in PAP(o, \mu, q)$ by a co-path $\pi_2 \in PAP(q, \texttt{co<>}, \omega)$ is another potential access path in $PAP(o, \mu, \omega)$.[1] New is the rule for the extension by association paths. An association path $\pi_2 \in PAP(q, \alpha\texttt{<>}, \omega)$ extend a path $\pi_1 \in PAP(o, m\texttt{<}\ldots, \alpha\texttt{=}\mu, \ldots\texttt{>}, q)$ with the necessary correlation to a potential access path $\pi_1 \bullet \pi_2 \in PAP(o, \mu, \omega)$.

Definition 6 Potential access paths $PAP(o, \mu, \omega)$ are defined based on the derivation rules in figure 6.5 and based on extended graph $\mathfrak{g}^\star$. Extended graph $\mathfrak{g}^\star$, ownership paths $Osh(o, \omega)$, sanctuaries $Sanc(o)$, and state representation $StRep(o)$ are defined as in base-JaM modulo correlations.

$$\begin{array}{lll}
\mathfrak{g}^\star & =_{\text{df}} & \mathfrak{g} \uplus \{\omega \xrightarrow{\texttt{co<>}} o \mid o \xrightarrow{\texttt{co<>}} \omega \in \mathfrak{g}\} \\
PAP_{\mathfrak{g}}(o, \mu, q) & =_{\text{df}} & \{\pi \mid \mathfrak{g}^\star \vdash \pi \in PAP(o, \mu, q)\} \\
Osh_{\mathfrak{g}}(o, \omega) & =_{\text{df}} & PAP_{\mathfrak{g}}(o, \texttt{rep<...>}, \omega) \cup PAP_{\mathfrak{g}}(o, \texttt{free<...>}, \omega) \\
Sanc_{\mathfrak{g}}(o) & =_{\text{df}} & \bigcup\limits_{\omega \text{ su. th. } PAP_{\mathfrak{g}}(o, \texttt{rep<...>}, \omega) \neq \emptyset} \Big(\{\omega\} \cup Sanc_{\mathfrak{g}}(\omega)\Big) \\
StRep_{\mathfrak{s},om}(o) & =_{\text{df}} & \{o\} \cup \bigcup\limits_{\omega \text{ su. th. } PAP_{fgr_{om}(\mathfrak{s})}(o, \texttt{rep<...>}, \omega) \neq \emptyset} StRep_{\mathfrak{s},om}(\omega)
\end{array}$$

Other definitions do not change at all, like composite state $CState_{\mathfrak{s},om}(o)$, field locations $flds(o)$ and field subgraph $fgr_{om}(\mathfrak{s})$.

Remark 1. The concatenation of correlation-carrying paths with association paths allows for much more complicated potential access paths than base-JaM's μ-reference

[1] Note that in *valid* modes, which paragraph *6* will define, base-modes `co` and $\alpha \in \mathbb{A}$ never come with any correlations.

$$\frac{o \xrightarrow{\mu} \omega \in \mathfrak{g}}{\mathfrak{g} \vdash o \xrightarrow{\mu} \omega \in PAP(o,\mu,\omega)} \qquad \frac{\mathfrak{g} \vdash \pi_1 \in PAP(o,\mu,q) \quad \mathfrak{g} \vdash \pi_2 \in PAP(q,\texttt{co<>},\omega)}{\mathfrak{g} \vdash \pi_1 \bullet \pi_2 \in PAP(o,\mu,\omega)}$$

$$\frac{\mathfrak{g} \vdash \pi_1 \in PAP(o, m\texttt{<}\ldots,\alpha\texttt{=}\mu,\ldots\texttt{>}, q) \quad \mathfrak{g} \vdash \pi_2 \in PAP(q,\alpha\texttt{<>},\omega)}{\mathfrak{g} \vdash \pi_1 \bullet \pi_2 \in PAP(o,\mu,\omega)}$$

Figure 6.5: Potential access paths in object graphs labeled with full modes

followed by co-references. In order to reduce the complexity of reasoning about these paths to a size manageable within the space of a dissertation, a restriction at a very basic level will be imposed: The nesting structure of *valid* modes $\mu \in \mathcal{M}$, that replace base-JaM's modes $\mathcal{M} = \{\texttt{free}, \texttt{rep}, \texttt{co}, \texttt{read}\}$, will be constrained. The effect is that certain, hard-to-handle combinations of two paths to a potential access paths (which, at least in the map example, are not needed) can simply not occur in a JaM object graph (labeled with *valid* modes).

Remark 2. In base-JaM, μ-paths were the extensions $o \xrightarrow{\mu} q \xrightarrow{\texttt{co}}{}^* \omega$ of a μ-edge by an optional sequence of co-edges. In full JaM, potential access paths are the (potentially trivial) further extensions $o \xrightarrow{\mu} q \xrightarrow{\texttt{co}}{}^* q_1 \bullet \pi_1 \bullet \ldots \bullet \pi_n$ of a base-JaM path by optional sequences of association paths $\pi_i \in PAP(q_i, \alpha_i\texttt{<>}, q_{i+1})$ ending in $q_{n+1} = \omega$. The *mode* of this path depends solely on the initial edge's mode μ and the sequence $\alpha_1 \ldots \alpha_n$ of the paths' association roles. The initial edge and the association role-sequence is captured in the notion of a path's *shape*:

Definition 7 A path $\pi \in PAP(o,\mu,\omega)$ has ***shape*** "$o \xrightarrow{\mu} q \xrightarrow{\texttt{co}}{}^* \bullet \xrightarrow{\alpha_1 \ldots \alpha_n} \bullet$" if $\pi = o \xrightarrow{\mu} q \xrightarrow{\texttt{co}}{}^* q_1 \bullet \pi_1 \bullet \ldots \bullet \pi_n$, i.e., if π starts with the edge $o \xrightarrow{\mu} q$ followed by an optional sequence $q \xrightarrow{\texttt{co}}{}^* q_1$ of co-edges, and then a sequence of association paths $\pi_i \in PAP(q_i, \alpha_i\texttt{<>}, q_{i+1})$ with some end-point $q_{i+1} = \omega$. Two paths π_1 and π_2 are ***shape-equivalent*** if they have the same shape, i.e., they start with the same edge, are extended by possibly different co-edges and association paths, however, the sequence of the association paths' association roles is the same.

Remark 3. The base-mode $m \in \{\texttt{free}, \texttt{rep}, \texttt{co}, \texttt{read}\} \cup \mathbb{A}$ clearly is still the *main* classification of potential access paths. The annotated correlations δ mean an orthogonal, second order classification w.r.t. the modes of the potential access path's potential extensions by association paths of certain association roles. Since the latter modes recursively specify the modes of further extensions by association paths, the classification of a path π as a μ-path is a classification according to the base-modes $m_{\alpha_1 \ldots \alpha_n}$ of π's extensions by $\alpha_1 \ldots \alpha_n$-sequences of association paths to potential access paths. Therefore, full mode μ can also be understood as a mapping from sequences $\alpha_1 \ldots \alpha_n$

of association roles to base-modes $m_{\alpha_1 \ldots \alpha_n}$ written $\mu(\alpha_1 \ldots \alpha_n)$.[2] In particular, extensions of edges $o \xrightarrow{\mu} q$ by association path-sequences $q \overset{\alpha_1 \ldots \alpha_n}{\dashrightarrow} \omega$, i.e., paths of shape $o \xrightarrow{\mu} q \xrightarrow{\mathtt{co}}{}^{*} \bullet \overset{\alpha_1 \ldots \alpha_n}{\dashrightarrow} \bullet$, are paths with base-mode $\mu(\alpha_1 \ldots \alpha_n)$.

Definition 8 The use of a mode-term μ as a mapping $\mu : \mathbb{A}^* \longrightarrow \mathcal{B}_\perp$ is defined as follows:

$$\begin{array}{lll} \mu(\epsilon) & =_{\text{df}} m & \text{if } \mu = m\texttt{<}\ldots\texttt{>} \\ \mu(\alpha.\vec{\alpha}) & =_{\text{df}} \begin{cases} \mu'(\vec{\alpha}) \\ \perp \end{cases} & \begin{array}{l} \text{if } \mu = m\texttt{<}\ldots, \alpha{=}\mu', \ldots\texttt{>} \\ \text{otherwise} \end{array} \end{array}$$

In particular, formulæ will use $\mu(\epsilon)$ to refer to "μ's base-mode," i.e., the base-mode m which mode μ gives to the μ-paths themselves. Take for example map d's entry-set reference of mode $\mu_{entryset} = \texttt{rep<elem=rep<fst=key<>, snd=value<>>>}$:

$$\begin{array}{ll} \mu_{entryset}(\epsilon) & = \texttt{rep} \\ \mu_{entryset}(\texttt{elem}) & = \texttt{rep} \\ \mu_{entryset}(\texttt{elem.fst}) & = \texttt{key} \\ \mu_{entryset}(\texttt{elem.snd}) & = \texttt{value} \end{array}$$

Remark 4. Obviously the order of the correlations in the mode-term μ and possible repeated occurrence of the same correlation $\alpha{=}\mu'$ are irrelevant both for the main classification of the path, and for the classification of its extensions by association paths. More specifically, they are irrelevant for the mode's understanding as a mapping $\mu : \mathbb{A}^* \longrightarrow \mathcal{B}_\perp$. This consideration is captured in the notion of mode-equivalence:

Definition 9 Two mode-terms $\mu, \mu' \in \mathcal{M}$ are ***mode-equivalent***, $\mu \equiv \mu'$, if they have the same base-mode and their correlation-sets configure the same association role to mode-equivalent modes, in other words, if they are the same as mappings from association role-sequences to roles:

$$\mu \equiv \mu' \Leftrightarrow_{\text{df}} \forall \vec{\alpha} \in \mathbb{A}^* \bullet\ \mu(\vec{\alpha}) = \mu'(\vec{\alpha})$$

When we write a mode-term $\mu \in \mathcal{M}$, it is normally meant as a representative for the equivalence class $[\mu]_\equiv$ of modes equivalent to μ.

5. INTEGRITY INVARIANTS OF JAM SYSTEMS are adaptions of base-JaM's integrity invariants w.r.t. the new correlations (see figure 6.6).

- The ***Unique Owner*** property UO holds in graph $\mathfrak{g}$ if all objects have at most one owner, i.e., are at most target of a unique object's ownership paths.
- The ***Unique Head*** property UH holds in graph $\mathfrak{g}$ if the initial edge in all ownership paths to a `free` object is the same and has multiplicity one.
- The ***Mutator Control Path*** property MCP holds in graph $\mathfrak{g}$ and stack $\vec{\eta}$ if *mutators* were invoked on receivers $\mathbf{r}_i$ only through a sequence of calls along the edges $h_j, \ldots, h_i$ of an ownership path to $\mathbf{r}_i$.

[2]Actually, this can only be a mapping if μ does not have multiple, incompatible correlations for the same association roles. This is one of the conditions mode-terms have to satisfy to be *valid* modes $\mu \in \mathcal{M}$ (see paragraph *6*).

$$
\begin{array}{lll}
\mathfrak{g} \models \mathsf{UO} & \Leftrightarrow_{\mathrm{df}} \forall o, \tilde{o}, \omega \bullet & Osh_{\mathfrak{g}}(o, \omega) \neq \emptyset \;\wedge\; Osh_{\mathfrak{g}}(\tilde{o}, \omega) \neq \emptyset \;\Rightarrow\; \tilde{o} = o \\
\mathfrak{g} \models \mathsf{UH} & \Leftrightarrow_{\mathrm{df}} \forall o, \tilde{o}, \omega, h, \pi, \tilde{h}, \tilde{\pi} \bullet & h \bullet \pi \in PAP_{\mathfrak{g}}(o, \mathtt{free<\ldots>}, \omega) \wedge \tilde{h} \bullet \tilde{\pi} \in Osh_{\mathfrak{g}}(\tilde{o}, \omega) \\
 & & \Rightarrow\; \tilde{h} = h \wedge mult(h, \mathfrak{g}) = 1 \\
\mathfrak{g}, \vec{\eta} \models \mathsf{MCP} & \Leftrightarrow_{\mathrm{df}} \forall i \in \{1, \ldots, n\} \bullet & \kappa_i = \mathtt{mut} \Rightarrow \exists j \leq i \bullet\, h_j \bullet \ldots \bullet h_i \in Osh_{\mathfrak{g}}(\mathbf{r}_{j-1}, \mathbf{r}_i) \\
\mathfrak{g}, \vec{\eta} \models \mathsf{MC} & \Leftrightarrow_{\mathrm{df}} \forall i \in \{1, \ldots, n\}, o \bullet & \kappa_i = \mathtt{mut} \wedge \mathbf{r}_i \in Sanc_{\mathfrak{g}}(o) \\
 & & \Rightarrow\; \exists k \leq i \bullet\, \mathbf{r}_k = o \wedge \kappa_k = \mathtt{mut} \\
\multicolumn{3}{l}{\text{where } \vec{\eta} = \eta_1{}^{\kappa_1}_{h_1} \bullet \ldots \bullet \eta_n{}^{\kappa_n}_{h_n} \text{ with call-links } h_i = \langle \mathbf{r}_{i-1}, \mu_i, \mathbf{r}_i \rangle}
\end{array}
$$

Figure 6.6: JaM integrity invariants

- The ***Mutator Control*** property MC holds in graph $\mathfrak{g}$ and stack $\vec{\eta}$ if members of o's sanctuary are executing *mutators* only nested to mutator executions of o, and thus (indirectly) initiated by o through a sequence of calls.

6.2.3 The Full Mode System

6. VALID MODES. As mentioned above, there are conditions on the nesting structure of those mode-terms derived from nonterminal M which are to take the place of base-JaM's modes in full JaM. These conditions help to keep the increase of complexity of the formal treatment in the step to JaM within a manageable size.

Definition 10 A mode-term $m{<}\alpha_1{=}\mu_1, \ldots, \alpha_n{=}\mu_n{>}$ derived from axiom M with the rules in figure 6.4 is a ***valid mode***, or mode for short, in symbols, $\vdash \mu$ **ok**, if it satisfies the following conditions (see figure 6.7):

1. For each association role there is at most one correlation: $\alpha_i = \alpha_j \Rightarrow i = j$.
2. co-modes and association modes have no correlations: $m \in \{\mathtt{co}\} \cup \mathbb{A} \Rightarrow n = 0$.
3. There are no correlations to a co or free mode: $\mu_i(\epsilon) \notin \{\mathtt{co}, \mathtt{free}\}$
4. The correlations' modes must be valid modes: $\vdash \mu_i$ **ok**.

Let $\mathcal{M}$ be only the set of *valid* modes: $\mathcal{M} = \{\mu \mid \;\vdash \mu \textbf{ ok}\}$.

Condition 1 generally simplifies the formal treatment through uniqueness: Valid modes μ specify a unique mode for the extension $\pi \bullet \pi'$ of μ-paths π by association paths π' (cf. paragraph *1*). Valid modes can rightfully be considered mappings to unique base-modes $\mu(\vec{\alpha})$. Consequently, the call-link's mode can uniquely determine the mode as which the sender imports handles of association modes returned by the receiver (see paragraph *9* below).

Condition 2 simplifies the recursive construction of longer and longer potential access paths: Paths that can extend others, i.e., co- and association paths, can never be extended by JaM's new association paths, but only by base-JaM's simpler co-paths. Reasoning about co- and association paths $\pi \in PAP(o, \mu, \omega)$ with correlations

$$\text{[mode]}\ \frac{\begin{array}{l}\forall i,j \in \{1,\ldots,n\}\bullet\ \alpha_i = \alpha_j \Rightarrow \mu_i \equiv \mu_j \\ m \in \{\texttt{co}\} \cup \mathbb{A} \Rightarrow n = 0 \\ \forall i \in \{1,\ldots,n\}\bullet\ \mu_i \neq \texttt{free<...>} \wedge \mu_i \neq \texttt{co<>}\ \wedge \vdash \mu_i\ \mathbf{ok}\end{array}}{\vdash m\texttt{<}\alpha_1\texttt{=}\mu_1,\ldots,\alpha_n\texttt{=}\mu_n\texttt{>}\ \mathbf{ok}}$$

Figure 6.7: Mode-specific definitions for JaM (part 1)

would require some invariant about the consistency between these correlations and the correlations of paths $\pi' \in PAP(q, \mu', o)$ extensible to $\pi' \bullet \pi$. This generalization is left for future work; it is not needed for the example of maps and iterators.

Condition 3(a). In base-JaM, the possibility of new potential access paths by the supply of a co parameter through a co-call-link was handled by reasoning in the extended graph $\mathfrak{g}^\star$, with inverted co-edges. This ensured that each new path-extending co-path π with the new, received parameter handle had a precursor π' with the inverted call-link and handle argument instead. This technical trick does not work for co-paths which do not entirely consist of co-edges, like the co-paths $o \xrightarrow{m\texttt{<elem=co<>>}} \mathsf{s} \xrightarrow{\texttt{elem<>}} \mathsf{e}_i$ which o derives from set object s's association paths to the set elements. This co-path would mean that o and e_i should have the same owners. But if d owns the element e_i through the ownership path $\mathsf{d} \xrightarrow{\texttt{rep<...>}} \mathsf{e}_i$, it is not guaranteed that there is also an ownership path from d to o, since the co-path $o \xrightarrow{m\texttt{<elem=co<>>}} \mathsf{s} \xrightarrow{\texttt{elem<>}} \mathsf{e}_i$ has no inverse for extending $\mathsf{d} \xrightarrow{\texttt{rep<...>}} \mathsf{e}_i$ to o. Instead of introducing complicated constraints on the creation and exchange of handles with co-correlations to enforce the necessary invariants on co-paths and ownership paths, we avoid this problem at a more basic level by simply prohibiting all correlations to a co-mode.

Condition 3(b). The exchange of free handles and handles extensible to free paths moves the corresponding targets between composite sender and receiver objects. Excluding correlations to free and co modes ensures that free paths in JaM have the same structure as in base-JaM: A free edge followed by co-edges. This will help to reduce the complexity of reasoning about the preservation of unique object ownership (Lemma 23) and of a property on intermediate objects in free paths necessary for coherence (Lemma 26).

7. State Encapsulation: Controlling the Mutation of Objects. For the question whether a mutator may be invoked through a call-link $\mathsf{s} \xrightarrow{\mu} \mathbf{r}$, it is only relevant from what kind κ of method the call is made, and what the base-mode of μ are (correlations in μ say nothing about the target's status but about the targets of the call-link's extensions by association paths).[3] Hence the considerations for modes rep<...>, free<...>, co<>, and read<...> are the same as those for base-JaM's

[3]The mode $\mu(\alpha)$ of the correlation for association role α can become relevant in an effects system extension of JaM that uses effects region this.α for handing mutator calls through α-paths (§7.3.2).

$$\mathbf{Wr}(\mathtt{obs}) =_{\mathrm{df}} \{\mathtt{free<\ldots>}\}$$
$$\mathbf{Wr}(\mathtt{mut}) =_{\mathrm{df}} \{\mathtt{free<\ldots>}, \mathtt{rep<\ldots>}, \mathtt{co<>}\}$$

$$\begin{array}{ll}
m\mathtt{<}\delta\mathtt{>} & \leq^1_{\mathrm{m}} \mathtt{read<}\delta\mathtt{>} \\
\mathtt{free<}\delta\mathtt{>} & \leq^1_{\mathrm{m}} \mathtt{rep<}\delta\mathtt{>} \\
\mathtt{read<}\delta, \alpha{=}\mu, \delta'\mathtt{>} & \leq^1_{\mathrm{m}} \mathtt{read<}\delta, \delta'\mathtt{>} \\
\mathtt{read<}\delta, \alpha{=}\mu, \delta'\mathtt{>} & \leq^1_{\mathrm{m}} \mathtt{read<}\delta, \alpha{=}\mu', \delta'\mathtt{>} \quad \text{if } \mu \leq^1_{\mathrm{m}} \mu'
\end{array}$$

Figure 6.8: Mode-specific definitions for JaM (part 2)

modes `rep`, `free`, `co`, and `read` in §5.4.2: (cf. figure 6.8): Through `free` call-links, mutators may always be called, no matter their correlations: $\mathtt{free<\ldots>} \in \mathbf{Wr}(\kappa)$ for $\kappa = \mathtt{mut}$ and `obs`. Through `rep` and `co` call-links, the source may call mutators on the target from within mutators: $\mathtt{rep<\ldots>}, \mathtt{co<>} \in \mathbf{Wr}(\mathtt{mut})$. Through `read` call-links, mutators may never be called: $\mathtt{read<\ldots>} \notin \mathbf{Wr}(\kappa)$ for $\kappa = \mathtt{mut}$ and `obs`.

Handles of the new base-mode $\alpha \in \mathbb{A}$, like `read` handles, do not fix its target's membership in sanctuaries (relative to the source). Hence from the perspective of the sender's code, invocations of mutators through association handles are in general not guaranteed to be safe: $\alpha\mathtt{<\ldots>} \notin \mathbf{Wr}(\kappa)$ for $\kappa = \mathtt{mut}$ and `obs`.

8. MODE COMPATIBILITY. If two base-modes m and m' were treated as compatible in base-JaM ($m \leq_{\mathrm{m}} m'$), it *should* be possible to extend this to full modes that have the same correlations ($\mu = m\mathtt{<}\delta\mathtt{>} \leq_{\mathrm{m}} m'\mathtt{<}\delta\mathtt{>} = \mu'$). Moreover, there are new possibility for full modes similar to *width* and *depth subtyping* for record types: Removing a correlation from the mode's correlation-set should be safe since it means merely that some extensions of the converted handles are now no potential access paths any more. And the variation of the mode in a correlation to a compatible mode (covariance) should be safe since it means that the mode specified to the extensions of the converted handles varies in a compatible way. However, it is not really that simple:

First, the conversion of (`free`) handles $o \xrightarrow{\mathtt{free<>}} \omega_1$ to co-handles $o \xrightarrow{\mathtt{co<>}} \omega_1$ and association handles $o \xrightarrow{\alpha\mathtt{<>}} \omega_1$ is problematic: It means not only the potential extension of o-targeting paths $\pi = q \overset{\mu}{\dashrightarrow} o$ to ω_1 and its co-object, as in base-JaM. In JaM, these paths may be furthermore extended, depending on μ, by association path sequences $\omega_1 \overset{\alpha_1}{\dashrightarrow} \omega_2 \overset{\alpha_2}{\dashrightarrow} \omega_3 \ \ldots$ to more distant objects ω_k or, what is even harder to deal with, back to q or o. For reasoning about such paths when it comes to the preservation of the unique owner property, the four integrity invariants adapted base-JaM (§6.2.2) would not suffice. We would need to show an additional invariant about all sequences $\omega_1 \overset{\vec{\alpha}}{\dashrightarrow} \omega_{n+1}$ of association paths starting from targets ω_1 of `free<>` handles.

In order to keep for the formal treatment of conversion as simple as in base-JaM, let us ignore conversions (from `free<>`) to `co<>` and $\alpha\mathtt{<>}$ (which are not needed for the map example with iterators), and focus on the conversion of received `free` handles to rep and their subsequent storage in sub-objects as α-handles. Even if objects can then not create co- and association handles for themselves any more, they can still obtain them as parameters from other objects: For example, node object n1 can obtain co-handles $\texttt{n1} \xrightarrow{\texttt{co<>}} \texttt{n2}$ and data-handles $\texttt{n1} \xrightarrow{\texttt{data<>}} \texttt{e1}$ from set representative s through $\texttt{s} \xrightarrow{\texttt{rep<data=elem<>>}} \texttt{n1}$ if s calls n1's methods SetNext and SetData and supplies handles $\texttt{s} \xrightarrow{\texttt{rep<data=elem<>>}} \texttt{n2}$ or $\texttt{s} \xrightarrow{\texttt{elem<>}} \texttt{e2}$, respectively.

Second, depth-compatibility is problematic since a change from mode $\mu_{\mathrm{r}} = m\texttt{<}\alpha\texttt{=}\mu\texttt{>}$ to $\mu'_{\mathrm{r}} = m\texttt{<}\alpha\texttt{=}\mu'\texttt{>}$ on a handle h means not only *in the higher-level view* a harmless mode change of h's extensions by α-paths from μ-paths to $\mu' \geq_{\mathrm{m}} \mu$-paths, but also means *in the type system* a change in the handle's signature from the target's α-moded result and parameter types from $\mu_{\mathrm{r}} \circ \alpha\texttt{<>} = \mu$ to $\mu'_{\mathrm{r}} \circ \alpha\texttt{<>} = \mu' \geq_{\mathrm{m}} \mu$. Now, it is a well-established result in type-theory that parameter types do not change covariantly (but contravariantly), i.e., that changing the *parameter* type τ in a function type $\tau \rightarrow \sigma$ to a supertype $\tau' \geq \sigma$, i.e., a compatible type, does *not* produce a compatible function type $\tau' \rightarrow \sigma \geq \tau \rightarrow \sigma$ (on the contrary, $\tau' \rightarrow \sigma \leq \tau \rightarrow \sigma$). This result applies, *mutatis mutandis*, also to the conversion of correlations in JaM, since it may change an operation's parameter mode in the handle's signature: It could lead to an unsafe state in which, for example, a read-handle can be converted to a rep-handle. This example, and further manifestations of the same, general phenomenon in other type systems (e.g., const pointer types) will be discussed in §7.1.4.

The conversion of correlation $\alpha\texttt{=}\mu$ on h can not cause any problems if through the converted handle no operations with α-parameters can be invoked. Hence we can obtain a minimum of depth-conversion by treating handles of mode $\texttt{read<}\alpha\texttt{=}\mu\texttt{>}$ as compatible to handles of mode $\texttt{read<}\alpha\texttt{=}\mu'\texttt{>}$ if $\mu \leq_{\mathrm{m}} \mu'$ if we disallow the supply of α-parameters through read-handles. (For rep and free handles, on the other hand, it is more important to allow the composite source to store its μ-handles as α-handles in the target component.) This means that read-handles not only disallow the invocation of mutators but also the invocation of operations with association moded parameters: The iterator can use its read-handle to the set's nodes to read their data-handles as its dest-handles, but cannot pass them its dest-handles as their data-parameters. (The invocation of operations with co-parameters is prohibited since base-JaM.)

Third, the same problem exists with width-compatibility since a change from mode $\mu = m\texttt{<}\delta_i\texttt{>}$ to $\mu' = m\texttt{<>}$ on a handle h implies in the handle's signature a mode change of the target's co-moded result and parameter types from $\mu \circ \texttt{co<>} = \mu$ to $\mu' \circ \texttt{co<>} = \mu' \geq_{\mathrm{m}} \mu$. Also the removal of correlations on PNode handles, in conjunction with the use of co-parameterized operation SetNext, would allow to reach an unsafe state in which a read-handle can be converted to a rep-handle. This will be discussed in §7.1.4. Note that through read handles, co-parameters can anyway not be supplied (to avoid violating the Unique Owner invariant) because read handles do not reveal any information about their target's owner. Hence read handles can have their correlations converted away safely.

$$
\begin{array}{lll}
\mu_{\mathbf{r}}\circ\texttt{read<}\overline{\alpha_i\texttt{=}\mu_i}\texttt{>} & =_{\mathrm{df}} & \texttt{read<}\overline{\alpha_i\texttt{=}\mu_{\mathbf{r}}\circ\mu_i}\texttt{>} \\
\mu_{\mathbf{r}}\circ\texttt{free<}\overline{\alpha_i\texttt{=}\mu_i}\texttt{>} & =_{\mathrm{df}} & \texttt{free<}\overline{\alpha_i\texttt{=}\mu_{\mathbf{r}}\circ\mu_i}\texttt{>} \\
\mu_{\mathbf{r}}\circ\texttt{rep<}\overline{\alpha_i\texttt{=}\mu_i}\texttt{>} & =_{\mathrm{df}} & \texttt{read<}\overline{\alpha_i\texttt{=}\mu_{\mathbf{r}}\circ\mu_i}\texttt{>} \\
\mu_{\mathbf{r}}\circ\texttt{co<>} & =_{\mathrm{df}} & \mu_{\mathbf{r}} \\
\mu_{\mathbf{r}}\circ\alpha\texttt{<>} & =_{\mathrm{df}} & \mu' \quad \text{if } \mu_{\mathbf{r}} = m_{\mathbf{r}}\texttt{<}\ldots,\alpha\texttt{=}\mu',\ldots\texttt{>}
\end{array}
$$

$$
\frac{\begin{array}{l}\vdash \mathit{FldsMths}(c) = \langle\Gamma, F\rangle \qquad F(f) = \kappa\ \mu\ d\, f\,\texttt{(}\overline{\mu_i\ d_i\ y_i}\texttt{)\{}\ldots\texttt{\}} \\
\forall i, \vec{\alpha}\bullet ((\mu_{\mathbf{r}}\circ\mu_i)(\vec{\alpha}) = \texttt{read} \Rightarrow \mu_i(\vec{\alpha}) = \texttt{read}) \wedge (\mu_i(\vec{\alpha}) \in \{\texttt{co}\} \cup \mathbb{A} \Rightarrow \mu_{\mathbf{r}}(\epsilon) \notin \{\texttt{read}\} \cup \mathbb{A}) \\
\forall \vec{\alpha}, \alpha\bullet\ \mu(\vec{\alpha}) = \texttt{free} \wedge \mu(\vec{\alpha}.\alpha) \in \mathbb{A} \wedge \mu_{\mathbf{r}}\circ\mu(\vec{\alpha}.\alpha) \neq \texttt{read} \Rightarrow \mu_{\mathbf{r}}(\epsilon) \neq \texttt{read}\end{array}}{\vdash (\, f : \overline{\mu_{\mathbf{r}}\circ\mu_i\ d_i} \xrightarrow{\kappa} \mu_{\mathbf{r}}\circ\mu\ d\,) \in \Sigma(\mu_{\mathbf{r}}\ c)}
$$

Figure 6.9: Mode-specific definitions for JaM (part 3)

To sum up, besides being compatible to themselves,

- all modes are compatible to `read`-modes with the same correlations,
- all `free` modes are compatible the `rep` mode with the same correlations,
- all `read` modes are compatible to `read` modes with fewer correlations, and
- all `read` modes with a correlation are compatible to the `read` mode in which the correlation's mode μ is replaced by a mode μ' to which μ is compatible.

Definition 11 The mode compatibility relation $(\leq_{\mathrm{m}}) \subseteq \mathcal{M} \times \mathcal{M}$ in JaM is the transitive reflexive closure of the relation $\leq^1_{\mathrm{m}}$ defined in figure 6.8.

9. Import of Returned Handles. When a handle is returned in a step with $\{\mathrm{ret}\}$ then, as in base-JaM, some adaption of the mode is necessary (see figure 6.9). W.r.t. the old base-modes, this adaption is the same as in base-JaM: `read`- and `rep`-handles are imported as `read`-handles, `free`-handles as `free`-handles, and `co`-handles are imported as handles of the call-link's mode $\mu_{\mathbf{r}}$. A handle of association mode $\alpha\texttt{<>}$ should be imported as a handle of the mode μ' which the call-link's mode $\mu_{\mathbf{r}} = m_{\mathbf{r}}\texttt{<}\ldots,\alpha\texttt{=}\mu',\ldots\texttt{>}$ specifies for α. This is the same mode as the mode of the combination $\mathbf{s} \xrightarrow{\mu_{\mathbf{r}}} \mathbf{r} \xrightarrow{\alpha\texttt{<>}} \omega$ of the call-link and the returned handle in the receiver. The return of the handle $\mathbf{r} \xrightarrow{\alpha\texttt{<>}} \omega$ shortens this μ'-path to a direct μ'-handle $\mathbf{s} \xrightarrow{\mu'} \omega$, just like the return of a `co`-handle $\mathbf{r} \xrightarrow{\texttt{co<>}} \omega$ shortens the $\mu_{\mathbf{r}}$-path $\mathbf{s} \xrightarrow{\mu_{\mathbf{r}}} \mathbf{r} \xrightarrow{\texttt{co<>}} \omega$ to the direct $\mu_{\mathbf{r}}$-handle $\mathbf{s} \xrightarrow{\mu_{\mathbf{r}}} \omega$.

A correlation $\alpha_i\texttt{=}\mu_i$ in the mode of a handle $\mathbf{r} \xrightarrow{\mu} \omega$ of the receiver specifies that its extensions $\pi' = \mathbf{r} \xrightarrow{\alpha\texttt{<>}} \omega \bullet \pi_i$ by α_i-paths are potential access paths $\pi' \in \mathit{PAP}(\mathbf{r}, \mu_i, \omega)$ of mode μ_i. When the receiver returns handles with such correlations, the sender will now have a potential access path $\pi = \mathbf{s} \xrightarrow{\mu_{\mathbf{r}}\circ\alpha\texttt{<>}} \omega \bullet \pi_i$. It is as if the receiver "returned" its potential access path π' to the sender as a "virtual result." (Analogously, "virtual parameters" are supplied when a parameter handle with correlations is supplied.) Hence in generalization of returned handles (paths of length one), when the receiver

$$
\begin{array}{l}
\Sigma(\texttt{PSetImp}) = \\
\{\ \texttt{contains} : (\texttt{read<> Pair}) \xrightarrow{\texttt{obs}} \texttt{boolean}, \\
\quad \texttt{Add} \qquad\ : (\texttt{elem<> Pair}) \xrightarrow{\texttt{mut}} \texttt{void}, \\
\quad \texttt{Remove} \quad : (\texttt{read<> Pair}) \xrightarrow{\texttt{mut}} \texttt{elem<> Pair}, \\
\quad \texttt{elements} : () \xrightarrow{\texttt{obs}} \texttt{free<dest=elem<>> PIter} \\
\} \\
\Sigma(\texttt{rep<elem=rep<fst=key<>, snd=value<>>> PSetImp}) = \\
\{\ \texttt{contains} : (\texttt{read<> Pair}) \xrightarrow{\texttt{obs}} \texttt{boolean}, \\
\quad \texttt{Add} \qquad\ : (\texttt{rep<fst=key<>, snd=value<>> Pair}) \xrightarrow{\texttt{mut}} \texttt{void}, \\
\quad \texttt{Remove} \quad : (\texttt{read<> Pair}) \xrightarrow{\texttt{mut}} \texttt{rep<fst=key<>, snd=value<>> Pair}, \\
\quad \texttt{elements} : () \xrightarrow{\texttt{obs}} \texttt{free<dest=rep<fst=key<>, snd=value<>>> PIter} \\
\}
\end{array}
$$

(For accordance with the limited JaM syntax, expand $\texttt{boolean}$ to $\texttt{read<> Boolean}$ and $\texttt{void}$ to $\texttt{read<> Void}$, and define dummy classes $\texttt{Boolean}$ and $\texttt{Void}$.)

Figure 6.10: Signature of $\texttt{MapImp}$ object's $\texttt{entryset}$ handle

"returns" a potential access path π' of mode μ_i, then the mode of the corresponding imported path in the sender should be $\mu_{\mathbf{r}}\circ\mu_i$. Since this mode is derived from the correlation in the mode $\mu_{\mathbf{r}}\circ\mu$ of the returned handle $\mathbf{s} \xrightarrow{\mu_{\mathbf{r}}\circ\mu} \omega$, this means that the α_i-correlation in the handle's mode must have the mode $\mu_{\mathbf{r}}\circ\mu_i$. That is, the import operator $\circ$ should be *associative*: $\mu_{\mathbf{r}}\circ(\mu\circ\alpha_i\texttt{<>}) = \mu_{\mathbf{r}}\circ\mu_i = (\mu_{\mathbf{r}}\circ\mu)\circ\alpha_i\texttt{<>}$. In other words, mode import $\circ$ is defined by *recursively importing* the modes of correlations in the imported mode μ.

10. SIGNATURE OF HANDLES. As in base-JaM, the definition of handles' signatures $\Sigma(\mu_{\mathbf{r}}\ c)$ is based on the adaption of *FldsMths*(c), which implicitly contains the signature $\Sigma(c)$ of c-objects, by the use of mode import "$\circ$" (see figure 6.9). As an example, figure 6.10 shows how the signature $\Sigma(\texttt{PSetImp})$ of $\texttt{PSetImp}$ objects is adapted to the signature $\Sigma(\mu_{\mathbf{r}}\ \texttt{PSetImp})$ of $\texttt{MapImp}$ objects' $\texttt{entryset}$ handles with mode $\mu_{\mathbf{r}} = \texttt{rep<elem=rep<fst=key<>, snd=value<>>>}$.

More detailed considerations are necessary for the *conditions* under which a method f of c-objects can be used through handles of type $\mu_{\mathbf{r}}\ c$ and with signature $\Sigma(\mu_{\mathbf{r}}\ c)$:

Regarding the new case of association moded parameters in JaM, observe that a parameter of mode $\alpha\texttt{<>}$ means that the receiver $\mathbf{r}$ does not know about the target's owner. But it does not mean there are no expectations toward the owner of the supplied parameter object at all (that would be a $\texttt{read}$ parameter). The receiver expects a parameter object owned by that object o, should it exist, which is "destined" to own *all* its α-objects since there is a path of edges from o to $\mathbf{r}$ that is extended by any α-path to an *ownership* path.

1. $\mu_{\mathbf{r}}\circ\mu_i = \texttt{read<...>} \Rightarrow \mu_i = \texttt{read<...>}$. All cases where a non-$\texttt{read}$ parameter in $\Sigma(c)$ ($\mu_i \neq \texttt{read<...>}$) becomes a $\texttt{read}$ parameter in $\Sigma(\mu_{\mathbf{r}}\ c)$ ($\mu_{\mathbf{r}}\circ\mu_i = \texttt{read<...>}$)

are problematic and have to be excluded: In JaM, as in base-JaM, methods with `rep` parameter cannot be called at all, and methods with `co` parameters cannot be called through `read` call-links since the caller cannot guarantee that its handles of corresponding mode $\mu_\mathbf{r}\circ\texttt{rep} = \texttt{read}$ or $\texttt{read}\circ\texttt{co} = \texttt{read}$ point to objects with the owner desired by the receiver. In JaM, the same problem additionally arises with the supply of α-parameters through call-links correlating α to a `read`-mode, i.e., where $\mu_\mathbf{r}\circ\alpha\texttt{<>} = \texttt{read<...>}$.

2. $\mu_i = \texttt{co<>} \Rightarrow \mu_\mathbf{r} \neq \alpha\texttt{<>}$. Like the targets of two `read`-handles, the targets of two α-handles do not necessarily have the same owner: There is no guarantee that there is any object "destined" to own *all* α-objects of the caller. But without this one, each of the α-objects could have a different owner. Hence, the same way it is not safe to link two `read`-targets by supplying a `read`-handle as `co`-parameter through a `read`-call-link, it is not safe to link two α-targets by the supply of an α-handle as `co`-parameter through an α-call-link.
3. $\mu_i = \alpha\texttt{<>} \Rightarrow \mu_\mathbf{r} \neq \texttt{read<>}$. The supply of association moded parameters through `read`-call-links was prohibited in paragraph *8* in order to make the conversion of correlations (depth-compatibility) safe.
4. $\mu = \texttt{free<}\alpha\texttt{=}\beta\texttt{<>,...>} \wedge \mu_\mathbf{r}\circ\mu(\alpha) \neq \texttt{read} \Rightarrow \mu_\mathbf{r} \neq \texttt{read<...>}$. New in JaM is a condition on the *result* of operations in handle signatures, but indirectly it also has to do with parameter passing: Correlations $\alpha\texttt{=}\mu$ in a handle's mode specify the mode μ by which the source can access the target's α-objects. The recursive, associative definition of $\circ$ ensured that the mode $(\mu_\mathbf{r}\circ\mu)\circ\alpha\texttt{<>}$ by which the sender $\mathbf{s}$ can access returned object $\mathbf{o}$'s α-results through returned handle $\mathbf{s} \xrightarrow{\mu_\mathbf{r}\circ\mu} \mathbf{o}$ is the same as the mode $\mu_\mathbf{r}\circ(\mu\circ\alpha\texttt{<>})$ of a (theoretical) indirect access via the receiver $\mathbf{r}$ through call-link $\mathbf{s} \xrightarrow{\mu_\mathbf{r}} \mathbf{r}$ and its result handle $\mathbf{r} \xrightarrow{\mu} \mathbf{o}$ (cf. paragraph *9*). But also the access to the returned object's *α-parameters* should not exceed the access via the receiver. The new condition on result modes in JaM ensures this:
   There is access to the result's α-parameters if the returned handle's mode $\mu_\mathbf{r}\circ\mu$ is a mode $m\texttt{<...,}\alpha\texttt{=}\mu'\texttt{,...>}$ where μ' is a non-`read` mode (condition 1), and m is not `read` (condition 3), i.e., where m is `free` or `rep` (proper `co`- and association modes cannot have correlations). If the result's mode μ is a `co`- or association mode then $\mathbf{s} \xrightarrow{\mu_\mathbf{r}} \mathbf{r} \xrightarrow{\mu} \mathbf{o}$ was for $\mathbf{s}$ the wanted potential access path of mode $\mu_\mathbf{r}\circ\mu$ to $\mathbf{o}$ and its α-parameters. Otherwise, μ was a mode $\texttt{free<...,}\alpha\texttt{=}\mu''\texttt{,...>}$ with $\mu_\mathbf{r}\circ\mu'' = \mu' \neq \texttt{read<...>}$, so that μ'' can neither be `read` nor `rep`. In the case that μ'' is a `free` mode, the sender was theoretically able to pass its (also `free`) $\mu_\mathbf{r}\circ\mu''$ handles to the receiver as `free` μ''-parameters, which could pass them to $\mathbf{o}$ as α-parameters. But in the case that μ'' is an *association* mode $\beta\texttt{<>}$, the sender could pass its $\mu_\mathbf{r}\circ\mu''$ handles as the receiver's μ''-parameters *only* if the call-link is *not* `read` (condition 3).

The return of handles of modes with correlations means the virtual return of the potential access paths starting with that handle (see paragraph *9*). The same way, the supply of parameter handles with correlations means the virtual supply of potential access paths starting with that handle. To these potential access paths the same considerations apply as the exchanged handle itself (which is just the base-case of a potential access path). That is, the discussed conditions on parameter and result modes w.r.t. their and their imports' base-modes must be generalized to conditions on all potential extensions of the exchanged handle. Consequently, the definition of handle signatures (fig. 6.9) rephrases the above conditions over modes $\tilde{\mu}$ (with correlations $\gamma{=}\tilde{\mu}'$) as conditions over mappings $\tilde{\mu}(\vec{\gamma})$ (and $\tilde{\mu}(\vec{\gamma}.\gamma)$).

The overall consequences of these restrictions on the flow of handles in the object system will be considered in §7.1.3.

6.3 Structural Integrity of Object Ownership

This section develops the proof for the Unique Owner and Unique Head invariants in JaM. In subsection 6.3.1, we will first consider proving the ownership theorem for JaM, like its base-JaM-counterpart (Theorem 2), by simple induction on the number N of reduction steps: $\mathfrak{g}_0 \models \mathsf{UH}, \mathsf{UO}$, and in each possible reduction step from object graph $\mathfrak{g}$ to $\mathfrak{g}'$, $\mathfrak{g} \models \mathsf{UH}, \mathsf{UO}$ implies $\mathfrak{g}' \models \mathsf{UH}, \mathsf{UO}$. However, when the case of reductions with {call} is reached, it will turn out that the new potential access paths after supply of a handle parameter may connect previously only very indirectly related objects, objects between which no single *forward* paths of edges existed in the (extended) graph before. To prove the UO- and UH-consistency of two ownership paths with the same target in $\mathfrak{g}'$, a much stronger property will be needed in $\mathfrak{g}$, an UO- and UH-like consistency between ownership paths to very indirectly related objects. This indirect relation can be seen as a "reservation" for ownership. Subsection 6.3.3 will investigate the change of ownership reservations during execution, and subsection 6.3.5 will prove the consistency. Unique Head and Unique Owner are then followed from it as the conclusion of this section (§6.3.6).

6.3.1 Change at the Level of Potential Access Paths

This subsection collects results on the changes to potential access paths effected by the addition of new edges to the object graph through reductions with {new}, with {upd} (because of potential implicit mode conversions), with {ret}, and with {call}.

1. TALKING ABOUT POTENTIAL EXTENSION. In JaM, reasoning about the structure of object ownership, or even describing the new ownership paths after a change to the object graph, is much more complex than in base-JaM since potential access paths of mode μ are not just a μ-edge followed by co-edges any more: In base-JaM, whenever a new co-edge $o \xrightarrow{\mathsf{co}} o'$ appeared in the extended object graph $\mathfrak{g}^\star$ (by whatever operation), the only new ownership paths this entailed were from o's old owner to o' and all its co-objects ω. In full JaM, a new co-edge $o \xrightarrow{\mathsf{co<>}} o'$ may entail new ownership paths $\pi = \pi_1 \bullet o \xrightarrow{\mathsf{co<>}} o' \bullet \pi_2 \in PAP(u, \mu, \omega)$ to all objects ω that can be reached from o' via association path sequences $o'' \overset{\vec{\alpha}}{\dashrightarrow} \omega$ from o' and its co-objects o'' (i.e., $\pi_2 = o' \xrightarrow{\mathtt{co<>}}{}^{*}\, o'' \overset{\vec{\alpha}}{\dashrightarrow} \omega$). This is possible whenever any potential direct extensions of the first segment π_1 to o by $\vec{\alpha}$-association path sequences were already potential access paths of mode μ.

Stating this in a more formal way is not as straight forward as it might seem: First, we are talking not about actual extensions but about potential extensions—the new ownership path π is possible no matter whether any association path sequence $o \overset{\vec{\alpha}}{\dashrightarrow} \omega'$ actually exists in the object graph. Second, the first segment π_1, whose extension we are considering, is not necessarily a potential access path. For example, in the map example, we have the path $\pi_1 = \mathsf{d} \xrightarrow{\mathtt{rep<elem=rep<...>>}} \mathsf{s} \xrightarrow{\mathtt{rep<data=elem<>>}} \mathsf{n}$ from `MapImp` object `d` to `PNode` object `n1`. It is not a potential access path; but any `data`-path which `n` had, would extend it to a `rep` path. Even if `n` actually has no `data`-path, when it obtains a co-edge $\mathsf{n} \xrightarrow{\mathtt{co<>}} \mathsf{n}'$ to another node with a `data`-handle $\pi_2 = \mathsf{n}' \xrightarrow{\mathtt{data<>}} \mathsf{e}'$, then $\pi_1 \bullet \mathsf{n} \xrightarrow{\mathtt{co<>}} \mathsf{n}' \bullet \pi_2$ will be an ownership path from `d` to e'.

A formal trick to handle a potential for extension is to guarantee that there is always at least one $\vec{\alpha}$-association path sequence by adding dummy association handles $o \xrightarrow{\vec{\alpha}} o.\vec{\alpha}$ to new dummy objects $o.\vec{\alpha}$ to the object graph $\mathfrak{g}$. In such an extended graph $\mathfrak{g}^{\circledast}$, the potential extensions become one actual extension, so that we can then simply say: There are new ownership paths $\pi = \pi_1 \bullet o \xrightarrow{\mathtt{co<>}} o' \xrightarrow{\mathtt{co<>}}{}^{*}\, o'' \overset{\vec{\alpha}}{\dashrightarrow} \omega \in PAP_{\mathfrak{g}}(u, \mu, \omega)$ iff $\pi' = \pi_1 \bullet o \xrightarrow{\vec{\alpha}} o.\vec{\alpha}$ is a potential access path $\pi' = PAP_{\mathfrak{g}^{\circledast}}(u, \mu, o.\vec{\alpha})$.

A similar problem is the description of the consequences of a new association handle $o \xrightarrow{\beta<>} o'$ in the object graph. Also here, dummy association handles will help: There are new ownership paths $\pi = \pi_1 \bullet o \xrightarrow{\beta<>} o' \xrightarrow{\mathtt{co<>}}{}^{*}\, o'' \overset{\vec{\alpha}}{\dashrightarrow} \omega \in PAP_{\mathfrak{g}}(u, \mu, \omega)$ iff $\pi' = \pi_1 \bullet o \xrightarrow{\beta.\vec{\alpha}} o.\beta.\vec{\alpha}$ is a potential access path $\pi' = PAP_{\mathfrak{g}^{\circledast}}(u, \mu, o.\beta.\vec{\alpha})$.

Definition 12 The notion of "objects reachable from o via possible $\vec{\alpha}$-sequence of association paths" shall be called ***o's $\vec{\alpha}$-region***. For each object $o \in \mathbb{O}$ and possible association paths with the sequence $\vec{\alpha} \in \mathbb{A}^*$ of association roles, i.e., for each region, let the corresponding dummy object be $o.\vec{\alpha} \in \mathbb{O}_{\mathbf{r}} =_{\mathrm{df}} \mathbb{O} \times \mathbb{A}^* = \mathbb{O} \cup \mathbb{O} \times \mathbb{A} \cup \mathbb{O} \times \mathbb{A} \times \mathbb{A} \cup \ldots$, and call it the ***region object***. Since $\mathbb{O} \times \mathbb{A}^* = \mathbb{O} \times (\mathbb{A}^* \times \mathbb{A}^*) = (\mathbb{O} \times \mathbb{A}^*) \times \mathbb{A}^*$, formally also region objects $o = q.\vec{\gamma}$ have their region objects $o.\vec{\alpha}$ but these are simply the region objects of q's $\vec{\gamma} \bullet \vec{\alpha}$-region: $o.\vec{\alpha} = (q.\vec{\gamma}).\vec{\alpha} = q.(\vec{\gamma} \bullet \vec{\alpha})$. The extension $\mathfrak{g}^{\circledast}$ of graph $\mathfrak{g} \in \mathfrak{Graph}$ by dummy edges is $\mathfrak{g}^{\circledast} =_{\mathrm{df}} \mathfrak{g} \cup \{o \xrightarrow{\alpha<>} o.\alpha \mid o \in \mathbb{O}_{\mathbf{r}}, \alpha \in \mathbb{A}\}$. Extended graphs $\mathfrak{g}^{\circledast}$ are multisets of mode-labeled edges between region object: $\mathfrak{g}^{\circledast} \in \mathfrak{Graph}_{\mathbf{r}} =_{\mathrm{df}} \mathbf{N}^{\mathbb{O}_{\mathbf{r}} \times \mathcal{M} \times \mathbb{O}_{\mathbf{r}}}$

We will write simply $q.\vec{\gamma}.\vec{\alpha}$ for indirect region objects $(q.\vec{\gamma}).\vec{\alpha} = q.(\vec{\gamma} \bullet \vec{\alpha})$. Region objects include the special case of $o.\epsilon = o \in \mathbb{O} \subseteq \mathbb{O}_{\mathbf{r}}$. The dummy objects are the proper region objects $o \in \mathbb{O} \times \mathbb{A}^+ = \mathbb{O}_{\mathbf{r}} \setminus \mathbb{O}$, i.e., region objects $o = o'.\vec{\alpha}$ with $\vec{\alpha} \neq \epsilon$. In the following, the term "object" will normally apply both for dummy objects and proper objects (in other words, for region objects), unless noted otherwise. In particular, meta variables o, ω, q, etc. for objects normally range over all of $\mathbb{O}_{\mathbf{r}}$.

Since the edges added in $\mathfrak{g}^{\circledast}$ lead only to region objects and do not connect proper objects, there is a potential access path between proper objects in $\mathfrak{g}^{\circledast}$ if and only if there is a potential access path in $\mathfrak{g}$.[4] The move from $\mathfrak{g}$ to $\mathfrak{g}^{\circledast}$ in reasoning will not affect ownership and representations between proper objects.

2. {NEW}. The creation of a new object $\mathbf{o}$ by the evaluation of object creation expression `new<δ>` c`()` in the execution of a method called on (creator) object $\mathbf{r}$ adds an edge $\mathbf{r} \xrightarrow{\mathtt{free<}\delta\mathtt{>}} \mathbf{o}$ to a *fresh* object $\mathbf{o}$. "Fresh" implies that $\mathbf{o}$ is not the currently active object $\mathbf{r}$, and neither target nor source of any edge in $\mathfrak{g}$.

Lemma 6 Consider the addition of a `free` edge $h_{\mathbf{o}} = \mathbf{r} \xrightarrow{\mu_{\mathbf{o}}} \mathbf{o}$ to fresh object $\mathbf{o}$ such that $\mathfrak{g}' = \mathfrak{g} \oplus \mathbf{r} \xrightarrow{\mu_{\mathbf{o}}} \mathbf{o}$. In $\mathfrak{g}'^{\circledast}$, there are two kinds of new potential access paths $\pi \in PAP_{\mathfrak{g}'^{\circledast}}(o, \mu, \omega)$:

1. *Initially new paths* go from $o = \mathbf{r}$ to $\omega = \mathbf{o}.\vec{\alpha}$. They are extensions $\pi_{\vec{\alpha}} = \mathbf{r} \xrightarrow{\mu_{\mathbf{o}}} \mathbf{o} \xrightarrow{\vec{\alpha}} \mathbf{o}.\vec{\alpha}$ of $h_{\mathbf{o}}$ by dummy edges for every $\vec{\alpha} \in \mathbb{A}^*$ with $\mu_{\mathbf{o}}(\vec{\alpha}) \neq \bot$ If it is an association path, i.e., $\mu_{\mathbf{o}}(\vec{\alpha}) = \beta \in \mathbb{A}$ then it has an unchanged witness $\sigma(\pi_{\vec{\alpha}}) = \mathbf{r} \xrightarrow{\beta} \mathbf{r}.\beta$.
2. *Internally new paths* are concatenations $\pi_1 \bullet \pi_{\vec{\alpha}.\vec{\alpha}'} = \pi_1 \bullet \pi_{\vec{\alpha}} \bullet \pi_3$ of a non-trivial unchanged path π_1 of edges from o to $\mathbf{r}$, an initially new path $\pi_{\vec{\alpha}}$ of some association mode β<>, i.e., $\mu_{\mathbf{o}}(\vec{\alpha}) = \beta$, and a possibly empty sequence $\pi_3 = \mathbf{o}.\vec{\alpha} \xrightarrow{\vec{\alpha}'} \mathbf{o}.\vec{\alpha}.\vec{\alpha}'$ of further dummy edges. π has an unchanged witness $\sigma(\pi) = \pi_1 \bullet \sigma(\pi_{\vec{\alpha}}) \bullet \mathbf{r}.\beta \xrightarrow{\vec{\alpha}'} \mathbf{r}.\beta.\vec{\alpha}' = \pi_1 \bullet \mathbf{r} \xrightarrow{\beta.\vec{\alpha}'} \mathbf{r}.\beta.\vec{\alpha}' \in PAP_{\mathfrak{g}'^{\circledast}}(o, \mu, \mathbf{r}.\beta.\vec{\alpha}')$ of the same shape.

Since all new potential access paths in $\mathfrak{g}'^{\circledast}$ target the fresh object $\mathbf{o}$ and its region objects, and since these cannot be the target of ownership paths in $\mathfrak{g}^{\circledast}$, an UO- or UH-conflict could exist in $\mathfrak{g}'^{\circledast}$ only among two new ownership paths: There can be none among two initially new ownership paths $\pi_{\vec{\alpha}}$ and $\pi_{\vec{\alpha}'}$ with the same target $\mathbf{o}.\vec{\alpha} = \mathbf{o}.\vec{\alpha}'$, since this implies that they are identical. There can be none among two internally new ownership paths $\pi_1 \bullet \pi_{\vec{\alpha}.\vec{\alpha}'}$ and $\pi_1' \bullet \pi_{\vec{\alpha}.\vec{\alpha}'}$ with the same target $\mathbf{o}.\vec{\alpha}.\vec{\alpha}'$, since then their witnesses have the same, uniquely determined target $\sigma(\mathbf{o}.\vec{\alpha}.\vec{\alpha}') = \mathbf{r}.\mu_{\mathbf{o}}(\vec{\alpha}).\vec{\alpha}'$, and thus would have been in conflict in $\mathfrak{g}^{\circledast}$. And there can be none between initially new ownership paths $\pi_{\vec{\alpha}}$ and internally new ownership paths $\pi_1 \bullet \pi_{\vec{\alpha}''.\vec{\alpha}'}$ since target $\mathbf{o}.\vec{\alpha}$ cannot be the same as target $\mathbf{o}.\vec{\alpha}''.\vec{\alpha}'$: The initially new path implies $\mu_{\mathbf{o}}(\vec{\alpha}) \in \{\mathtt{free}, \mathtt{rep}\}$ while the internally new path implies $\mu_{\mathbf{o}}(\vec{\alpha}'') \in \mathbb{A}$, so that the nesting constraints on association modes prevents $\mu_{\mathbf{o}}(\vec{\alpha}''.\vec{\alpha}') \in \{\mathtt{free}, \mathtt{rep}\}$.

Proof of the lemma: The only added edge $h_{\mathbf{o}}$ in $\mathfrak{g}'^{\circledast}$ targets a *fresh* object $\mathbf{o}$. This means that in $\mathfrak{g}^{\circledast}$, $\mathbf{o}$ was only connected with other objects by dummy edges $\mathbf{o} \xrightarrow{\alpha\texttt{<>}} \mathbf{o}.\alpha$ to its own region objects. In $\mathfrak{g}'^{\circledast}$, $\mathbf{o}$ still has no other edges, since it is not $\mathbf{r}$. And the only edge targeting $\mathbf{o}$ in $\mathfrak{g}'^{\circledast}$ is $h_{\mathbf{o}}$. Hence the only way to extend new handle $h_{\mathbf{o}}$ to a new potential access path is along dummy edges to $\mathbf{o}$'s region objects. Since dummy edges are not co-edges, this extension depends on the correlations $\mu_{\mathbf{o}}$ in $h_{\mathbf{o}}$'s mode: Iff $\mu_{\mathbf{o}}(\vec{\alpha}) \neq \bot$, then $\pi_{\vec{\alpha}} = h_{\mathbf{o}} \bullet \mathbf{o} \xrightarrow{\vec{\alpha}} \mathbf{o}.\vec{\alpha}$ is a potential access path of corresponding

[4] To be precise, this presupposes that the given set $\mathbb{O}$ of object-identifiers contains no dummy objects: $\mathbb{O} \cap \mathbb{O} \times \mathbb{A}^+ = \emptyset$.

mode, an *initially new path.* In particular, $\pi_{\vec{\alpha}}$'s mode is β<> iff $\mu_{\mathbf{o}}(\vec{\alpha}) = \beta$. This includes handle $h_{\mathbf{o}}$ itself as an initially new path with $\vec{\alpha} = \epsilon$.

Nesting-constraints on modes prevent extensions of `free` $h_{\mathbf{o}}$ to initially new paths from being `co`. An initially new path $\pi_{\vec{\alpha}}$ that is an association path of mode β<> can extend any path π_1 to $\mathbf{r}$ with a correlation β=μ to an internally new path $\pi = \pi_1 \bullet \pi_{\vec{\alpha}} = \pi_1 \bullet \pi_{\vec{\alpha}.\epsilon}$ of mode μ. Its witness $\sigma(\pi)$ is $\pi_1 \bullet \mathbf{r} \xrightarrow{\beta<>} \mathbf{r}.\beta$. If an internally new path $\pi = \pi_1 \bullet \pi_{\vec{\alpha}.\vec{\alpha}'}$ has a mode with a correlation β=μ, it is extended further to potential access path $\pi \bullet \mathbf{o}.\vec{\alpha}.\vec{\alpha}' \xrightarrow{\beta<>} \mathbf{o}.\vec{\alpha}.\vec{\alpha}'.\beta = \pi_1 \bullet \pi_{\vec{\alpha}.\vec{\alpha}'.\beta}$. Since it has a witness $\sigma(\pi) \bullet \mathbf{r}.\mu_{\mathbf{o}}(\vec{\alpha}).\vec{\alpha}' \xrightarrow{\beta<>} \mathbf{r}.\mu_{\mathbf{o}}(\vec{\alpha}).\vec{\alpha}'.\beta = \pi_1 \bullet \mathbf{r} \xrightarrow{\mu_{\mathbf{o}}(\vec{\alpha}).\vec{\alpha}'.\beta} \mathbf{r}.\mu_{\mathbf{o}}(\vec{\alpha}).\vec{\alpha}'.\beta$, it is an *internally new path* again. If there is an unchanged potential access path π' that has a correlation β=μ and targets the source of an internally new path π of association mode β<> then it is extended to the potential access path $\pi' \bullet \pi = (\pi' \bullet \pi_1) \bullet \pi_{\vec{\alpha}}$. Since it has a witness $\pi' \bullet \sigma(\pi) = (\pi' \bullet \pi_1) \bullet \mathbf{r} \xrightarrow{\mu_{\mathbf{o}}(\vec{\alpha})} \mathbf{r}.\mu_{\mathbf{o}}(\vec{\alpha})$, it is an *internally new path* again. Other combinations are not possible, since all initially new and internally new paths target $\mathbf{o}$ and region objects of it, and thus cannot target the source $\mathbf{r}$ of initially new potential access paths, nor the source of any non-dummy unchanged edges, and thus not the source of any unchanged or internally new potential access paths. ■

3. {UPD}. The update of a variable (formally, of the store at a location ℓ) by assignment of handle $\langle \mathbf{c}, \tilde{\mu}_0, \mathbf{o} \rangle$ adds a new edge $\mathbf{c} \xrightarrow{\tilde{\mu}} \mathbf{o}$ iff it involves a real conversion of the assigned handle from mode $\tilde{\mu}_0$ to another mode $\tilde{\mu} \neq \tilde{\mu}_0$. We can ignore the destruction $\mathfrak{g}_0 = \mathfrak{g} \ominus \mathbf{c} \xrightarrow{\tilde{\mu}} \mathbf{o}'$ of (the edge corresponding to) the variable's old value and focus on replacing in the resulting graph $\mathfrak{g}_0$ the right-hand handle $\mathbf{c} \xrightarrow{\tilde{\mu}} \mathbf{o}$ by the variable's new value $\mathbf{c} \xrightarrow{\tilde{\mu}_0} \mathbf{o}$. Note that the typeability of the redex $\hat{e}$ following from Theorem 6 and $\models \mathfrak{s}$ following from Theorem 6 guarantees $\mu' \leq_{\mathrm{m}} \mu$. Hence the step from $\mathfrak{g}_0$ to $\mathfrak{g}'$ can be handled by induction on the number k of elementary conversions from $\tilde{\mu}_0$ to $\tilde{\mu}$, i.e., $\tilde{\mu}_0 \leq^1_{\mathrm{m}} \tilde{\mu}_1 \leq^1_{\mathrm{m}} \tilde{\mu}_2 \ldots \tilde{\mu}_{k-1} \leq^1_{\mathrm{m}} \tilde{\mu}_k = \tilde{\mu}$, with corresponding intermediate graphs $\mathfrak{g}_{i+1} = \mathfrak{g}_i \ominus \mathbf{c} \xrightarrow{\tilde{\mu}_i} \mathbf{o} \oplus \mathbf{c} \xrightarrow{\tilde{\mu}_{i+1}} \mathbf{o}$ up to $\mathfrak{g}_k = \mathfrak{g}'$.

Lemma 7 Consider elementary mode conversion $\mathfrak{g}_{i+1} = \mathfrak{g}_i \ominus \mathbf{c} \xrightarrow{\tilde{\mu}_i} \mathbf{o} \oplus \mathbf{c} \xrightarrow{\tilde{\mu}_{i+1}} \mathbf{o}$, i.e., the substitution of an edge $h_{i+1} = \mathbf{c} \xrightarrow{\tilde{\mu}_{i+1}} \mathbf{o}$ for an edge $h_i = \mathbf{c} \xrightarrow{\tilde{\mu}_i} \mathbf{o}$ with $\tilde{\mu}_i \leq^1_{\mathrm{m}} \tilde{\mu}_{i+1}$. All potential access paths $\pi \in PAP_{\mathfrak{g}^{\circledast}_{i+1}}(o, \mu, \omega)$ in $\mathfrak{g}^{\circledast}_{i+1}$ have a precursor $\pi' = \pi[h_i/h_{i+1}] \in PAP_{\mathfrak{g}^{\circledast}_i}(o, \mu', \omega)$ in $\mathfrak{g}^{\circledast}_i$ with the old edge instead of the new one. Two kinds of new potential access paths can be distinguished:

1. *Initially new paths* $\pi \in PAP_{\mathfrak{g}^{\circledast}_{i+1}}(\mathbf{c}, \mu, \omega)$ start with the converted edge $\mathbf{c} \xrightarrow{\tilde{\mu}_{i+1}} \mathbf{o}$ and have a shape $\mathbf{c} \xrightarrow{\tilde{\mu}_{i+1}} \mathbf{o} \xrightarrow{\mathtt{co}}{}^{*} \bullet \dashrightarrow^{\vec{\alpha}} \bullet$. Their precursor $\pi[h_i/h_{i+1}]$ has the same mode μ or a directly compatible mode $\mu' \leq^1_{\mathrm{m}} \mu$, and has, save for the initial edge, the same shape $\mathbf{c} \xrightarrow{\tilde{\mu}_i} \mathbf{o} \xrightarrow{\mathtt{co}}{}^{*} \bullet \dashrightarrow^{\vec{\alpha}} \bullet$.
2. *Internally new paths* $\pi \in PAP_{\mathfrak{g}^{\circledast}_{i+1}}(o, \mu, \omega)$ have a precursor $\pi[h_i/h_{i+1}] \in PAP_{\mathfrak{g}^{\circledast}_i}(o, \mu, \omega)$ of the same mode and shape.

Note that the constraints on mode compatibility and on the nesting of modes ensure that $\tilde{\mu}_{i+1} \gtrsim_{\mathrm{m}} \tilde{\mu}_i$ cannot be `free`. Hence, first, the nesting constraints on modes exclude that the extensions $\pi = \mathbf{c} \xrightarrow{\tilde{\mu}_{i+1}} \mathbf{o} \bullet \ldots$ of the converted edge, i.e., the initially new paths, have a `free` mode. Second, the only (initial) edge whose multiplicity is increased in $\mathfrak{g}^{\circledast}_{i+1}$ is non-`free`. There can be no multiplicity problem for UH. Consequently, neither the internally new paths, with their shape-equivalent precursors, nor those non-`free` initially new paths that have a mode-equivalent precursor, can introduce any new UO- or UH-conflicts. An initially new path $\pi \in PAP_{\mathfrak{g}^{\circledast}_{i+1}}(\mathbf{c}, \mu, \omega)$ with directly compatible precursor $\pi' \in PAP_{\mathfrak{g}^{\circledast}_{i}}(\mathbf{c}, \mu', \omega)$ with $\mu' \leq^1_{\mathrm{m}} \mu$ can be an ownership path only in case of $\mu' = \texttt{free<}\delta\texttt{>} \leq^1_{\mathrm{m}} \texttt{rep<}\delta\texttt{>} = \mu$. But then the precursor is already an ownership path, so that π does not change ownerships and does not affect UO. And since the precursor is `free`, it guarantees that all old ownership paths $\hat{\pi}'$ to ω had the initial edge $\mathbf{c} \xrightarrow{\tilde{\mu}_i} \mathbf{o}$, whose multiplicity was one. Since this multiplicity is reduced by one in $\mathfrak{g}^{\circledast}_{i+1}$, no ownership path with unchanged initial edge (unchanged or internally new path) can have the target ω. $\texttt{rep<}\delta\texttt{>} = \mu$ Since all initially new ownership paths $\hat{\pi}'$ had an ownership path $\hat{\pi}$ as precursor (of the same or compatible mode), $\hat{\pi}$'s shape $\mathbf{c} \xrightarrow{\tilde{\mu}_i} \mathbf{o} \xrightarrow{\texttt{co}}{}^* \bullet \dashrightarrow^{\bar{\alpha}} \bullet$ means for $\hat{\pi}$ that it starts with $\mathbf{c} \xrightarrow{\tilde{\mu}_{i+1}} \mathbf{o}$, i.e., π's initial edge. Hence there is no UH-conflict.

Proof of the lemma: In the base case of potential access paths π in $\mathfrak{g}^{\circledast}_{i+1}$, π is a single edge. It is new only if it is the converted edge $h_{i+1} = \mathbf{c} \xrightarrow{\tilde{\mu}_{i+1}} \mathbf{o}$. This is an *initially new path* $\pi \in PAP_{\mathfrak{g}^{\circledast}_{i+1}}(\mathbf{c}, \tilde{\mu}_{i+1}, \mathbf{o})$ with shape $\mathbf{c} \xrightarrow{\tilde{\mu}_{i+1}} \mathbf{o} \xrightarrow{\texttt{co}}{}^* \bullet \dashrightarrow^{\epsilon} \bullet$ and directly compatible precursor $h_i = \mathbf{c} \xrightarrow{\tilde{\mu}_i} \mathbf{o} = \pi[h_{i+1}/h_i]$.

Larger potential access paths π in $\mathfrak{g}^{\circledast}_{i+1}$ are the extensions $\pi_1 \bullet \pi_2$ of paths $\pi_1 \in PAP_{\mathfrak{g}^{\circledast}_{i+1}}(o, \mu_1, q)$ by a `co`- or α-path $\pi_2 \in PAP_{\mathfrak{g}^{\circledast}_{i+1}}(q, \mu_2, \omega)$. Since there is no mode $\mu_2' \leq^1_{\mathrm{m}} \mu_2$ compatible to `co<>` nor α`<>`, π_2 can only be one of these potential access paths that, by induction hypothesis, have a precursor $\pi_2' \in PAP_{\mathfrak{g}^{\circledast}_{i}}(q, \mu_2, \omega)$.

- If π_1 has a precursor of the same mode then the precursors' combination $\pi_1[h_{i+1}/h_i] \bullet \pi_2[h_{i+1}/h_i]$ is a precursor $\pi[h_{i+1}/h_i]$ for π. π is *unchanged* if π_1 and π_2 are unchanged, it is *initially new* if π_1 if initially new, and otherwise *internally new* since the parallel extension of π_1 and precursor $\pi_1[h_{i+1}/h_i]$ by a `co`- or α-path preserves the relation between their shapes (Lemma 10).
- If π_1 is initially new with a precursor of compatible mode $\mu_1' \leq^1_{\mathrm{m}} \mu_1$ and π_1's shape, then in case of $\mu_2 = \texttt{co<>}$, the precursor's extension $\pi_1[h_{i+1}/h_i] \bullet \pi_2[h_{i+1}/h_i]$ is always possible and also has mode μ_1' and π_1's shape. Since π has mode $\mu = \mu_1$ and π_1's shape, this makes π a *initially new* path.
- And in case of $\mu_2 = \alpha$`<>`, μ_1 must be a mode m'`<`$\ldots, \alpha$=$\mu, \ldots$`>`, so that μ_1' is m`<`$\ldots, \alpha$=$\mu', \ldots$`>`. Hence the combination $\pi_1[h_{i+1}/h_i] \bullet \pi_2[h_{i+1}/h_i]$ of precursors has the mode μ' and a compatible shape. If $\mu_1' \leq^1_{\mathrm{m}} \mu_1'$ was by depth-compatibility in the α-correlation, then $\mu' \leq^1_{\mathrm{m}} \mu$; otherwise $\mu' = \mu$: π is *initially new* again. ■

4. {RET}. The return of a handle $\langle \mathbf{r}, \mu_\mathbf{o}, \mathbf{o}\rangle$ at the end of the execution of a method called through call-link $\mathbf{s} \xrightarrow{\mu_\mathbf{r}} \mathbf{r}$ adds the sender's edge $\mathbf{s} \xrightarrow{\mu_\mathbf{r}\circ\mu_\mathbf{o}} \mathbf{o}$ to the object graph (imported edge) and subtracts the receiver's edge $\mathbf{r} \xrightarrow{\mu_\mathbf{o}} \mathbf{o}$ (exported edge) and the call-link: $\mathfrak{g}'' = \mathfrak{g} \ominus \mathbf{r} \xrightarrow{\mu_\mathbf{o}} \mathbf{o} \ominus \mathbf{s} \xrightarrow{\mu_\mathbf{r}} \mathbf{r} \oplus \mathbf{s} \xrightarrow{\mu_\mathbf{r}\circ\mu_\mathbf{o}} \mathbf{o}$. We can ignore the second substep, the removal $\mathfrak{g}' = \mathfrak{g}'' \ominus \mathfrak{s}(\mathrm{im}(\eta^\star))$ of the handles from the environment's locations.

Observe that there are no `free` *edges* with multiplicity larger than one: The only edges whose multiplicity is increased in $\mathfrak{g}''$ are $\mathbf{s} \xrightarrow{\mu_\mathbf{r}\circ\mu_\mathbf{o}} \mathbf{o}$ and, if $\mu_\mathbf{r}\circ\mu_\mathbf{o} = \texttt{co<>}$, its inverse $\mathbf{o} \xrightarrow{\texttt{co<>}} \mathbf{s}$. There are two cases how $\mu_\mathbf{r}\circ\mu_\mathbf{o}$ can be `free`: If $\mu_\mathbf{o}$ is a `free` mode then assumption $\mathfrak{g}^\circledast \models \mathsf{UH}$ ensures for the exported handle $\mathbf{r} \xrightarrow{\mu_\mathbf{o}} \mathbf{o}$ that in $\mathfrak{g}^\circledast$ there can already be an edge $\mathbf{s} \xrightarrow{\mu_\mathbf{r}\circ\mu_\mathbf{o}} \mathbf{o}$ equal to the to-be imported handle only if it is the exported handle and has multiplicity one. But then the decrease of the exported handle's multiplicity is undone by the increase of the imported handle's multiplicity. The multiplicity remains one. If $\mu_\mathbf{o} = \texttt{co<>}$ then $\mu_\mathbf{r}$ is `free`, so that $\mathbf{s} \xrightarrow{\mu_\mathbf{r}} \mathbf{r} \xrightarrow{\mu_\mathbf{o}} \mathbf{o}$ was a `free` path in $\mathfrak{g}^\circledast$. Hence $\mathfrak{g}^\circledast \models \mathsf{UH}$ ensures that there can be $\mathbf{s} \xrightarrow{\mu_\mathbf{r}\circ\mu_\mathbf{o}} \mathbf{o}$ already in $\mathfrak{g}^\circledast$ only if it equals $\mathbf{s} \xrightarrow{\mu_\mathbf{r}} \mathbf{r}$ and that the multiplicity of $\mathbf{s} \xrightarrow{\mu_\mathbf{r}} \mathbf{r}$ is one. But since the call-link's multiplicity is decreased while that of $\mathbf{s} \xrightarrow{\mu_\mathbf{r}\circ\mu_\mathbf{o}} \mathbf{o}$ is increased, the multiplicity remains one. The case with $\mu_\mathbf{o} = \alpha\texttt{<>}$ and a correlation α`=free<...>` is excluded by the nesting constraint on modes.

Lemma 8 Consider result return $\mathfrak{g}'' = \mathfrak{g} \ominus \mathbf{r} \xrightarrow{\mu_\mathbf{o}} \mathbf{o} \ominus \mathbf{s} \xrightarrow{\mu_\mathbf{r}} \mathbf{r} \oplus \mathbf{s} \xrightarrow{\mu_\mathbf{r}\circ\mu_\mathbf{o}} \mathbf{o}$, i.e., the substitution of the imported edge $h'_\mathbf{o} = \mathbf{s} \xrightarrow{\mu_\mathbf{r}\circ\mu_\mathbf{o}} \mathbf{o}$. for the exported edge $h_\mathbf{o} = \mathbf{r} \xrightarrow{\mu_\mathbf{o}} \mathbf{o}$ and the call-link $h_\mathbf{r} = \mathbf{s} \xrightarrow{\mu_\mathbf{r}} \mathbf{r}$. In $\mathfrak{g}''^\circledast$, there are three kinds of new potential access paths $\pi \in PAP_{\mathfrak{g}''^\circledast}(o, \mu, \omega)$:

1. *Co-closure paths* $\pi \in PAP_{\mathfrak{g}''^\circledast}(o, \texttt{co<>}, \omega)$ are co-paths that exist if $\mu_\mathbf{r}\circ\mu_\mathbf{o}$ is $\texttt{co<>}$. They have a precursor $\pi' \in PAP_{\mathfrak{g}^\circledast}(o, \mu, \omega)$ with the imported handle $\mathbf{s} \xrightarrow{\texttt{co<>}} \mathbf{o}$ replaced by the call-link/exported handle combination $\mathbf{s} \xrightarrow{\texttt{co<>}} \mathbf{r} \xrightarrow{\texttt{co<>}} \mathbf{o}$, and its inverse $\mathbf{s} \xleftarrow{\texttt{co<>}} \mathbf{o}$ replaced by the combination $\mathbf{s} \xleftarrow{\texttt{co<>}} \mathbf{r} \xleftarrow{\texttt{co<>}} \mathbf{o}$ of inverse exported handle and inverse call-link: $\pi' = \pi[h_\mathbf{r} \bullet h_\mathbf{o}/h'_\mathbf{o}, h_\mathbf{o}^{-1} \bullet h_\mathbf{r}^{-1}/h_\mathbf{o}'^{-1}]$.
2. *Internally new paths* $\pi \in PAP_{\mathfrak{g}''^\circledast}(o, \mu, \omega)$ have a precursor $\pi' \in PAP_{\mathfrak{g}^\circledast}(o, \mu, \omega)$ with the same shape and $\pi' = \pi[h_\mathbf{r} \bullet h_\mathbf{o}/h'_\mathbf{o}, h_\mathbf{o}^{-1} \bullet h_\mathbf{r}^{-1}/h_\mathbf{o}'^{-1}]$ in case of $\mu_\mathbf{r}\circ\mu_\mathbf{o} = \texttt{co<>}$, and $\pi' = \pi[h_\mathbf{r} \bullet h_\mathbf{o}/h'_\mathbf{o}]$ otherwise.
3. *Initially new paths* are non-co paths $\pi = h'_\mathbf{o} \bullet \tilde{\pi} \in PAP_{\mathfrak{g}''^\circledast}(\mathbf{s}, \mu, \omega)$ that exist if $\mu_\mathbf{r}\circ\mu_\mathbf{o} \neq \texttt{co<>}$. They start with $h'_\mathbf{o}$, have a shape $\mathbf{s} \xrightarrow{\mu_\mathbf{r}\circ\mu_\mathbf{o}} \mathbf{o} \xrightarrow{\texttt{co}}{}^* \bullet \dashrightarrow^{\vec{\alpha}} \bullet$, and a counterpart $\mathfrak{expo}(\pi) = h_\mathbf{o} \bullet \tilde{\pi}' \in PAP_{\mathfrak{g}^\circledast}(\mathbf{r}, \mu', \omega)$ with a mode μ' such that $\mu = \mu_\mathbf{r}\circ\mu'$. The counterpart starts with $h_\mathbf{o}$ instead of $h'_\mathbf{o}$ and has shape $\mathbf{r} \xrightarrow{\mu_\mathbf{o}} \mathbf{o} \xrightarrow{\texttt{co}}{}^* \bullet \dashrightarrow^{\vec{\alpha}} \bullet$. The two paths' postfixes are related like internally new paths: $\tilde{\pi}' = \tilde{\pi}[h_\mathbf{r} \bullet h_\mathbf{o}/h'_\mathbf{o}]$, and similarly related are π and the combination $h_\mathbf{r} \bullet \mathfrak{expo}(\pi)$ of the call-link and the counterpart: $h_\mathbf{r} \bullet \mathfrak{expo}(\pi) = \pi[h_\mathbf{r} \bullet h_\mathbf{o}/h'_\mathbf{o}]$. It is a potential access path in $PAP_{\mathfrak{g}^\circledast}(\mathbf{s}, \mu, \omega)$ if $\mu_\mathbf{o}(\vec{\alpha}) \in \{\texttt{co}\} \cup \mathbb{A}$, and its shape is $\mathbf{s} \xrightarrow{\mu_\mathbf{r}} \mathbf{r} \xrightarrow{\texttt{co}}{}^* \bullet \dashrightarrow^{\vec{\alpha}} \bullet$ if $\mu_\mathbf{o} = \texttt{co<>}$ or $\mathbf{s} \xrightarrow{\mu_\mathbf{r}} \mathbf{r} \xrightarrow{\texttt{co}}{}^* \bullet \dashrightarrow^{\beta.\vec{\alpha}} \bullet$ if $\mu_\mathbf{o} = \beta\texttt{<>}$.

Co-closure paths and internally new paths are harmless since they are not ownership paths, or have a precursor with the same shape (note that there is no multiplicity problem). Consider an initially new ownership path π of the kind with a precursor π' and another ownership path $\tilde{\pi}'$ with the same target ω that is unchanged, internally new or initially new with a precursor $\tilde{\pi}'$. Then $\mathfrak{g}^{\circledast} \models \mathsf{UO}$ guarantees that π' and $\tilde{\pi}'$ have the same source, and thus so have π and $\tilde{\pi}$. There is no UO-conflict. And if π or $\tilde{\pi}$ were `free`, then $\mathfrak{g}^{\circledast} \models \mathsf{UH}$ would guarantee that precursor $\tilde{\pi}'$ has precursor π''s shape $\mathbf{s} \xrightarrow{\mu_\mathbf{r}} \mathbf{r} \xrightarrow{\mathtt{co}}{}^{*} \bullet \dashrightarrow^{\vec{\alpha}} \bullet$ and the initial edge, the call-link, has multiplicity one. But since the call-link's multiplicity is decreased by one in $\mathfrak{g}''^{\circledast}$, it cannot be the unchanged initial edge of unchanged or internally new $\tilde{\pi}$. And if $\tilde{\pi}$ is initially new like π then they both start with the imported handle. There is no UH-conflict.

Consider an initially new π without precursor, i.e., of shape $\mathbf{s} \xrightarrow{\mu_\mathbf{r} \circ \mu_\mathbf{o}} \mathbf{o} \xrightarrow{\mathtt{co}}{}^{*} \bullet \dashrightarrow^{\vec{\alpha}} \bullet$ with $\mu_\mathbf{o}(\vec{\alpha}) \notin \{\mathtt{co}\} \cup \mathbb{A}$. It can only be an ownership path if it is `free` and if it has a `free` counterpart $\mathfrak{expo}(\pi)$, i.e., $\mu_\mathbf{o}(\vec{\alpha}) = \mathtt{free}$. But then $\mathfrak{g}^{\circledast} \models \mathsf{UH}$ guarantees for all ownership paths $\tilde{\pi}'$ in $\mathfrak{g}^{\circledast}$ sharing π's target that they have $\mathfrak{expo}(\pi)$'s initial edge, i.e., the exported handle $h_\mathbf{o}$, and that $h_\mathbf{o}$'s multiplicity is one. Since its multiplicity is reduced by one in $\mathfrak{g}''^{\circledast}$, no unchanged or internally new path $\tilde{\pi}$ can have π's target. If an initially new ownership path $\tilde{\pi}$ with precursor $\tilde{\pi}'$ has π's target, then ownership path $\tilde{\pi}'$'s initial edge $\mathbf{s} \xrightarrow{\mu_\mathbf{r}} \mathbf{r}$ must be the same as $\mathfrak{expo}(\pi)$'s initial edge $\mathbf{r} \xrightarrow{\mu_\mathbf{o}} \mathbf{o}$. But then $\tilde{\pi}$ and π start with the same edge. If an initially new path $\tilde{\pi}$ without precursor has π's target, then its counterpart $\mathfrak{expo}(\tilde{\pi})$ is an ownership path. Hence it must have $\mathfrak{expo}(\pi)$'s initial edge $\mathbf{r} \xrightarrow{\mu_\mathbf{o}} \mathbf{o}$, so that $\tilde{\pi}$ and π start with the same edge. There is no UO- and no UH-conflict with initially new π.

Proof of the lemma: For uniformity, let σ be the substitution $[h_\mathbf{r} \bullet h_\mathbf{o}/h'_\mathbf{o}, h_\mathbf{o}^{-1} \bullet h_\mathbf{r}^{-1}/h_\mathbf{o}'^{-1}]$ in case of $\mu_\mathbf{r} \circ \mu_\mathbf{o} = \mathtt{co<>}$ and $[h_\mathbf{r} \bullet h_\mathbf{o}/h'_\mathbf{o}]$ otherwise. In the base case of potential access paths π in $\mathfrak{g}''^{\circledast}$, π is a single edge. It is new only if it is the imported handle $\pi_\mathbf{o} = \mathbf{s} \xrightarrow{\mu_\mathbf{r} \circ \mu_\mathbf{o}} \mathbf{o}$, or its inverse $\pi_\mathbf{o}^{-1} = \mathbf{o} \xrightarrow{\mathtt{co<>}} \mathbf{s}$ should it be $\mu_\mathbf{r} \circ \mu_\mathbf{o} = \mathtt{co<>}$. If $\mu_\mathbf{r} \circ \mu_\mathbf{o} = \mathtt{co<>}$ then $\mu_\mathbf{r} = \mu_\mathbf{o} = \mathtt{co<>}$. Hence in $\mathfrak{g}^{\circledast}$, call-link and exported handle combine to the co-path $\mathbf{s} \xrightarrow{\mathtt{co<>}} \mathbf{r} \xrightarrow{\mathtt{co<>}} \mathbf{o}$, and they have inverses that combine to the co-path $\mathbf{o} \xrightarrow{\mathtt{co<>}} \mathbf{s} \xrightarrow{\mathtt{co<>}} \mathbf{s}$. These are the necessary precursors $\sigma(\pi_\mathbf{o})$ and $\sigma(\pi_\mathbf{o}^{-1})$ that make $\pi_\mathbf{o}$ and $\pi_\mathbf{o}^{-1}$ *co-closure paths.* If $\mu_\mathbf{r} \circ \mu_\mathbf{o} \neq \mathtt{co<>}$, $\pi_\mathbf{o}$ is a *initially new path* $\pi_\mathbf{o} \in PAP_{\mathfrak{g}''^{\circledast}}(\mathbf{s}, \mu_\mathbf{r} \circ \mu_\mathbf{o}, \mathbf{o})$ and with exported handle $\mathbf{r} \xrightarrow{\mu_\mathbf{o}} \mathbf{o}$ as counterpart $\mathfrak{expo}(\pi_\mathbf{o}) \in PAP_{\mathfrak{g}^{\circledast}}(\mathbf{r}, \mu_\mathbf{o}, \mathbf{o})$ such that $h_\mathbf{r} \bullet \mathfrak{expo}(\pi_\mathbf{o}) = \sigma(\pi)$. If $\mu_\mathbf{o} = \mathtt{co<>}$ then $\mu_\mathbf{r} \circ \mu_\mathbf{o} = \mu_\mathbf{r}$, and call-link and exported handle combined to $\pi_\mathbf{o}$'s precursor $\mathbf{s} \xrightarrow{\mu_\mathbf{r}} \mathbf{r} \xrightarrow{\mu_\mathbf{o}} \mathbf{o}$. If $\mu_\mathbf{o} = \alpha\mathtt{<>}$, then $\mu_\mathbf{r} \circ \mu_\mathbf{o}$ presupposes a correlation $\alpha{=}\mu$ in $\mu_\mathbf{r}$. But then call-link and exported handle already combined in $\mathfrak{g}^{\circledast}$ to the potential access path $\mathbf{s} \xrightarrow{\mu_\mathbf{r}} \mathbf{r} \xrightarrow{\mu_\mathbf{o}} \mathbf{o}$ of mode $\mu_\mathbf{r} \circ \alpha\mathtt{<>} = \mu_\mathbf{r} \circ \mu_\mathbf{o} = \mu$.

Larger potential access paths π in $\mathfrak{g}''^{\circledast}$ are the result of extending paths $\pi_1 \in PAP_{\mathfrak{g}''^{\circledast}}(o, \mu_1, q)$ by a co- or α-path $\pi_2 \in PAP_{\mathfrak{g}''^{\circledast}}(q, \mu_2, \omega)$, which always has a precursor $\sigma(\pi_2)$.

- If π_1 is unchanged of internally new then its shape-equivalent precursor $\sigma(\pi_1)$ is extended by precursor $\sigma(\pi_2)$ to a precursor $\sigma(\pi) = \sigma(\pi_1) \bullet \sigma(\pi_2)$ of $\pi = \pi_1 \bullet \pi_2$ with the same mode and shape. If π_1 and π_2 are unchanged, π is *unchanged.* Otherwise, π is *internally new.*

- If π_1 is a co-closure-path then its mode co<> has no correlations and thus allows only for extension by a co-path π_2. Hence π's mode is π_1's mode, and the precursor's combination $\sigma(\pi_1) \bullet \sigma(\pi_2)$ is possible and a precursor $\sigma(\pi)$ for π. π is again a *co-closure path.*
- If π_1 is initially new then its counterpart $\mathfrak{expo}(\pi_1)$ is extended by $\sigma(\pi_2)$ to a counterpart $\mathfrak{expo}(\pi) = \mathfrak{expo}(\pi_1) \bullet \sigma(\pi_2)$ with $h_\mathbf{r} \bullet \mathfrak{expo}(\pi) = h_\mathbf{r} \bullet \mathfrak{expo}(\pi_1) \bullet \sigma(\pi_2) = \sigma(\pi) \bullet \sigma(\pi_2) = \sigma(\pi)$. If π_1's shape is $\mathbf{s} \xrightarrow{\mu_\mathbf{r} \circ \mu_\mathbf{o}} \mathbf{o} \xrightarrow{\mathsf{co}}{}^* \bullet \dashrightarrow^{\vec{\alpha}} \bullet$ and $\mathfrak{expo}(\pi_1)$'s shape is $\mathbf{r} \xrightarrow{\mu_\mathbf{o}} \mathbf{o} \xrightarrow{\mathsf{co}}{}^* \bullet \dashrightarrow^{\vec{\alpha}} \bullet$ then their extensions by π_2 and $\sigma(\pi_2)$ to π and $\mathfrak{expo}(\pi)$ have the following shapes by Lemma 10: $\mathbf{s} \xrightarrow{\mu_\mathbf{r} \circ \mu_\mathbf{o}} \mathbf{o} \xrightarrow{\mathsf{co}}{}^* \bullet \dashrightarrow^{\vec{\alpha}} \bullet$ and $\mathbf{r} \xrightarrow{\mu_\mathbf{o}} \mathbf{o} \xrightarrow{\mathsf{co}}{}^* \bullet \dashrightarrow^{\vec{\alpha}} \bullet$ in case of $\mu_2 =$ co<>, and $\mathbf{s} \xrightarrow{\mu_\mathbf{r} \circ \mu_\mathbf{o}} \mathbf{o} \xrightarrow{\mathsf{co}}{}^* \bullet \dashrightarrow^{\vec{\alpha}.\beta} \bullet$ and $\mathbf{r} \xrightarrow{\mu_\mathbf{o}} \mathbf{o} \xrightarrow{\mathsf{co}}{}^* \bullet \dashrightarrow^{\vec{\alpha}.\beta} \bullet$ in case of $\mu_2 = \beta$<>. Moreover, if $\mu_\mathbf{o}(\vec{\alpha}.\beta) = \gamma$, then $\mathfrak{expo}(\pi)$ has mode γ<> and $\mu_\mathbf{r} \circ \mu_\mathbf{o}$ implies $\mu_\mathbf{r} \circ \gamma$<>. This means that $\mu_\mathbf{r}$ must have an α-correlation α=μ to π's mode μ. Hence the concatenation $\mathbf{s} \xrightarrow{\mu_\mathbf{r}} \mathbf{r} \bullet \mathfrak{expo}(\pi)$ is a potential access path that is a precursor for π. π is an *initially new path.* ■

5. {CALL}, THE PROBLEM. The invocation of a method through call-link $\mathbf{s} \xrightarrow{\mu_\mathbf{r}} \mathbf{r}$ adds the receiver's self-edge $\mathbf{r} \xrightarrow{\mathsf{co<>}} \mathbf{r}$, and adds one edge $\mathbf{r} \xrightarrow{\mu_j} \mathbf{o}_j$ in the receiver for each non-nil-handle argument $\langle \mathbf{s}, \mu'_j, \mathbf{o}_j \rangle$ in the sender. The self-edge is harmless since it allows merely for the construction of new ownership paths containing superfluous reflexive co-edges. The typeability of reachable redices $\hat{e}$ implied by type consistency (Theorem 6) ensures that, if the modes of the called method's k parameters are $\mu_1, \dots, \mu_k$, then the modes $\mu'_1, \dots, \mu'_k$ of the k handle arguments are compatible to $\mu_\mathbf{r} \circ \mu_1, \dots, \mu_\mathbf{r} \circ \mu_k$, i.e., $\mu_j \leq_\mathrm{m} \mu_\mathbf{r} \circ \mu_j$. From the case of {upd}, we know that the edges $\mathbf{s} \xrightarrow{\mu'_j} \mathbf{o}_j$ in $\mathfrak{g}$ can be replaced, in an intermediate step, to compatible edges $\mathbf{s} \xrightarrow{\mu_\mathbf{r} \circ \mu_j} \mathbf{o}_j$ without danger for the structure of object ownership. But does the replacement of each converted handle argument, the *sent handle* $\mathbf{s} \xrightarrow{\mu_\mathbf{r} \circ \mu_j} \mathbf{o}_j$, by the corresponding *received handle* $\mathbf{r} \xrightarrow{\mu_j} \mathbf{o}_j$ preserve the structure of object ownership?

Normally, for each received handle, one expects first the *initially new paths* $\pi_{\vec{\alpha}} = \mathbf{r} \xrightarrow{\mu_j} \mathbf{o}_j \dashrightarrow^{\vec{\alpha}} \omega$ that extend the handle to objects ω reachable via $\mathbf{o}_j$ and that have a counterpart $\mathfrak{sent}(\pi_{\vec{\alpha}}) = \mathbf{s} \xrightarrow{\mu_\mathbf{r} \circ \mu_j} \mathbf{o}_j \dashrightarrow^{\vec{\alpha}} \omega$ in the graph before the supply. Furthermore, based on initially new co- or association path $\pi_{\vec{\alpha}}$ there should be *internally new paths* $\pi = \pi' \bullet \pi_{\vec{\alpha}} \bullet q \dashrightarrow^{\vec{\gamma}} \omega$ that target objects reachable before the supply from $\mathbf{s}$ by a path $\mathbf{s} \xrightarrow{\mu_\mathbf{r} \circ \mu_j} \mathbf{o}_j \dashrightarrow^{\vec{\alpha}} q \dashrightarrow^{\vec{\gamma}} \omega$, and that have a witness $\mathfrak{wit}(\pi) = \pi' \bullet \mathbf{s} \xrightarrow{\vec{\gamma}} \mathbf{s}.\vec{\gamma}$ (if $\pi_{\vec{\alpha}}$ is a co-path) or $\mathfrak{wit}(\pi) = \pi' \bullet \mathbf{s} \xrightarrow{\mu_j(\vec{\alpha}).\vec{\gamma}} \mathbf{s}.\mu_j(\vec{\alpha}).\vec{\gamma}$ (if $\pi_{\vec{\alpha}}$ is an association path) in the graph before the supply.

However, with received handles that are, or can be extended to, (initially new) co- or association paths, things are different to everything we have seen before: As opposed to the {upd}- and {ret}-cases, these co- and association paths neither necessarily have a precursor—so that the extension of potential access paths by them may produce an unpredictable number of completely new potential access paths to objects reachable from $\mathbf{o}_j$. Nor do they have targets from which no other objects can be reached, as in the {new}-case—"higher-order new" potential access paths may result from the extension of one new path by another one (if $\mathbf{r}$ is reachable from $\mathbf{o}_j$).

At the lowest level of the object graph, every new potential access path $\pi \in PAP(o, \mu, \omega)$ from o to ω of course has a "precursor" $\pi' = \pi[\mathbf{r} \xleftarrow{\mu_\mathbf{r}} \mathbf{s} \xrightarrow{\mu_\mathbf{r}\circ\mu_j} \mathbf{o}_j / \mathbf{r} \xrightarrow{\mu_j} \mathbf{o}_j]$ where the received handle $\mathbf{r} \xrightarrow{\mu_j} \mathbf{o}_j$ is expanded to the call-link $\mathbf{s} - -\mu_\mathbf{r} - - > \mathbf{r}$ in reverse and the sent handle $\mathbf{s} \xrightarrow{\mu_\mathbf{r}\circ\mu_j} \mathbf{o}_j$. (We ignore the inverse received handle in case of $\mu_j = \texttt{co<>}$.)

$$\begin{tikzcd}
 & & \mathbf{s} \arrow[dl, "\mu_\mathbf{r}"'] \arrow[dr, "\mu_\mathbf{r}\circ\mu_j"] & & & \mathbf{s} \arrow[dl, "\mu_\mathbf{r}"'] \arrow[dr, "\mu_\mathbf{r}\circ\mu_j"] & & & & \mathbf{s} \arrow[dl, "\mu_\mathbf{r}"'] \arrow[dr, "\mu_\mathbf{r}\circ\mu_j"] & & \big\}\ \text{old} \\
o \arrow[r, "*"] & \mathbf{r} \arrow[rr, Rightarrow, "\mu_j"] & & \mathbf{o}_j \arrow[r, "*"] & \mathbf{r} \arrow[rr, Rightarrow, "\mu_j"] & & \mathbf{o}_j \arrow[r, "*"] & \cdots \arrow[r, "*"] & \mathbf{r} \arrow[rr, Rightarrow, "\mu_j"] & & \mathbf{o}_j \arrow[r, "*"] & \omega \\
 & & \underbrace{}_{\text{new}} & & & \underbrace{}_{\text{new}} & & & & \underbrace{}_{\text{new}} & &
\end{tikzcd}$$

But π' is not a directed path of references, is no potential access path. There are also not one or two simple subsequences of it that characterize as potential access paths with modes related to π's mode μ how π's source and target were connected before the parameter supply. Several times the forward path of old edges may be interrupted by a gap between $\mathbf{r}$ and $\mathbf{o}_j$ that can only be bridged by following the call-link $\mathbf{r} \xleftarrow{\mu_\mathbf{r}} \mathbf{s}$ *against its referencing direction* before continuing in forward direction with the sent handle $\mathbf{s} \xrightarrow{\mu_j} \mathbf{o}_j$. For reasoning about new *ownership* paths π based on such a lose kind of connection between o and ω before supply, the properties UO and UH are too weak. A stronger property is needed which allows one to extend the uniqueness of ownership and `free` paths through such connections. This property is like a *reservation* for μ-ownership on ω, a reservation which the supply realizes by collapsing the connection to the potential access path π. The assumption about reserved ownership has to be formulated strongly enough to ensure its own preservation when $\mathbf{s}$'s reservation for a $\mu_\mathbf{r}\circ\mu_j$-path to $\mathbf{o}_j$ or for a $\mu_\mathbf{r}$-edge to $\mathbf{r}$ is realized. Reserved ownership, it must be pointed out, is not a concept of JaM programming, but the name given to a proof-technical concept that is molded to the needs of the proof about the structure of object ownership.

6. Region-Coupling. When shall we say that an object o has a reservation for ownership on object ω, or more generally, for a μ-path to ω?: When there is a potential access path $\pi \in PAP(o, \mu, q.\vec{\gamma})$ to a region object $q.\vec{\gamma}$ and from q to ω there is a corresponding $\vec{\gamma}$-sequence of association paths *modulo region-coupling*, written, $\varphi = q \dashrightarrow^{\vec{\gamma}}\!/_{\rightleftharpoons}\ \omega$! That is, the pair $\langle \pi, \varphi \rangle$ formalizes the ***μ-reservation***. The "modulo region-coupling"-qualification is the crucial part that takes the gaps $u \xleftarrow{\mu} v \xrightarrow{\mu\circ\mu'} w$ into account that could be closed to $u \xrightarrow{\mu'} w$ by the supply of handle $v \xrightarrow{\mu\circ\mu'} w$ to u through call-link $u \xleftarrow{\mu} v$.

The ***region-coupling*** relation $\rightleftharpoons$, defined formally in figure 6.11, is the transitive reflexive symmetric closure of four cases of region-coupling:[5]

[5]No separate rule for reflexivity is necessary: Empty $o \dashrightarrow^{\epsilon}\!/_{\rightleftharpoons}\ o$ entails $o.\vec{\gamma} = o.\epsilon.\vec{\gamma} \rightleftharpoons o.\vec{\gamma}$.

$$\frac{\pi \in PAP_{\mathfrak{g}^{\circledast}}(o,\mu,\omega) \quad \mu(\epsilon) \in \{\texttt{free},\texttt{rep}\} \quad \mu(\beta) \in \mathbb{A}_{\perp}}{o.\mu(\beta) \rightleftharpoons \omega.\beta \ \ \textbf{via}\ \{\pi\}} \qquad \frac{o \overset{\vec{\alpha}}{\dashrightarrow}\!/_{\rightleftharpoons}\, \omega \ \ \textbf{via}\ \Pi}{o.\vec{\alpha}.\vec{\gamma} \rightleftharpoons \omega.\vec{\gamma} \ \ \textbf{via}\ \Pi}$$

$$\frac{o.\vec{\alpha} \rightleftharpoons \omega.\vec{\gamma} \ \ \textbf{via}\ \Pi}{\omega.\vec{\gamma} \rightleftharpoons o.\vec{\alpha} \ \ \textbf{via}\ \Pi} \qquad \frac{o.\vec{\alpha} \rightleftharpoons q.\vec{\beta} \ \ \textbf{via}\ \Pi \quad q.\vec{\beta} \rightleftharpoons \omega.\vec{\gamma} \ \ \textbf{via}\ \Pi'}{o.\vec{\alpha} \rightleftharpoons \omega.\vec{\gamma} \ \ \textbf{via}\ \Pi \cup \Pi'} \qquad \frac{}{o.\perp \rightleftharpoons \omega.\perp \ \ \textbf{via}\ \emptyset}$$

$$\frac{\pi \in PAP_{\mathfrak{g}^{\circledast}}(o,\beta\texttt{<>},\omega)}{o \overset{\beta}{\dashrightarrow}\!/_{\rightleftharpoons}\, \omega \ \ \textbf{via}\ \{\pi\}} \qquad \frac{o.\vec{\alpha} \rightleftharpoons q.\vec{\gamma} \ \ \textbf{via}\ \Pi \quad q \overset{\vec{\gamma}}{\dashrightarrow}\!/_{\rightleftharpoons}\, \omega \ \ \textbf{via}\ \Pi'}{o \overset{\vec{\alpha}}{\dashrightarrow}\!/_{\rightleftharpoons}\, \omega \ \ \textbf{via}\ \Pi \cup \Pi'}$$

$$\frac{}{o \overset{\epsilon}{\dashrightarrow}\!/_{\rightleftharpoons}\, o \ \ \textbf{via}\ \emptyset} \qquad \frac{o \overset{\vec{\alpha}_1}{\dashrightarrow}\!/_{\rightleftharpoons}\, q \ \ \textbf{via}\ \Pi \quad q \overset{\vec{\alpha}_2}{\dashrightarrow}\!/_{\rightleftharpoons}\, \omega \ \ \textbf{via}\ \Pi'}{o \overset{\vec{\alpha}_1 \bullet \vec{\alpha}_2}{\dashrightarrow}\!/_{\rightleftharpoons}\, \omega \ \ \textbf{via}\ \Pi \cup \Pi'}$$

Figure 6.11: Region-coupling, and association paths modulo it

- Through *ownership* paths $\pi \in PAP(o,\mu,\omega)$ with correlations $\beta{=}\alpha\texttt{<>}$, o's α-handles $\langle o,\alpha\texttt{<>},q\rangle$ and ω's β-handles $\langle \omega,\beta\texttt{<>},q\rangle$ can be exchanged in both directions. That is, members q of o's α-region (cf. Definition 12) can become members of ω's β-region, and vice versa. Hence we will say that through π, o's α-region and ω's β-regions are coupled, in symbols $o.\alpha \rightleftharpoons \omega.\beta$.
- If objects q enter or leave the $\vec{\gamma}$-region of an object ω in o's $\vec{\alpha}$-region ($o \overset{\vec{\alpha}}{\dashrightarrow} \omega$) then they also, respectively, enter or leave o's $\vec{\alpha}.\vec{\gamma}$-region. Hence we will say that the two regions are coupled: $o.\vec{\alpha}.\vec{\gamma} \rightleftharpoons \omega.\vec{\gamma}$. To facilitate later proofs, this is generalized from objects ω currently in o's $\vec{\alpha}$-region to objects ω that might become a member through handle exchange since it is reachable from o via an $\vec{\alpha}$-sequence of association paths *modulo region-coupling* ($o \overset{\vec{\alpha}}{\dashrightarrow}\!/_{\rightleftharpoons}\, \omega$).
- For formal reason, if an *ownership* path $\pi \in PAP(o,\mu,\omega)$ does *not* correlate an association role β ($\mu(\beta) = \perp$), then we say that the target's β-region is coupled with the source's "*undefined region*," in symbols $o.\perp \rightleftharpoons \omega.\beta$.
- One "undefined region" is as good as another; they are all mutually coupled: $o.\perp \rightleftharpoons \omega.\perp$.[6]

It is also defined formally in figure 6.11 what it means to be an $\vec{\alpha}$-sequence of ***association paths modulo region-coupling*** from o to ω, or a $\rightleftharpoons$***-path*** $o \overset{\vec{\alpha}}{\dashrightarrow}\!/_{\rightleftharpoons}\, \omega$ for short: If there is an association path $\pi \in PAP_{\mathfrak{g}^{\circledast}}(o,\beta\texttt{<>},\omega)$ then there is a $\rightleftharpoons$-path $o \overset{\beta}{\dashrightarrow}\!/_{\rightleftharpoons}\, \omega$. If there is a $\rightleftharpoons$-path $q \overset{\vec{\gamma}}{\dashrightarrow}\!/_{\rightleftharpoons}\, \omega$ then, modulo the region-coupling $o.\vec{\alpha} \rightleftharpoons q.\vec{\gamma}$, there is also the $\rightleftharpoons$-path $o \overset{\vec{\alpha}}{\dashrightarrow}\!/_{\rightleftharpoons}\, \omega$. And intrinsic to any notion of path, two consecutive $\rightleftharpoons$-paths can be concatenated to another $\rightleftharpoons$-path, and there are empty $\rightleftharpoons$-paths (which are neutral elements in path-concatenation).

For reasoning about coherence in §6.5, judgments about $\rightleftharpoons$-relationships and $\rightleftharpoons$-paths are derived with an annotation "**via** Π" that records the set of potential access

[6]Alternatively, we could work with "the" undefined region $\perp$. But each object having its own undefined region will provide for a more uniform treatment in the proof.

paths on which they are based, the ***path-base***. Equivalence $\Pi \equiv \Pi'$ and inequivalence $\Pi \Subset \Pi'$ between two path-bases means that the set H of the edges in Π's paths is, respectively, the same, or a subset of, the set H of edges in Π''s paths.

Note that all above association role sequences $\vec{\alpha}$, $\vec{\beta}$, and $\vec{\gamma}$ may also be empty, and that all above objects o, ω, and q can be region objects in the extended graph $\mathfrak{g}^{\circledast}$. No distinction between objects' regions and region objects (or region objects' regions) needs to be made: A region-coupling $o.\vec{\alpha} \rightleftharpoons \omega.\vec{\gamma}$ with the $\vec{\gamma}$-region of a region object $\omega = q.\vec{\beta}$ is equivalent to the region-coupling $o.\vec{\alpha} \rightleftharpoons q.\vec{\beta}.\vec{\gamma}$. This follows formally from dummy edges $q \xrightarrow{\vec{\beta}} q.\vec{\beta} = \omega$, via $q \overset{\vec{\beta}}{\dashrightarrow}_{/\rightleftharpoons} \omega$ and $q.\vec{\beta}.\vec{\gamma} \rightleftharpoons \omega.\vec{\gamma}$.

Definition 13 The ***reserved ownership assumption*** means for any two ownership reservations $\langle\tilde{\pi}, \tilde{\varphi}\rangle$ and $\langle\tilde{\pi}', \tilde{\varphi}'\rangle$ on the same object v, i.e., for ownership paths $\tilde{\pi} \in PAP(w, \tilde{\mu}, u.\vec{\beta})$ and $\tilde{\pi}' \in PAP(w', \tilde{\mu}', u'.\vec{\beta}')$, and $\rightleftharpoons$-paths $\tilde{\varphi} = u \overset{\vec{\beta}}{\dashrightarrow}_{/\rightleftharpoons} v$ and $\tilde{\varphi}' = u' \overset{\vec{\beta}'}{\dashrightarrow}_{/\rightleftharpoons} v$, that

a) $w' = w$ and $\tilde{\mu}' = \tilde{\mu}$,and

b) if $\tilde{\mu}$ or $\tilde{\mu}'$ is `free` then $\tilde{\pi}$ and $\tilde{\pi}'$ have the same initial edge, and its multiplicity is one.

Obviously, this assumption is a strengthening of Unique Owner and Unique Head (they follow with $\vec{\beta}' = \vec{\beta} = \epsilon$, so that $u.\vec{\beta} = u = v = u' = u'.\vec{\beta}'$).

7. $\{$CALL$\}$, THE SOLUTION. Return now to the jth supply substep $\mathfrak{g}_j = \mathfrak{g}_{j-1} \oplus \mathbf{r} \xrightarrow{\mu_\mathbf{o}} \mathbf{o} \ominus \mathbf{s} \xrightarrow{\mu_\mathbf{r} \circ \mu_\mathbf{o}} \mathbf{o}$, i.e., the substitution of a received handle $h'_\mathbf{o} = \mathbf{r} \xrightarrow{\mu_\mathbf{o}} \mathbf{o}$ for a sent handle $h_\mathbf{o} = \mathbf{s} \xrightarrow{\mu_\mathbf{r} \circ \mu_\mathbf{o}} \mathbf{o}$ in the presence of a call-link $h_\mathbf{r} = \mathbf{s} \xrightarrow{\mu_\mathbf{r}} \mathbf{r}$. We can now better describe how source and target of the aforementioned initially new paths $\pi_{\vec{\alpha}} = \mathbf{r} \xrightarrow{\mu_\mathbf{o}} \mathbf{o} \overset{\vec{\alpha}}{\dashrightarrow} \omega$ and internally new paths $\pi = \pi' \bullet \pi_{\vec{\alpha}} \bullet q \overset{\vec{\gamma}}{\dashrightarrow} \omega$ were connected before the supply of handle parameters extensible to association paths, i.e., with $\mu_\mathbf{o}(\vec{\alpha}_i) \in \mathbb{A}$ for some $\vec{\alpha}_i$: First, there was, respectively, a *witness* $\mathfrak{wit}(\pi_{\vec{\alpha}}) = \mathbf{s} \xrightarrow{\mu_\mathbf{r} \circ \mu_\mathbf{o}} \mathbf{o} \xrightarrow{\vec{\alpha}} \mathbf{o}.\vec{\alpha}$ or $\mathfrak{wit}(\pi) = \pi' \bullet \mathbf{r} \xrightarrow{\mu_\mathbf{o}(\vec{\alpha}).\vec{\gamma}} \mathbf{r}.\mu_\mathbf{o}(\vec{\alpha}).\vec{\gamma}$. Second, there was an initial $\rightleftharpoons$-path $\mathbf{o} \overset{\vec{\alpha}}{\dashrightarrow}_{/\rightleftharpoons} \mathbf{r}.\mu_\mathbf{o}(\vec{\alpha}_1).\vec{\gamma}_1$ or $\mathbf{r} \overset{\mu_\mathbf{o}(\vec{\alpha}).\vec{\gamma}}{\dashrightarrow}_{/\rightleftharpoons} \mathbf{r}.\mu_\mathbf{o}(\vec{\alpha}_1).\vec{\gamma}_1$, respectively, followed by a *series* $\mathbf{o} \overset{\vec{\alpha}_1.\vec{\gamma}_1}{\dashrightarrow}_{/\rightleftharpoons} \mathbf{r}.\mu_\mathbf{o}(\vec{\alpha}_2).\vec{\gamma}_2, \ldots, \mathbf{o} \overset{\vec{\alpha}_h.\vec{\gamma}_h}{\dashrightarrow}_{/\rightleftharpoons} \omega$ of $\rightleftharpoons$-paths with $\mu_\mathbf{o}(\vec{\alpha}_i) \in \mathbb{A}$. Third, these paths were linked by extensions $\mathbf{s} \xrightarrow{\mu_\mathbf{r}} \mathbf{r} \xrightarrow{\mu_\mathbf{o}(\vec{\alpha}_{i-1}).\vec{\gamma}_{i-1}} \mathbf{r}.\mu_\mathbf{o}(\vec{\alpha}_{i-1}).\vec{\gamma}_{i-1}$ of the call-link, and $\mathbf{s} \xrightarrow{\mu_\mathbf{r} \circ \mu_\mathbf{o}} \mathbf{o} \xrightarrow{\vec{\gamma}_{i-1}} \mathbf{o}.\vec{\alpha}_i.\vec{\gamma}_i$ of the sent handle to *pairs of ownership paths* $\mathbf{r}.\mu_\mathbf{o}(\vec{\alpha}_{i-1}).\vec{\gamma}_{i-1} \overset{\hat{\mu}_i}{\dashleftarrow} \mathbf{s} \overset{\hat{\mu}_i}{\dashrightarrow} \mathbf{o}.\vec{\alpha}_i.\vec{\gamma}_i$ of the same mode. For example, the structure of the connection between source and target of an internally new path $\pi \in PAP(o, \mu, \omega)$ could be depicted as follows:

$$
\begin{array}{cccccccccccc}
 & & & & \mathbf{s} & & & & & \mathbf{s} & & \\
 & & & \mu_\mathbf{r} \swarrow & & \searrow \mu_\mathbf{r}\circ\mu_\mathbf{o} & & & \mu_\mathbf{r} \swarrow & & \searrow \mu_\mathbf{r}\circ\mu_\mathbf{o} & \\
\mathbf{s} \xrightarrow{\mu_\mathbf{r}\circ\mu_\mathbf{o}}{}^{*} & \mathbf{o} & & \mathbf{r} & & \mathbf{o} & & & \mathbf{r} & & \mathbf{o} & \\
\overset{\mu}{\searrow} & \downarrow & \overset{\vec{\alpha}}{\searrow} & \downarrow & & \downarrow & \overset{\vec{\alpha}_1.\vec{\gamma}_1}{\searrow} & \cdots \;\; \overset{\vec{\alpha}_{h-1}.\vec{\gamma}_{h-1}}{\searrow} & \downarrow & & \downarrow & \overset{\vec{\alpha}_h.\vec{\gamma}_h}{\searrow} \\
 & \mathbf{o}.\vec{\alpha} & /\rightleftharpoons & \mathbf{r}.\mu_\mathbf{o}(\vec{\alpha}_1).\vec{\gamma}_1 & & \mathbf{o}.\vec{\alpha}_1.\vec{\gamma}_1 & /\rightleftharpoons \; \mathbf{r}.\mu_\mathbf{o}(\vec{\alpha}_2).\vec{\gamma}_2 & /\rightleftharpoons & \mathbf{r}.\mu_\mathbf{o}(\vec{\alpha}_h).\vec{\gamma}_h & & \mathbf{o}.\vec{\alpha}_h.\vec{\gamma}_h & /\rightleftharpoons \;\; \omega
\end{array}
$$

Each forward ownership path $\mathbf{s} \overset{\hat{\mu}_i}{\dashrightarrow} \mathbf{o}.\vec{\alpha}_i.\vec{\gamma}_i$ and following $\rightleftharpoons$-path together constitute a $\hat{\mu}_i$-ownership reservation of $\mathbf{s}$ on, respectively, $\mathbf{r}.\mu_\mathbf{o}(\vec{\alpha}_{i+1}).\vec{\gamma}_{i+1}$ (for $i < h$) or ω

(for $i = h$). The connection can be described more generally as one initial potential access path to an object q_0's region object $q_0.\vec{\alpha}_0.\vec{\gamma}_0$ followed by a "$\vec{\alpha}_0.\vec{\gamma}_0$-*bridge*" from q_0 to ω defined as follows:

Definition 14 An $\vec{\alpha}$-***bridge*** from o to ω in a supply substep is an initial $\rightleftharpoons$-path $o \overset{\vec{\alpha}}{\dashrightarrow}\!/_{\rightleftharpoons}\ \omega_0$ **via** Π_0 followed by a series of triples $\langle \pi_i', \pi_i, \varphi_i \rangle$ of two ownership paths $\pi_i' = \omega_i \overset{\hat{\mu}_i}{\leftarrow}$ s and $\pi_i = \mathrm{s} \overset{\hat{\mu}_i}{\rightarrow} q_i.\vec{\alpha}_i.\vec{\gamma}_i$, and a $\rightleftharpoons$-path $\varphi_i = q_i \overset{\vec{\alpha}_i.\vec{\gamma}_i}{\dashrightarrow}\!/_{\rightleftharpoons}\ \omega_{i+1}$ **via** Π_i from $i = 1$ to some $n \geq 0$, such that $\omega_{n+1} = \omega$. In case of parameter mode $\mu_{\mathbf{o}} =$ co<>, each $\vec{\alpha}_i$ is ϵ, and ownership paths $\pi_i' = h_i' \bullet \tilde{\pi}_i'$ and $\pi_i = h_i \bullet \tilde{\pi}_i$ each start with $h_i', h_i \in \{h_{\mathbf{r}}, h_{\mathbf{o}}\}$ and have matching shape $\mathrm{s} \xrightarrow{\mu_{\mathbf{o}}} \mathbf{o} \xrightarrow{\mathsf{co}}{}^* \bullet \overset{\vec{\gamma}_i}{\dashrightarrow} \bullet$ or $\mathrm{s} \xrightarrow{\mu_{\mathbf{r}}} \mathbf{r} \xrightarrow{\mathsf{co}}{}^* \bullet \overset{\vec{\gamma}_i}{\dashrightarrow} \bullet$. In case of parameter mode $\mu_{\mathbf{o}}$ with $\mu_{\mathbf{o}}(\vec{\alpha}) \in \mathbb{A}$ for some $\vec{\alpha}$, each $\pi_i = h_i \bullet \tilde{\pi}_i$ starts with $h_i = h_{\mathbf{o}}$ and has shape $\mathrm{s} \xrightarrow{\mu_{\mathbf{r}} \circ \mu_{\mathbf{o}}} \mathbf{o} \xrightarrow{\mathsf{co}}{}^* \bullet \overset{\vec{\alpha}_i.\vec{\gamma}_i}{\dashrightarrow} \bullet$ with $\mu_{\mathbf{o}}(\vec{\alpha}_i) \in \mathbb{A}$ while $\pi_i' = h_i' \bullet \tilde{\pi}_i$ starts with $h_i' = h_{\mathbf{r}}$ and has shape $\mathrm{s} \xrightarrow{\mu_{\mathbf{r}}} \mathbf{r} \xrightarrow{\mathsf{co}}{}^* \bullet \overset{\mu_{\mathbf{o}}(\vec{\alpha}_i).\vec{\gamma}_i}{\dashrightarrow} \bullet$. The bridge's path-base is $\Pi_0 \cup \bigcup_{i=1}^{n} \{\pi_i, \pi_i\} \cup \Pi_i$.

Lemma 9 In $\mathfrak{g}_j^{\circledast}$, there are three kinds of new potential access paths $\pi \in PAP_{\mathfrak{g}_j^{\circledast}}(o, \mu, \omega)$:

1. *Co-closure paths* $\pi \in PAP_{\mathfrak{g}_j^{\circledast}}(o, \mathsf{co<>}, \omega)$ are co-paths that exist if $\mu_{\mathbf{o}}$ is co<>. They connect old co-objects of $\mathbf{r}$ and $\mathbf{o}$. That is, there is a (possibly empty) path of co-edges $o \xrightarrow{\mathsf{co}}{}^* \mathbf{r}$ or $o \xrightarrow{\mathsf{co}}{}^* \mathbf{o}$, and $\omega \xrightarrow{\mathsf{co}}{}^* \mathbf{r}$ or $\omega \xrightarrow{\mathsf{co}}{}^* \mathbf{o}$. If $\mu_{\mathbf{o}} = \mu_{\mathbf{r}} =$ co<> then π has a precursor $\pi' \in PAP_{\mathfrak{g}_{j-1}^{\circledast}}(o, \mu, \omega)$ where $\pi' = \pi[h_{\mathbf{r}}^{-1} \bullet h_{\mathbf{o}} / h_{\mathbf{o}}',\ h_{\mathbf{o}}^{-1} \bullet h_{\mathbf{r}} / h_{\mathbf{o}}'^{-1}]$.
2. *Internally new paths* $\pi \in PAP_{\mathfrak{g}_j^{\circledast}}(o, \mu, \omega)$ are non-co potential access paths that exist if $\mu_{\mathbf{o}}(\vec{\alpha}) \in \{\mathsf{co}\} \cup \mathbb{A}$ for some $\vec{\alpha}$. They start with an unchanged edge h and have a shape $o \xrightarrow{\tilde{\mu}} q \xrightarrow{\mathsf{co}}{}^* \bullet \overset{\vec{\beta}.\vec{\alpha}_0.\vec{\gamma}_0}{\dashrightarrow} \bullet$. In case of $\mu_{\mathbf{o}} = \mu_{\mathbf{r}} =$ co<>, π has a precursor $\pi' \in PAP_{\mathfrak{g}_{j-1}^{\circledast}}(o, \mu, \omega)$ with the same shape and $\pi' = \pi[h_{\mathbf{r}}^{-1} \bullet h_{\mathbf{o}} / h_{\mathbf{o}}',\ h_{\mathbf{o}}^{-1} \bullet h_{\mathbf{r}} / h_{\mathbf{o}}'^{-1}]$ (π is an "internally-only new" path). Otherwise there was a witness $\mathfrak{wit}(\pi) \in PAP_{\mathfrak{g}_{j-1}^{\circledast}}(o, \mu, q_0.\vec{\alpha}_0.\vec{\gamma}_0)$ and a non-trivial $\vec{\alpha}_0.\vec{\gamma}_0$-bridge from q_0 to ω (π is an "internally really new" path). π and $\mathfrak{wit}(\pi)$ have the same shape and have a common, non-trivial prefix π_1: $\pi = \pi_1 \bullet \pi_2$, $\mathfrak{wit}(\pi) = \pi_1 \bullet q_0 \xrightarrow{\vec{\alpha}_0.\vec{\gamma}_0} q_0.\vec{\alpha}_0.\vec{\gamma}_0$. If $\mu_{\mathbf{o}} \neq$ co<> then $q_0 = \mathbf{r}$ and $\vec{\alpha}_0.\vec{\gamma}_0 \neq \epsilon$; if $\mu_{\mathbf{o}} =$ co<> then $q_0 \in \{\mathbf{r}, \mathbf{o}\}$ $\vec{\alpha}_0.\vec{\gamma}_0$ may be empty. The bridge's path-base is a set Π of paths which together with $\Pi_{\mathbf{o}} = \{h_{\mathbf{o}}'\}$ or, in case of $\mu_{\mathbf{o}} =$ co<>, with non-empty $\Pi_{\mathbf{o}} \subseteq \{h_{\mathbf{o}}', h_{\mathbf{o}}'^{-1}\}$ contains all edges of π's second half $\tilde{\pi}_2$, edges $\Pi_\Lambda = \{h_{\mathbf{r}}, h_{\mathbf{o}}\}$ and some dummy edges $\Pi_{\circledast}$: $\{\pi_2\} \cup \Pi_\Lambda \cup \Pi_{\circledast} \equiv \Pi \cup \Pi_{\mathbf{o}}$
3. *Initially new paths* $\pi \in PAP_{\mathfrak{g}_j^{\circledast}}(\mathbf{r}, \mu, \omega)$ are non-co potential access paths $\pi = h_{\mathbf{o}}' \bullet \pi_1 \bullet \pi_2$ that exist if $\mu_{\mathbf{o}} \neq$ co<>. They start with the received handle and have some shape $\mathbf{r} \xrightarrow{\mu_{\mathbf{o}}} \mathbf{o} \xrightarrow{\mathsf{co}}{}^* \bullet \overset{\vec{\alpha}_0.\vec{\gamma}_0}{\dashrightarrow} \bullet$. In $\mathfrak{g}_{j-1}^{\circledast}$, they have a witness $\mathfrak{wit}(\pi) = h_{\mathbf{o}} \bullet \pi_1 \bullet \pi_2' \in PAP_{\mathfrak{g}_{j-1}^{\circledast}}(\mathrm{s}, \mu_{\mathbf{r}} \circ \mu, q_0.\vec{\alpha}_0.\vec{\gamma}_0)$ which starts with sent handle $h_{\mathbf{o}}$ followed by edges π_1 shared with π to q_0 and then dummy edges π_2' to $q_0.\vec{\alpha}_0.\vec{\gamma}_0$, and which has shape $\mathrm{s} \xrightarrow{\mu_{\mathbf{r}} \circ \mu_{\mathbf{o}}} \mathbf{o} \xrightarrow{\mathsf{co}}{}^* \bullet \overset{\vec{\alpha}_0.\vec{\gamma}_0}{\dashrightarrow} \bullet$. It is followed by a $\vec{\alpha}_0.\vec{\gamma}_0$-bridge from $q_0 = \mathbf{r}$ to ω

via Π. In the simple case, the bridge is trivial, $q_0.\vec{\alpha}_0.\vec{\gamma}_0 = q_0 = \omega$, and $\pi_2 = \pi_2' = \epsilon$. Otherwise, π_1 goes to $q_0 = \mathbf{r}$, π_2 starts with $h_{\mathbf{o}}'$ followed by edges from Π and more $h_{\mathbf{o}}'$-edges, while Π contains all edges of π_2 other than $h_{\mathbf{r}}$ and additionally the edges $\Pi_\Lambda = \{h_{\mathbf{r}}, h_{\mathbf{o}}\}$ and some dummy edges $\Pi_\circledast$: $\{\pi_2\} \cup \Pi_\Lambda \cup \Pi_\circledast \equiv \Pi \cup \{h_{\mathbf{o}}'\}$.

Unique Owner and Unique Head are preserved since the reserved ownership assumption guarantees UO- and UH-consistency for each subsegment of the connection realized by new ownership paths π and π' with the same target, namely between

- $\mathfrak{wit}(\pi) = o \overset{\mu}{\dashrightarrow} q_0.\vec{\alpha}_0.\vec{\gamma}_0$ or $\mathfrak{wit}(\pi) = \mathbf{s} \overset{\mu_{\mathbf{r}}\circ\mu}{\dashrightarrow} q_0.\vec{\alpha}_0.\vec{\gamma}_0$, respectively,[7] and $\omega_0 \overset{\hat{\mu}_1}{\dashleftarrow} \mathbf{s}$ connected by $q_0 \overset{\vec{\alpha}_0.\vec{\gamma}_0}{\dashrightarrow}_{/\rightleftharpoons} \omega_0$ (and trivial $\omega_0 \overset{\epsilon}{\dashrightarrow}_{/\rightleftharpoons} \omega_0$);
- every two ownership paths $\mathbf{s} \overset{\hat{\mu}_i}{\dashrightarrow} q_i.\vec{\alpha}_i.\vec{\gamma}_i$ and $\omega_i \overset{\hat{\mu}_{i+1}}{\dashleftarrow} \mathbf{s}$ connected by $q_i \overset{\vec{\alpha}_i.\vec{\gamma}_i}{\dashrightarrow}_{/\rightleftharpoons} \omega_i$;
- the ownership path $\mathbf{s} \overset{\hat{\mu}_j}{\dashrightarrow} q_j.\vec{\alpha}_j.\vec{\gamma}_j$ on π's side and $q_{j'}'.\vec{\alpha}_{j'}'.\vec{\gamma}_{j'}' \overset{\hat{\mu}_{j'}'}{\dashleftarrow} \mathbf{s}$ on π''s side connected by $q_j \overset{\vec{\alpha}_j.\vec{\gamma}_j}{\dashrightarrow}_{/\rightleftharpoons} \omega$ on π's side and $\omega \overset{\vec{\alpha}_{j'}'.\vec{\gamma}_{j'}'}{\dashleftarrow}_{/\rightleftharpoons} \omega_{j'}'$ on π''s side;
- every two ownership paths $\mathbf{s} \overset{\hat{\mu}_{i+1}'}{\dashrightarrow} \omega_i'$ and $q_i'.\vec{\alpha}_i'.\vec{\gamma}_i' \overset{\hat{\mu}_i'}{\dashleftarrow} \mathbf{s}$ connected by $\omega_i' \overset{\vec{\alpha}_i'.\vec{\gamma}_i'}{\dashleftarrow}_{/\rightleftharpoons} q_i'$;
- and between $\mathbf{s} \overset{\hat{\mu}_1'}{\dashrightarrow} \omega_i'$ and $\mathfrak{wit}(\pi') = q_0'.\vec{\alpha}_0'.\vec{\gamma}_0' \overset{\mu'}{\dashleftarrow} o'$ or $\mathfrak{wit}(\pi') = q_0'.\vec{\alpha}_0'.\vec{\gamma}_0' \overset{\mu_{\mathbf{r}}\circ\mu'}{\dashleftarrow} \mathbf{s}$ connected by $\omega_0' \overset{\vec{\alpha}_0'.\vec{\gamma}_0'}{\dashleftarrow}_{/\rightleftharpoons} q_0'$.

Consequently, π's and π''s source must coincide, and if one of them is `free`, they have the same shape and initial edge of multiplicity one. That is, Unique Owner and Unique Head are preserved by new ownership paths.

Proof of the lemma: The proof uses a few technical lemmas supplemented in the next subsection. Let us cut short the special case that the parameter edge is the same as the argument edge, i.e., $\mathbf{r} = \mathbf{s}$ and $\mu_j = \mu_{\mathbf{r}}\circ\mu_j$. Then nothing changed at all: $\mathfrak{g}'^{\circledast} = \mathfrak{g}_0^{\circledast}$.

And in case of $\mu_{\mathbf{o}} = \mu_{\mathbf{r}} = \texttt{co<>}$, the only new edges are the j^{th} received handle $h_{\mathbf{o}}' = \mathbf{r} \xrightarrow{\texttt{co<>}} \mathbf{o}$ and its inverse $h_{\mathbf{o}}'^{-1} = \mathbf{o} \xrightarrow{\texttt{co<>}} \mathbf{r}$. Call-link $h_{\mathbf{r}} = \mathbf{r} \xrightarrow{\texttt{co<>}} \mathbf{s}$, sent handle $h_{\mathbf{o}}' = \mathbf{s} \xrightarrow{\texttt{co<>}} \mathbf{o}$ of mode $\mu_{\mathbf{r}}\circ\mu_{\mathbf{o}} = \texttt{co<>}\circ\texttt{co<>} = \texttt{co<>}$, and their inverses combine to the necessary precursors: $\mathbf{r} \xrightarrow{\texttt{co<>}} \mathbf{s} \xrightarrow{\texttt{co<>}} \mathbf{o} = h_{\mathbf{r}}^{-1} \bullet h_{\mathbf{o}} = \sigma(h_{\mathbf{o}}')$ and $\mathbf{o} \xrightarrow{\texttt{co<>}} \mathbf{s} \xrightarrow{\texttt{co<>}} \mathbf{r} = h_{\mathbf{o}}^{-1} \bullet h_{\mathbf{r}}\sigma(h_{\mathbf{o}}')$, respectively, where $\sigma = [h_{\mathbf{r}}^{-1} \bullet h_{\mathbf{o}}/h_{\mathbf{o}}', h_{\mathbf{o}}^{-1} \bullet h_{\mathbf{r}}/h_{\mathbf{o}}'^{-1}]$. Then whenever a judgment $\mathfrak{g}'^{\circledast} \vdash \pi \in PAP(o,\mu,\omega)$ can be derived in $\mathfrak{g}'^{\circledast}$, is was possible to derive $\mathfrak{g}_0^{\circledast} \vdash \sigma(\pi) \in PAP(o,\mu,\omega)$ based on replacing derivations of $\mathfrak{g}'^{\circledast} \vdash h_{\mathbf{o}}' \in PAP(o,\mu,\omega)$ from $h_{\mathbf{o}}' \in \mathfrak{g}'^{\circledast}$ and of $\mathfrak{g}'^{\circledast} \vdash h_{\mathbf{o}}' \in PAP(o,\mu,\omega)$ from $h_{\mathbf{o}}'^{-1} \in \mathfrak{g}'^{\circledast}$ to derivations of $\mathfrak{g}_0^{\circledast} \vdash \sigma(h_{\mathbf{o}}') \in PAP(o,\mu,\omega)$ and of $\mathfrak{g}_0^{\circledast} \vdash \sigma(h_{\mathbf{o}}'^{-1}) \in PAP(o,\mu,\omega)$. Consequently, all paths $\pi \in PAP_{\mathfrak{g}'^{\circledast}}(o,\mu,\omega)$ have a precursor $\sigma(\pi) \in PAP_{\mathfrak{g}_0^{\circledast}}(o,\mu,\omega)$. If π is new and its

[7]For uniformity, unchanged and internally-only new potential access paths can be treated like a special case of internally really new paths, with their precursor π' as witness $\mathfrak{wit}(\pi) \in PAP_{\mathfrak{g}'^{\circledast}}(o,\mu,\omega.\epsilon)$, and a corresponding trivial connection $\omega \overset{\epsilon}{\dashrightarrow}_{/\rightleftharpoons} \omega$. Note that the witness $\mathfrak{wit}(\pi)$ of an initially new *ownership* path π is an ownership path, namely a `free` one, since `rep`-modes in parameter mode μ_j have been excluded, so that π's mode μ can only be `free`, and consequently $\mathfrak{wit}(\pi)$'s mode $\mu_{\mathbf{r}}\circ\mu$ is `free` too.

mode μ is co<>, it must be a sequence of co-paths containing $h'_{\mathbf{o}}$ or $h'^{-1}_{\mathbf{o}}$. But then precursor $\sigma(\pi)$ ensures that o and ω are old co-objects of $\mathbf{r}$ and $\mathbf{o}$ (and of each other). π is a co-closure path. If π is not co, it is an internally-only new path.

For the remaining cases, proceed by induction on the derivation of judgments $\mathfrak{g}'^{\circledast} \vdash \pi \in PAP(o, \mu, \omega)$. In the base case, π is only one edge. It is new in $\mathfrak{g}'^{\circledast}$ compared to $\mathfrak{g}_0^{\circledast}$ only if it is the j^{th} received handle $h'_{\mathbf{o}} = \mathbf{r} \xrightarrow{\mu_{\mathbf{o}}} \mathbf{o}$ or its inverse $h'^{-1}_{\mathbf{o}} = \mathbf{o} \xrightarrow{\texttt{co<>}} \mathbf{r}$ in case of $\mu_{\mathbf{o}} = \texttt{co<>}$. If $\mu_{\mathbf{o}} = \texttt{co<>}$ then $h'_{\mathbf{o}}$ and $h'^{-1}_{\mathbf{o}}$ are obviously *co-closure paths.* If $\mu_{\mathbf{o}} \neq \texttt{co<>}$ then $h'_{\mathbf{o}}$ is *initially new* with shape $\bullet \rightarrow \bullet \xrightarrow{\texttt{co}}{}^{*} \bullet \overset{\epsilon}{\dashrightarrow} \bullet$, with the sent handle as witness $\mathfrak{wit}(h'_{\mathbf{o}}) = h_{\mathbf{o}} = \mathbf{s} \xrightarrow{\mu_{\mathbf{r}} \circ \mu_{\mathbf{o}}} \mathbf{o}.\epsilon$ of the same shape, and with the trivial $\mathbf{o}.\epsilon \overset{\epsilon}{\dashrightarrow}_{/\rightleftharpoons} \mathbf{o}$ as corresponding ϵ-bridge.

In the induction step, π is the extension $\pi_1 \bullet \pi_2$ of a potential access path $\pi_1 \in PAP_{\mathfrak{g}'^{\circledast}}(o, \mu_1, q)$ by a co- or α-path $\pi_2 \in PAP_{\mathfrak{g}'^{\circledast}}(q, \mu_2, \omega)$.

$\boxed{\text{First,}}$ consider the case where π_1 is *unchanged*:

(a) If π_2 is also unchanged then π is *unchanged* again.

(b) If π_2 is a co-closure path then $\mu_2 = \texttt{co<>}$, so that $\mu = \mu_1$. If $\mu_1 = \mu = \texttt{co<>}$ then π is a *co-closure path*: Unchanged π_1 means that o is an old co-object of q, which by co-closure path π_2 is an old co-object of $\mathbf{r}$ or $\mathbf{o}$.
If $\mu_1 \neq \texttt{co<>}$ then π is an *internally really new path* (because the case of $\mu_{\mathbf{r}} = \mu_{\mathbf{o}} = \texttt{co<>}$ was already covered before the induction). Since π_2's source q is an old co-object of $q' = \mathbf{r}$ or $\mathbf{o}$, there must be a possibly empty prefix $\pi_q = q \xrightarrow{\texttt{co}}{}^{*} q'$ of unchanged co-edges in $\pi_2 = \pi_q \bullet \tilde{\pi}_2$. Then extension $\pi_1 \bullet \pi_q$ is the necessary witness $\mathfrak{wit}(\pi) \in PAP_{\mathfrak{g}_0^{\circledast}}(o, \mu, q'.\epsilon)$ with $q' \in \{\mathbf{r}, \mathbf{o}\}$ containing all of π's edges from π_1. A corresponding ϵ-bridge to ω that contains all edges of π_2's postfix $\tilde{\pi}_2$ other than $h'_{\mathbf{o}}$ and $h'^{-1}_{\mathbf{o}}$ consists of the trivial $q'.\epsilon \overset{\epsilon}{\dashrightarrow} q'$ and three triples $\langle h_{q'}, h_{\mathbf{r}} \bullet \pi_{\mathbf{r}}, \varphi_{\mathbf{r}} \rangle$, $\langle h_{\mathbf{r}}, h_{\mathbf{o}} \bullet \pi_{\mathbf{o}}, \varphi_{\mathbf{o}} \rangle$, $\langle h_{\mathbf{o}}, h_{\omega} \bullet \pi_{\omega}, \varphi_{\omega} \rangle$: $h_{q'}$ is $h_{\mathbf{r}}$ if $q' = \mathbf{r}$ and $h_{\mathbf{o}}$ if $q' = \mathbf{o}$; h_{ω} is $h_{\mathbf{r}}$ or $h_{\mathbf{o}}$ which a path π_{ω} of unchanged co-edges can extend to ω (since ω was $\mathbf{r}$'s or $\mathbf{o}$'s co-object); the φ_x are trivial $\rightleftharpoons$-paths $x.\epsilon \overset{\epsilon}{\dashrightarrow} x$; and $\pi_{\mathbf{r}}$ and $\pi_{\mathbf{o}}$ are paths $\mathbf{r} \xrightarrow{\texttt{co}}{}^{*} \mathbf{r}$ and $\mathbf{o} \xrightarrow{\texttt{co}}{}^{*} \mathbf{o}$ of co-edges that include all of $\tilde{\pi}_2$'s co-edges on, respectively, $\mathbf{r}$'s and $\mathbf{o}$'s side and then lead back to their starting point. All the ownership paths in this bridge obviously have the same shape shape $\mathbf{s} \xrightarrow{\mu_{\mathbf{r}}} o \xrightarrow{\texttt{co}}{}^{*} \bullet \overset{\epsilon}{\dashrightarrow} \bullet$. The bridge's path-base is $\Pi = \{h_{q'}, h_{\mathbf{r}} \bullet \pi_{\mathbf{r}}, h_{\mathbf{r}}, h_{\mathbf{o}} \bullet \pi_{\mathbf{o}}, h_{\mathbf{o}}, h_{\omega} \bullet \pi_{\omega}\} \equiv \{h_{\mathbf{r}}, h_{\mathbf{o}}, \pi_{\mathbf{r}}, \pi_{\mathbf{o}}, \pi_{\omega}\}$ such that it contains, besides $h_{\mathbf{r}}$ and $h_{\mathbf{o}}$, all edges of π's postfix $\tilde{\pi}_2$ not common with $\mathfrak{wit}(\pi)$, save $h'_{\mathbf{o}}$ and/or $h'^{-1}_{\mathbf{o}}$: $\{\tilde{\pi}_2\} \cup \{h_{\mathbf{r}}, h_{\mathbf{o}}\} \equiv \Pi \cup \Pi_{\mathbf{o}}$ with non-empty $\Pi_{\mathbf{o}} \subseteq \{h'_{\mathbf{o}}, h'^{-1}_{\mathbf{o}}\}$.

(c) If π_2 is initially new then $\pi_2 = h'_{\mathbf{o}} \bullet \pi_3 \bullet \pi_4$, $q = \mathbf{r}$, $\mu_2 = \beta\texttt{<>}$, and π_2 has a witness $\mathfrak{wit}(\pi_2) = h_{\mathbf{o}} \bullet \pi_3 \bullet \pi'_4$ and a $\vec{\alpha}_0.\vec{\gamma}_0$-bridge from q_0 to ω. π is an *internally really new path* whose witness $\mathfrak{wit}(\pi)$ is the extension $\pi_1 \bullet \mathbf{r} \xrightarrow{\beta\texttt{<>}} \mathbf{r}.\beta$ of π_1. For the corresponding β-bridge from $\mathbf{r}$ to ω, consider the two alternatives guaranteed by Lemma 13:

- There may be a $\rightleftharpoons$-path $\mathbf{r} \overset{\beta}{\dashrightarrow}_{/\rightleftharpoons} \omega_0$ **via** $\Pi_{2,0} \cup \{h_{\mathbf{r}}, \mathfrak{wit}(\pi_2)\}$. Substituting it for the first $\rightleftharpoons$-path $q_0 \overset{\vec{\alpha}_0.\vec{\gamma}_0}{\dashrightarrow}_{/\rightleftharpoons} \omega_0$ **via** $\Pi_{2,0}$ in π_2's $\vec{\alpha}_0.\vec{\gamma}_0$-bridge **via** Π_2 produces a β-bridge to ω **via** $\Pi = \Pi_2 \cup \{h_{\mathbf{r}}, \mathfrak{wit}(\pi_2)\} \equiv \Pi_2 \cup \{h_{\mathbf{r}}, h_{\mathbf{o}}, \pi_3, \pi'_4\}$.

- Otherwise there is an ownership path-pair $\langle h_{\mathbf{r}} \bullet h_\beta, \mathfrak{wit}(\pi_2)\rangle = \mathbf{r}.\beta \xleftarrow{\hat{\mu}} \mathbf{s} \xrightarrow{\hat{\mu}} q_0.\vec{\alpha}_0.\vec{\gamma}_0$ where $h_\beta = \mathbf{r} \xrightarrow{\beta} \mathbf{r}.\beta$. The ownership paths have the right initial edges to connect the dummy edge h_β with π_2's $\vec{\alpha}_0.\vec{\gamma}_0$-bridge from q_0 to ω **via** Π_2 to a β-bridge to ω with path-base $\Pi \equiv \Pi_2 \cup \{h_{\mathbf{r}}, h_\beta, h_{\mathbf{o}}, \pi_3 \bullet \pi_4\}$.

By definition of initially new $\pi_2 = h'_{\mathbf{o}} \bullet \pi_3 \bullet \pi_4$, either $\pi_4 = \epsilon$ and $\Pi_2 = \emptyset$, or $\{\pi_4\} \cup \Pi_\Lambda \cup \Pi_\circledast \equiv \Pi_2 \cup \{h'_{\mathbf{o}}\}$. Hence in the $\rightleftharpoons$-path case, we have $\{\pi_2\} \cup \{h_{\mathbf{r}}, h_{\mathbf{o}}\} \cup (\Pi_\circledast \cup \{\pi'_4\}) \equiv \Pi \cup \{h'_{\mathbf{o}}\}$. And in the ownership path-pair case, $\{\pi_2\} \cup \{h_{\mathbf{r}}, h_{\mathbf{o}}\} \cup (\Pi_\circledast \cup \{\pi'_4, h_\beta\}) \equiv \Pi \cup \{h'_{\mathbf{o}}\}$.

(d) If π_2 internally really new then π is an *internally really new path* again: π_2's old witness $\mathfrak{wit}(\pi_2) \in PAP_{\mathfrak{g}_0^\circledast}(q, \mu_2, q_0.\vec{\alpha}_0.\vec{\gamma}_0)$ is extended by precursor π_1 to a witness $\mathfrak{wit}(\pi) = \pi_1 \bullet \mathfrak{wit}(\pi_2) \in PAP_{\mathfrak{g}_0^\circledast}(o, \mu, q_0.\vec{\alpha}_0.\vec{\gamma}_0)$ for π. The corresponding $\vec{\alpha}_0.\vec{\gamma}_0$-bridge from q_0 to ω **via** $\Pi = \Pi_2$ is guaranteed by π_2. If $\pi_2 = \pi_3 \bullet \pi_4$ and $\mathfrak{wit}(\pi_2) = \pi_3 \bullet \pi'_4$ with maximal common prefix π_3, then $\{\pi_4\} \cup \Pi_\Lambda \cup \Pi_\circledast \equiv \Pi_2 \cup \Pi_{\mathbf{o}}$ is guaranteed. Since π_4 is also the postfix of $\pi = \pi_1 \bullet \pi_2 = (\pi_1 \bullet \pi_3) \bullet \pi_4$ not common with $\mathfrak{wit}(\pi) = \pi_1 \bullet \mathfrak{wit}(\pi_2) = (\pi_1 \bullet \pi_3) \bullet \pi'_4$, and since $\Pi = \Pi_2$, we have $\{\pi_4\} \cup \Pi_\Lambda \cup \Pi_\circledast \equiv \Pi \cup \Pi_{\mathbf{o}}$.

$\boxed{\text{Second,}}$ *co-closure* paths π_1 extend always to another *co-closure path*: Since their mode μ_1 is `co<>`, they can only be extended by `co`-paths, so that no new witness is necessary and initially new and internally new paths π_2 are excluded. π's source o is an old co-object of $\mathbf{r}$ or $\mathbf{o}$ since it is the source of co-closure path π_1. π's target ω is by definition also an old co-object of $\mathbf{r}$ or $\mathbf{o}$ *if* π_2 is a co-closure path. And if π_2 is unchanged then π_2's precursor π'_2 of mode `co<>` means that ω is an old co-object of q, which is an old co-object of $\mathbf{r}$ or $\mathbf{o}$ since it is target of co-closure path π_1.

$\boxed{\text{Third,}}$ if π_1 is *initially new* or *internally really new*, and π_2 is a *`co`-path*, then π_2 can neither be initially new nor internally really new, and π is the same kind of new potential access path: Its witness is the same as that of π_1: $\mathfrak{wit}(\pi) = \mathfrak{wit}(\pi_1)$, except in one case where the bridge is empty (see below). The corresponding $\vec{\alpha}_0.\vec{\gamma}_0$-bridge from q_0 to ω follows from π_1's $\vec{\alpha}_0.\vec{\gamma}_0$-bridge from q_0 to q:

(a) If π_2 is unchanged, consider the last $\rightleftharpoons$-path $q_j \xrightarrow{\vec{\alpha}_j.\vec{\gamma}_j}\!\!/_{\rightleftharpoons}\, \omega_j = q$ **via** $\Pi_{1,j}$ in π_1's $\vec{\alpha}_0.\vec{\gamma}_0$-bridge to q:

- If it is not empty, i.e., $\vec{\alpha}_j.\vec{\gamma}_j \neq \epsilon$ or $q_j \neq \omega_j$, then π_2 can extend it to $q_j \xrightarrow{\vec{\alpha}_j.\vec{\gamma}_j}\!\!/_{\rightleftharpoons}\, \omega$ **via** $\Pi' = \Pi_{1,j} \cup \{\hat{\pi} \bullet \pi_2\}$ for some $\hat{\pi} \in \Pi_{1,j}$ (Lemma 11). By substituting it for $q_j \xrightarrow{\vec{\alpha}_j.\vec{\gamma}_j}\!\!/_{\rightleftharpoons}\, \omega_j = q$ π_1's $\vec{\alpha}_0.\vec{\gamma}_0$-bridge to q is redirected to ω **via** $\Pi \equiv \Pi_1 \cup \{\pi_2\}$.
- If $\vec{\alpha}_j.\vec{\gamma}_j = \epsilon$ and $q_j = \omega_j$, and $j > 0$, then the last $\rightleftharpoons$-path is the trivial $q_j \xrightarrow{\epsilon}\!\!/_{\rightleftharpoons}\, q$ **via** $\emptyset$ and the last ownership path is $\hat{\pi} = \mathbf{s} \xrightarrow{\hat{\mu}_j} q_j.\vec{\alpha}_j.\vec{\gamma}_j = q_j.\epsilon = q$. Extension $\hat{\pi} \bullet \pi_2$ is the ownership path $\mathbf{s} \xrightarrow{\hat{\mu}_j} \omega$. Substituting it for the original ownership path, and substituting $\omega \xrightarrow{\epsilon}\!\!/_{\rightleftharpoons}\, \omega$ **via** $\emptyset$ for $q_j \xrightarrow{\epsilon}\!\!/_{\rightleftharpoons}\, q$ redirects π_1's $\vec{\alpha}_0.\vec{\gamma}_0$-bridge to ω **via** $\Pi \equiv \Pi_1 \cup \{\pi_2\}$.
- If $\vec{\alpha}_j.\vec{\gamma}_j = \epsilon$ and $q_j = \omega_j$, and $j = 0$, then the bridge is trivial and **via** $\Pi_1 = \Pi_{1,j} = \emptyset$. This applies only to the case of initially new π_1. Its witness $\mathfrak{wit}(\pi_1)$

is $\mathbf{s} \xrightarrow{\mu_1} q_j.\vec{\alpha}_j.\vec{\gamma}_j = \omega_j = q$. Then $\mathfrak{wit}(\pi) = \mathfrak{wit}(\pi_1) \bullet \pi_2 = o \xrightarrow{\mu_1} \omega = \omega.\epsilon$. The corresponding ϵ-bridge is the trivial initial $\rightleftharpoons$-path **via** $\Pi = \emptyset$ $\omega \dashrightarrow_{\not\rightleftharpoons}^{\epsilon} \omega$ followed by the empty triple series.

(b) If π_2 is a co-closure path then $\mu_\mathbf{o} = \texttt{co<>}$, so that the case of initially new π_1 does not apply. Co-closure path π_2 means that q as well as ω are old co-objects of $\mathbf{r}$ or $\mathbf{o}$, i.e., there are co-paths $\pi_q = \mathbf{r} \xrightarrow{\texttt{co}}{}^* q$ or $\mathbf{o} \xrightarrow{\texttt{co}}{}^* q$, and $\pi_\omega = \mathbf{r} \xrightarrow{\texttt{co}}{}^* \omega$ or $\mathbf{o} \xrightarrow{\texttt{co}}{}^* \omega$. Since $\mu_\mathbf{o} = \texttt{co<>}$, $\mu_\mathbf{r} \circ \mu_\mathbf{o} = \mu_\mathbf{r}$. Hence $h_\mathbf{r}$ or $h_\mathbf{o}$ combined with π_q or π_ω to potential access paths $\hat{\pi}' \in PAP_{\mathfrak{g}_0^\circledast}(\mathbf{s}, \mu_\mathbf{r}, q)$ and $\hat{\pi} \in PAP_{\mathfrak{g}_0^\circledast}(\mathbf{s}, \mu_\mathbf{r}, \omega)$ of shape $\mathbf{s} \xrightarrow{\mu_\mathbf{r}} \mathbf{r} \xrightarrow{\texttt{co}}{}^* \bullet \dashrightarrow^{\epsilon} \bullet$. Since the case of $\mu_\mathbf{r} = \mu_\mathbf{o} = \texttt{co<>}$ was already handled before the induction, $\mu_\mathbf{r}$ must be a `free` or `rep` mode: $\mu_\mathbf{o} = \texttt{co<>}$ means that $\Gamma_n, \kappa_n \vdash_\epsilon \hat{e} : \hat{\tau}$ excludes `read` and association modes for $\mu_\mathbf{r}$. Hence $\langle \hat{\pi}', \hat{\pi} \rangle$ is an ownership path-pair $q \xleftarrow{\mu_\mathbf{r}} \mathbf{s} \xrightarrow{\mu_\mathbf{r}} \omega = \omega.\epsilon$ with call-link or sent handle as initial edge and with the same shape. It, and the trivial $\rightleftharpoons$-path $\omega \dashrightarrow_{\not\rightleftharpoons}^{\epsilon} \omega$ **via** $\emptyset$ extend π_1's $\vec{\alpha}_0.\vec{\gamma}_0$-bridge from ω_0 to q **via** Π_1 to π's $\vec{\alpha}_0.\vec{\gamma}_0$-bridge from ω_0 to ω **via** $\Pi = \Pi_1 \cup \{\pi_q, \pi_\omega\} \cup \emptyset$. In order to ensure coverage by Π for all edges from π_2 (see below), simply chose π_q so that it includes π_2's `co`-edges on q''s side and chose π_ω so that it includes π_2's `co`-edges on ω's side.

Initially new $\pi_1 = h'_\mathbf{o} \bullet \pi_3 \bullet \pi_4$ means either $\pi_4 = \epsilon$ and $\Pi_1 = \emptyset$, or $\{\pi_4\} \cup \Pi_\Lambda \cup \Pi_\circledast \equiv \Pi_1 \cup \{h'_\mathbf{o}\}$. And internally really new $\pi_1 = \pi_3 \bullet \pi_4$ means $\{\pi_4\} \cup \Pi_\Lambda \cup \Pi_\circledast \equiv \Pi_1 \cup \Pi'_\mathbf{o}$. In internally really new π, π_3 is still the maximal common prefix, and $\pi_4 \bullet \pi_2$ is the rest: $\pi = \pi_1 \bullet \pi_2 = \pi_3 \bullet (\pi_4 \bullet \pi_2)$.

In (a), the case of $\Pi = \Pi_1 = \emptyset$ means for initially new π_1 that $\pi_4 = \pi'_4 = \epsilon$ and $\Pi_1 = \emptyset$. Hence the postfix of π and $\mathfrak{wit}(\pi)$ are the same: $\pi = h'_\mathbf{o} \bullet \tilde{\pi}$ and $\mathfrak{wit}(\pi) = h_\mathbf{o} \bullet \tilde{\pi}$ with $\Pi = \emptyset$. The case of $\Pi \equiv \Pi_1 \cup \{\pi_2\}$ means for initially new π_1 that $\{\pi_4 \bullet \pi_2\} \cup \Pi_\Lambda \cup \Pi_\circledast \equiv \Pi \cup \{h'_\mathbf{o}\}$. For internally really new π, it means $\{\pi_4 \bullet \pi_2\} \cup \Pi_\Lambda \cup \Pi_\circledast \equiv \Pi \cup \Pi'_\mathbf{o}$.

In (b), a co-closure path π_2's edges are $h'_\mathbf{o}$ or $h'^{-1}_\mathbf{o}$ or both, as well as old co-edges on $\mathbf{r}$'s and on $\mathbf{o}$'s side, which are contained in π_q and π_ω, or vice versa. Hence $\Pi = \Pi_1 \cup \{\pi_q, \pi_\omega\}$ means for internally really new π_1 that $\{\pi_4 \bullet \pi_2\} \cup \Pi_\Lambda \cup \Pi_\circledast \equiv \Pi \cup \Pi'_\mathbf{o} \cup \Pi''_\mathbf{o}$ where $\Pi''_\mathbf{o}$ is that non-empty subset of $\{h'_\mathbf{o}, h'^{-1}_\mathbf{o}\}$ contained in π_2. The case of initially new π_1 does not apply.

$\boxed{\text{Fourth,}}$ if π_1 is *initially new* or *internally really new*, and π_2 is an *association* path, then π_2 cannot be a co-closure path, and π is the same kind of path as π_1. The witness $\mathfrak{wit}(\pi_1) = \mathbf{s} \xrightarrow{\mu_\mathbf{o}} \mathbf{o} \xrightarrow{\texttt{co}}{}^* \mathbf{o}' \dashrightarrow^{\vec{\alpha}_0.\vec{\gamma}_0} q_0.\vec{\alpha}_0.\vec{\gamma}_0$ or $o \xrightarrow{\mu} \mathbf{o} \xrightarrow{\texttt{co}}{}^* \mathbf{o}' \dashrightarrow^{\vec{\alpha}_0.\vec{\gamma}_0} q_0.\vec{\alpha}_0.\vec{\gamma}_0$, respectively, is extended by dummy edge $\pi'_\beta = q_0.\vec{\alpha}_0.\vec{\gamma}_0 \xrightarrow{\beta\texttt{<>}} q_0.\vec{\alpha}_0.\vec{\gamma}_0.\beta$ to π's witness $\mathfrak{wit}(\pi) = \mathfrak{wit}(\pi_1) \bullet h'_\beta$, except in one case where the bridge is trivial (see below). The extension preserves the relationship between the shapes of the path and its witness. The corresponding $\vec{\alpha}_0.\vec{\gamma}_0.\beta$-bridge from q_0 to ω follows from π_1's $\vec{\alpha}_0.\vec{\gamma}_0$-bridge from q_0 to q.

(a) If π_2 is unchanged then π_1's $\vec{\alpha}_0.\vec{\gamma}_0$-bridge **via** Π_1 can be extended along $\pi_2 = q \dashrightarrow^{\beta\texttt{<>}} \omega$ to a $\vec{\alpha}_0.\vec{\gamma}_0.\beta$-bridge to ω **via** $\Pi \equiv \Pi_1 \cup \{\pi_2\} \cup \Pi_\beta$ (Lemma 14). In case of initially new $\pi_1 = h'_{\mathbf{o}} \bullet \pi_3 \bullet \pi_4$ while $\pi_4 = \pi'_4 = \epsilon$ and $\Pi_1 = \emptyset$, witness $\mathfrak{wit}(\pi_1) = h_{\mathbf{o}} \bullet \pi_3 \bullet \pi'_4$ targets q and can be extend along π_2 to a witness $\mathfrak{wit}(\pi) = \mathfrak{wit}(\pi_1) \bullet \pi_2$ targeting ω.

(b) If π_2 is a initially new path $\pi_2 = h'_{\mathbf{o}} \bullet \pi_5 \bullet \pi_6$ then $q = \mathbf{r}$. For π_2, Lemma 13 guarantees two cases:

- There may be a $\rightleftharpoons$-path $\mathbf{r} \overset{\beta}{\dashrightarrow}_{/\rightleftharpoons} \omega'_0$ **via** $\Pi_{2,0} \cup \{h_{\mathbf{r}}, \mathfrak{wit}(\pi_2)\} \equiv \Pi_{2,0} \cup \{h_{\mathbf{r}}, h_{\mathbf{o}}, \pi_5 \bullet \pi_6\}$. Then π_1's $\vec{\alpha}_0.\vec{\gamma}_0$-bridge from q_0 to $q = \mathbf{r}$ **via** Π_1 can be extended along this $\rightleftharpoons$-path to a $\vec{\alpha}_0.\vec{\gamma}_0.\beta$-bridge to ω'_0 **via** $\Pi'_1 \equiv \Pi_1 \cup \Pi_{2,0} \cup \{h_{\mathbf{r}}, \mathfrak{wit}(\pi_2)\} \cup \Pi_\beta$ (Lemma 14). It is extended by the rest of π_2's bridge to a $\vec{\alpha}_0.\vec{\gamma}_0.\beta$-bridge to ω **via** $\Pi \equiv \Pi_1 \cup \Pi_2 \{h_{\mathbf{r}}, h_{\mathbf{o}}, \pi_5 \bullet \pi_6\} \cup \Pi_\beta = \Pi_1 \cup \Pi_2 \cup \{\pi_5 \bullet \pi_6\} \cup \Pi_\Lambda \cup \Pi_\circledast$ where $\Pi_\Lambda = \{h_{\mathbf{r}}, h_{\mathbf{o}}\}$ and $\Pi_\circledast = \Pi_\beta$.
- Otherwise there is an ownership path-pair $\langle h_{\mathbf{r}} \bullet h_\beta, \mathfrak{wit}(\pi_2)\rangle = \mathbf{r}.\beta \overset{\hat{\mu}}{\dashleftarrow} \mathbf{s} \overset{\hat{\mu}}{\dashrightarrow} q'_0.\vec{\alpha}'_0.\vec{\gamma}'_0$ where $h_\beta = \mathbf{r} \xrightarrow{\beta} \mathbf{r}.\beta$. On the other hand, π_1's $\vec{\alpha}_0.\vec{\gamma}_0$-bridge from q_0 to $q = \mathbf{r}$ **via** Π_1 can be extended along dummy edge $h_\beta = \mathbf{r} \xrightarrow{\beta\text{<>}} \mathbf{r}.\beta$ to a $\vec{\alpha}_0.\vec{\gamma}_0.\beta$-bridge **via** $\Pi'_1 \equiv \Pi_1 \cup \{h_\beta\} \cup \Pi_\beta$ (Lemma 14). The ownership paths have the right initial edges to connect this bridge with π_2's $\vec{\alpha}_0.\vec{\gamma}_0$-bridge from q_0 to ω **via** Π_2 to a $\vec{\alpha}_0.\vec{\gamma}_0.\beta$-bridge to ω with path-base $\Pi \equiv \Pi'_1 \cup \Pi_2 \cup \{h_{\mathbf{r}} \bullet h_\beta, \mathfrak{wit}(\pi_2)\} \equiv \Pi'_1 \cup \Pi_2 \cup \{h_{\mathbf{r}}, h_\beta, h_{\mathbf{o}}, \pi_5 \bullet \pi'_6\}$ $= \Pi_1 \cup \Pi_2 \cup \{\pi_5\} \cup \Pi_\Lambda \cup \Pi_\circledast$ where $\Pi_\Lambda = \{h_{\mathbf{r}}, h_{\mathbf{o}}\}$ and $\Pi_\circledast = \Pi_\beta \cup \{h_\beta, \pi'_6\}$.

(c) If π_2 is internally really new then π_2's witness $\mathfrak{wit}(\pi_2) = q \overset{\mu_2}{\dashrightarrow} q'_0.\vec{\alpha}'_0.\vec{\gamma}'_0$ and the first $\rightleftharpoons$-path $q'_0 \overset{\vec{\alpha}'_0.\vec{\gamma}'_0}{\dashrightarrow}_{/\rightleftharpoons} \omega'_0$ **via** $\Pi_{2,0}$ in π_2's $\vec{\alpha}'_0.\vec{\gamma}'_0$-bridge **via** Π_2 mean $q \overset{\beta}{\dashrightarrow}_{/\rightleftharpoons} \omega'_0$ **via** $\Pi_{2,0} \cup \{\mathfrak{wit}(\pi_2)\}$ (Lemma 12). Along this $\rightleftharpoons$-path, π_1's $\vec{\alpha}_0.\vec{\gamma}_0$-bridge from q_0 to q **via** Π_1 can be extended to a $\vec{\alpha}_0.\vec{\gamma}_0.\beta$-bridge **via** $\Pi'_1 \equiv \Pi_1 \cup \Pi_{2,0} \cup \{\mathfrak{wit}(\pi_2)\} \cup \Pi_\beta$ (Lemma 14). It is extended by the rest of π_2's bridge to a $\vec{\alpha}_0.\vec{\gamma}_0.\beta$-bridge to ω **via** $\Pi \equiv \Pi_1 \cup \Pi_2 \cup \{\mathfrak{wit}(\pi_2)\} \cup \Pi_\beta$.

Initially new $\pi_1 = h'_{\mathbf{o}} \bullet \pi_3 \bullet \pi_4$ means either $\pi_4 = \epsilon$ and $\Pi_1 = \emptyset$, or $\{\pi_4\} \cup \Pi_\Lambda \cup \Pi'_\circledast \equiv \Pi_1 \cup \{h'_{\mathbf{o}}\}$. And internally really new $\pi_1 = \pi_3 \bullet \pi_4$ means $\{\pi_4\} \cup \Pi_\Lambda \cup \Pi'_\circledast \equiv \Pi_1 \cup \Pi'_{\mathbf{o}}$. In internally really new π, π_3 is still the maximal common prefix, and $\pi_4 \bullet \pi_2$ is the rest: $\pi = \pi_1 \bullet \pi_2 = \pi_3 \bullet (\pi_4 \bullet \pi_2)$.

In (a), $\Pi \equiv \Pi_1 \cup \{\pi_2\} \cup \Pi_\beta$. For initially new π, we get $\{\pi_4 \bullet \pi_2\} \cup \Pi_\Lambda \cup (\Pi'_\circledast \cup \Pi_\beta \cup \{h'_\beta\}) \equiv \Pi \cup \{h'_{\mathbf{o}}\}$. For internally really new π, we get $\{\pi_4 \bullet \pi_2\} \cup \Pi_\Lambda \cup (\Pi'_\circledast \cup \Pi_\beta) \equiv \Pi \cup \Pi'_{\mathbf{o}}$.

In (b), initially new $\pi_2 = \pi_5 \bullet \pi_6$ means $\{\pi_6\} \cup \Pi_\Lambda \cup \Pi''_\circledast \equiv \{h'_{\mathbf{o}}\} \cup \Pi_2$; and we have $\Pi \equiv \Pi_1 \cup \Pi_2 \cup \{\pi_5\} \cup \Pi_\Lambda \cup \Pi_\circledast$. This means for initially new π_1 that $\{\pi_4 \bullet \pi_2\} \cup \Pi_\Lambda \cup (\Pi_\circledast \cup \Pi'_\circledast \cup \Pi''_\circledast) \equiv \Pi \cup \{h'_{\mathbf{o}}\}$. And for internally really new π_1, it means $\{\pi_4 \bullet \pi_2\} \cup \Pi_\Lambda \cup (\Pi_\circledast \cup \Pi'_\circledast \cup \Pi''_\circledast) \equiv \Pi \cup \Pi'_{\mathbf{o}}$.

In (c), internally really new $\pi_2 = \pi_5 \bullet \pi_6$ with witness $\mathfrak{wit}(\pi_2) = \pi_5 \bullet \pi'_6$ means $\{\pi_6\} \cup \Pi_\Lambda \cup \Pi''_\circledast \equiv \Pi_2 \cup \Pi''_{\mathbf{o}}$. Hence $\Pi \equiv \Pi_1 \cup \Pi_2 \cup \{\mathfrak{wit}(\pi_2)\} \cup \Pi_\beta$ means for initially new π_1 that $\{\pi_4 \bullet \pi_2\} \cup \Pi_\Lambda \cup (\Pi'_\circledast \cup \Pi''_\circledast \cup \Pi_\beta \cup \{\pi'_6\}) \equiv \Pi \cup \{h'_{\mathbf{o}}\}$, and means for internally really new π_1 that $\{\pi_4 \bullet \pi_2\} \cup \Pi_\Lambda \cup (\Pi'_\circledast \cup \Pi''_\circledast \cup \Pi_\beta \cup \{\pi'_6\}) \equiv \Pi \cup \Pi'_{\mathbf{o}} \cup \Pi''_{\mathbf{o}}$. ■

6.3.2 Technical Lemmas for the Potential Access Path Level

This subsection supplements the technical lemmas for the proof of Lemma 9 on the new potential access paths after supply of a handle parameter. The first three lemmas are about the shapes of co- or association path-extended paths, the closure of $\rightleftharpoons$-paths under co-paths, and the closure of $\rightleftharpoons$-paths under the $\rightleftharpoons$-regions corresponding to region objects.

Lemma 10 (Shape extension) If potential access path $\pi_1 \in PAP(o, \mu_1, q)$ has shape $o \xrightarrow{\mu} u \xrightarrow{\mathsf{co}}{}^{*} \bullet \overset{\vec{\alpha}}{\dashrightarrow} \bullet$ and $\pi_2 \in PAP(q, \mu_2, \omega)$ is a potential access path of mode $\mu_2 = \mathsf{co<>}$ or α<> then $\pi_1 \bullet \pi_2$ has shape $o \xrightarrow{\mu} u \xrightarrow{\mathsf{co}}{}^{*} \bullet \overset{\vec{\alpha}}{\dashrightarrow} \bullet$ or shape $o \xrightarrow{\mu} u \xrightarrow{\mathsf{co}}{}^{*} \bullet \overset{\vec{\alpha}.\alpha}{\dashrightarrow} \bullet$, respectively.

Proof: Shape $o \xrightarrow{\mu} u \xrightarrow{\mathsf{co}}{}^{*} \bullet \overset{\vec{\alpha}}{\dashrightarrow} \bullet$, with $\vec{\alpha} = \alpha_1 \dots \alpha_n$ means that π_1 is a path $o \xrightarrow{\mu} u \xrightarrow{\mathsf{co}}{}^{*} u_1 \overset{\alpha_1}{\dashrightarrow} u_2 \dots u_n \overset{\alpha_n}{\dashrightarrow} q$. Hence if $\mu_2 = \alpha$<>, then its extension by $\pi_2 = q \overset{\alpha}{\dashrightarrow} \omega$ obviously has the shape $o \xrightarrow{\mu} u \xrightarrow{\mathsf{co}}{}^{*} \bullet \overset{\vec{\alpha}.\beta}{\dashrightarrow} \bullet$. In case of $\mu_2 = \mathsf{co<>}$ and $\vec{\alpha} = \epsilon$, π_1 is $o \xrightarrow{\mu} u \xrightarrow{\mathsf{co}}{}^{*} q$. It is extended by $\pi_2 = q \overset{\mathsf{co}}{\dashrightarrow} \omega = q \xrightarrow{\mathsf{co}}{}^{*} \omega$ to a path $o \xrightarrow{\mu} u \xrightarrow{\mathsf{co}}{}^{*} \omega$ with shape $o \xrightarrow{\mu} u \xrightarrow{\mathsf{co}}{}^{*} \bullet \overset{\epsilon}{\dashrightarrow} \bullet$. In case of $\mu_2 = \mathsf{co<>}$ and $\vec{\alpha} \neq \epsilon$, there is a last association path $u_n \overset{\alpha_n}{\dashrightarrow} q$ in π_1. It is extended by $\pi_2 = q \overset{\mathsf{co}}{\dashrightarrow} \omega$ to $u_n \overset{\alpha_n}{\dashrightarrow} \omega$. Hence $\pi_1 \bullet \pi_2$ is $o \xrightarrow{\mu} u \xrightarrow{\mathsf{co}}{}^{*} u_1 \overset{\vec{\alpha}}{\dashrightarrow} \omega$, with the obvious shape $o \xrightarrow{\mu} u \xrightarrow{\mathsf{co}}{}^{*} \bullet \overset{\vec{\alpha}}{\dashrightarrow} \bullet$. ■

Lemma 11 If $o \overset{\vec{\alpha}}{\dashrightarrow}_{/\rightleftharpoons} q$ **via** Π with $\vec{\alpha} \neq \epsilon$ or $o \neq q$, and $\pi \in PAP(q, \mathsf{co<>}, \omega)$ then $o \overset{\vec{\alpha}}{\dashrightarrow}_{/\rightleftharpoons} \omega$ **via** $\Pi' = \Pi \cup \{\pi' \bullet \pi\}$ for some $\pi' \in \Pi$.

Proof by induction on the definition of $o \overset{\vec{\alpha}}{\dashrightarrow}_{/\rightleftharpoons} q$: Note that the condition excludes trivial $\rightleftharpoons$-paths $o \overset{\epsilon}{\dashrightarrow}_{/\rightleftharpoons} o = \omega$. Hence the only base case is a $\rightleftharpoons$-path **via** $\{\pi'\}$ with $\vec{\alpha} = \beta$ based on potential access path $\pi' \in PAP(o, \beta$<>$, q)$. Then also $\pi' \bullet \pi \in PAP(o, \beta$<>$, q)$, and thus $o \overset{\vec{\alpha}}{\dashrightarrow}_{/\rightleftharpoons} \omega$ **via** $\{\pi' \bullet \pi\}$. In the induction step, if $o \overset{\vec{\alpha}}{\dashrightarrow}_{/\rightleftharpoons} q$ because $o.\vec{\alpha} \rightleftharpoons q'.\vec{\gamma}$ **via** Π_1 and $q' \overset{\vec{\gamma}}{\dashrightarrow}_{/\rightleftharpoons} q$ **via** Π_2, then $q' \overset{\vec{\gamma}}{\dashrightarrow}_{/\rightleftharpoons} \omega$ **via** Π_2' by induction hypothesis, and thus $o \overset{\vec{\alpha}}{\dashrightarrow}_{/\rightleftharpoons} \omega$ **via** $\Pi_1 \cup \Pi_2'$ by $o.\vec{\alpha} \rightleftharpoons q'.\vec{\gamma}$. If $o \overset{\vec{\alpha}}{\dashrightarrow}_{/\rightleftharpoons} q$ because $o \overset{\vec{\alpha}_1}{\dashrightarrow}_{/\rightleftharpoons} q'$ **via** Π_1 and $q' \overset{\vec{\alpha}_2}{\dashrightarrow}_{/\rightleftharpoons} \omega$ **via** Π_2 with $\vec{\alpha}_1 \bullet \vec{\alpha}_2 = \vec{\alpha}$ then in case of $\vec{\alpha}_2 \neq \epsilon$ or $q' \neq q$, $q' \overset{\vec{\alpha}_2}{\dashrightarrow}_{/\rightleftharpoons} \omega$ **via** Π_2' by induction hypothesis, and thus $o \overset{\vec{\alpha}}{\dashrightarrow}_{/\rightleftharpoons} \omega$ **via** $\Pi_1 \cup \Pi_2'$. And in case of $\vec{\alpha}_2 = \epsilon$ and $q' = q$, $\Pi_2 = \emptyset$ and the condition "$\vec{\alpha} \neq \epsilon$ or $o \neq q$" implies that $\vec{\alpha}_1 \neq \epsilon$ or $o \neq q' = q$. Hence $o \overset{\vec{\alpha}_1}{\dashrightarrow}_{/\rightleftharpoons} q' = q$ and π guarantee $o \overset{\vec{\alpha}_1}{\dashrightarrow}_{/\rightleftharpoons} \omega$ **via** Π_1' by induction hypothesis. That is $o \overset{\vec{\alpha}}{\dashrightarrow}_{/\rightleftharpoons} \omega$ **via** Π_1' since $\vec{\alpha}_1 = \vec{\alpha}_1 \bullet \epsilon = \vec{\alpha}_1 \bullet \vec{\alpha}_2 = \vec{\alpha}$. ■

Lemma 12 If $o \overset{\vec{\gamma}}{\dashrightarrow}_{/\rightleftharpoons} q.\vec{\alpha}$ **via** Π and $q \overset{\vec{\alpha}}{\dashrightarrow}_{/\rightleftharpoons} \omega$ **via** Π' then $o \overset{\vec{\gamma}}{\dashrightarrow}_{/\rightleftharpoons} \omega$ **via** $\Pi \cup \Pi'$.

Proof: Axiom $q.\vec{\alpha} \overset{\epsilon}{\dashrightarrow}_{/\rightleftharpoons} q.\vec{\alpha}$ **via** $\emptyset$ implies $(q.\vec{\alpha}).\epsilon.\vec{\beta} \rightleftharpoons (q.\vec{\alpha}).\vec{\beta}$ **via** $\emptyset$. With $\vec{\beta} = \epsilon$, this makes $q \overset{\vec{\alpha}}{\dashrightarrow}_{/\rightleftharpoons} \omega$ mean $q.\vec{\alpha} \overset{\epsilon}{\dashrightarrow}_{/\rightleftharpoons} \omega$ **via** $\Pi' \cup \emptyset$. It extends $o \overset{\vec{\gamma}}{\dashrightarrow}_{/\rightleftharpoons} q.\vec{\alpha}$ to $o \overset{\vec{\gamma} \bullet \epsilon}{\dashrightarrow}_{/\rightleftharpoons} \omega$, i.e., $o \overset{\vec{\gamma}}{\dashrightarrow}_{/\rightleftharpoons} \omega$, **via** $\Pi \cup \Pi'$. ■

The next three lemmas are in the context of parameter supply substeps during a *legal* {call}-reduction step, i.e., with a typing $\Gamma_n, \kappa_n \vdash_\epsilon \hat{e} : \hat{\tau}$ for the operation call expression redex $\hat{e}$. They concern the graph $\mathfrak{g}'$ after addition of a new received handle $h_\mathbf{o} = \mathbf{r} \xrightarrow{\mu_\mathbf{o}} \mathbf{o}$ and removal of old sent handle $h'_\mathbf{o} = \mathbf{s} \xrightarrow{\mu_\mathbf{r} \circ \mu_\mathbf{o}} \mathbf{o}$ from the previous graph $\mathfrak{g}$, i.e., $\mathfrak{g}' = \mathfrak{g} \oplus \mathbf{r} \xrightarrow{\mu_\mathbf{o}} \mathbf{o} \ominus \mathbf{s} \xrightarrow{\mu_\mathbf{r} \circ \mu_\mathbf{o}} \mathbf{o}$.

Lemma 13 Consider an initially new association path $\pi \in PAP_{\mathfrak{g}'}(\mathbf{r}, \beta\texttt{<>}, \omega)$ whose $\vec{\alpha}_0.\vec{\gamma}_0$-bridge from q_0 to ω in $\mathfrak{g}$ starts with $\rightleftharpoons$-path $q_0 \overset{\vec{\alpha}_0.\vec{\gamma}_0}{\dashrightarrow}_{/\rightleftharpoons} \omega_0$ **via** Π_0. In $\mathfrak{g}$, there was

- there was a $\rightleftharpoons$-path $\mathbf{r} \overset{\beta}{\dashrightarrow}_{/\rightleftharpoons} \omega_0$ **via** $\{h_\mathbf{r}, \mathfrak{wit}(\pi)\} \cup \Pi_0$, or
- $h_\mathbf{r} \bullet \mathbf{r} \xrightarrow{\beta\texttt{<>}} \mathbf{r}.\beta$ and $\mathfrak{wit}(\pi)$ constitute an ownership path-pair $\mathbf{r}.\beta \overset{\hat{\mu}}{\dashleftarrow} \mathbf{s} \overset{\hat{\mu}}{\dashrightarrow} q_0.\vec{\alpha}_0.\vec{\gamma}_0$ of shape $\mathbf{s} \xrightarrow{\mu_\mathbf{r}} \mathbf{r} \xrightarrow{\texttt{co}}{}^* \bullet \overset{\mu_\mathbf{o}(\vec{\alpha}_0).\vec{\gamma}_0}{\dashrightarrow} \bullet$ and $\mathbf{s} \xrightarrow{\mu_\mathbf{r} \circ \mu_\mathbf{o}} \mathbf{o} \xrightarrow{\texttt{co}}{}^* \bullet \overset{\vec{\alpha}_0.\vec{\gamma}_0}{\dashrightarrow} \bullet$, respectively.

Proof: First, an initially new π has some shape $\mathbf{r} \xrightarrow{\mu_\mathbf{o}} \mathbf{o} \xrightarrow{\texttt{co}}{}^* \bullet \overset{\vec{\alpha}_0.\vec{\gamma}_0}{\dashrightarrow} \bullet$ with $\mu_\mathbf{o}(\vec{\alpha}_0.\vec{\gamma}_0) = \beta$ since π's mode is $\beta\texttt{<>}$. But if $\mu_\mathbf{o}$ contains β, then the existence $h'_\mathbf{o} = \mathbf{s} \xrightarrow{\mu_\mathbf{r} \circ \mu_\mathbf{o}} \mathbf{o}$ presupposes for some $\hat{\mu}$ a corresponding correlation β=$\hat{\mu}$ in the call-link's mode $\mu_\mathbf{r}$. It specifies that $\mu_\mathbf{r} \circ \beta\texttt{<>} = \hat{\mu}$, meaning that π's witness $\mathfrak{wit}(\pi)$ has mode $\mu_\mathbf{r} \circ \mu = \mu_\mathbf{r} \circ \beta\texttt{<>} = \hat{\mu}$.

Second, as a mode with a correlation β=$\hat{\mu}$, $\mu_\mathbf{r}$ cannot be a co- or association mode; and $\Gamma_n, \kappa_n \vdash_\epsilon \hat{e} : \hat{\tau}$ excludes that $\mu_\mathbf{r}$ is read—$\mu_\mathbf{r}$ can only be $\texttt{free<}\ldots, \beta\texttt{=}\hat{\mu}, \ldots\texttt{>}$ or $\texttt{rep<}\ldots, \beta\texttt{=}\hat{\mu}, \ldots\texttt{>}$.

Third, since $\hat{\mu}$ occurs in correlation β=$\hat{\mu}$, it cannot be co; and $\mu_\mathbf{r} \circ \beta\texttt{<>} = \hat{\mu}$ cannot be a read mode since $\mu_\mathbf{o}(\vec{\alpha}_0.\vec{\gamma}_0) = \beta$ is not read, so that $\Gamma_n, \kappa_n \vdash_\epsilon \hat{e} : \hat{\tau}$ ensures that $\mu_\mathbf{r} \circ \mu_\mathbf{o}(\vec{\alpha}_0.\vec{\gamma}_0)$ is not read either—$\hat{\mu}$ must be free, rep, or an association mode:

- In case of an association mode $\hat{\mu} = \alpha\texttt{<>}$, the call-link $h_\mathbf{r} = \mathbf{s} \xrightarrow{\mu_\mathbf{r}} \mathbf{r}$ established region-coupling $\mathbf{r}.\beta \rightleftharpoons \mathbf{s}.\alpha$ **via** $\{h_\mathbf{r}\}$ because $\mu_\mathbf{r} = \texttt{free<}\ldots, \beta\texttt{=}\alpha\texttt{<>}, \ldots\texttt{>}$ or $\texttt{rep<}\ldots, \beta\texttt{=}\alpha\texttt{<>}, \ldots\texttt{>}$. But then witness $\mathfrak{wit}(\pi) \in PAP_\mathfrak{g}(\mathbf{s}, \alpha\texttt{<>}, q_0.\vec{\alpha}_0.\vec{\gamma}_0)$ means the $\rightleftharpoons$-path $\mathbf{r} \overset{\beta}{\dashrightarrow}_{/\rightleftharpoons} q_0.\vec{\alpha}_0.\vec{\gamma}_0$ **via** $\{h_\mathbf{r}, \mathfrak{wit}(\pi)\}$. It and the first $\rightleftharpoons$-path $q_0 \overset{\vec{\alpha}_0.\vec{\gamma}_0}{\dashrightarrow}_{/\rightleftharpoons} \omega_0$ of π's $\vec{\alpha}_0.\vec{\gamma}_0$-bridge entail $\mathbf{r} \overset{\beta}{\dashrightarrow}_{/\rightleftharpoons} \omega_0$ **via** $\{h_\mathbf{r}, \mathfrak{wit}(\pi)\} \cup \Pi_0$ (Lemma 12).
- In case of a free or rep mode $\hat{\mu}$, consider on one side the extension of the call-link with correlation β=$\hat{\mu}$ to the $\hat{\mu}$-path $\hat{\pi} = \mathbf{s} \xrightarrow{\mu_\mathbf{r}} \mathbf{r} \xrightarrow{\beta\texttt{<>}} \mathbf{r}.\beta$ with shape $\mathbf{s} \xrightarrow{\mu_\mathbf{r}} \mathbf{r} \xrightarrow{\texttt{co}}{}^* \bullet \overset{\beta}{\dashrightarrow} \bullet$. On the other side is π's witness $\mathfrak{wit}(\pi) \in PAP_\mathfrak{g}(\mathbf{s}, \hat{\mu}, q_0.\vec{\alpha}_0.\vec{\gamma}_0)$ with shape $\mathbf{s} \xrightarrow{\mu_\mathbf{r} \circ \mu_\mathbf{o}} \mathbf{o} \xrightarrow{\texttt{co}}{}^* \bullet \overset{\vec{\alpha}_0.\vec{\gamma}_0}{\dashrightarrow} \bullet$ and $\mu_\mathbf{o}(\vec{\alpha}_0) \in \mathbb{A}$. Both $\hat{\mu}$-paths constitute the ownership path-pair $\mathbf{r}.\beta \overset{\hat{\mu}}{\dashleftarrow} \mathbf{s} \overset{\hat{\mu}}{\dashrightarrow} q_0.\vec{\alpha}_0.\vec{\gamma}_0$. Observe that $\hat{\pi}$ has the right shape: Since association modes have no correlations, $\mu_\mathbf{o}(\vec{\alpha}_0.\vec{\gamma}_0) = \beta$ from above means for $\mu_\mathbf{o}(\vec{\alpha}_0) \in \mathbb{A}$ that $\vec{\gamma}_0$ must be ϵ. Hence $\mu_\mathbf{o}(\vec{\alpha}_0) = \mu_\mathbf{o}(\vec{\alpha}_0.\vec{\gamma}_0) = \beta$, so that shape $\mathbf{s} \xrightarrow{\mu_\mathbf{r}} \mathbf{r} \xrightarrow{\texttt{co}}{}^* \bullet \overset{\beta}{\dashrightarrow} \bullet$ is shape $\mathbf{s} \xrightarrow{\mu_\mathbf{r}} \mathbf{r} \xrightarrow{\texttt{co}}{}^* \bullet \overset{\mu_\mathbf{o}(\vec{\alpha}_0).\vec{\gamma}_0}{\dashrightarrow} \bullet$. ■

Lemma 14 (Bridge extension) Consider an initially new or internally really new path's $\vec{\alpha}_0$-bridge from q_0 to q **via** Π. For any $\rightleftharpoons$-path $q \overset{\beta}{\dashrightarrow}_{/\rightleftharpoons} \omega$ **via** Π', there is a $\vec{\alpha}_0.\beta$-bridge from q_0 to ω. Its path-base $\Pi'' \equiv \Pi \cup \Pi' \cup \Pi_\beta$ contains besides Π and Π' a certain set Π_β of dummy edges $u \xrightarrow{\beta\texttt{<>}} u.\beta$.

Proof: Each $\rightleftharpoons$-path $q_i \overset{\vec{\alpha}_i}{\dashrightarrow}_{/\rightleftharpoons} \omega_{i+1}$ **via** Π_i in the bridge can be extended to $q_i \overset{\vec{\alpha}_i.\beta}{\dashrightarrow}_{/\rightleftharpoons} \omega_{i+1}.\beta$ **via** $\Pi_i \cup \{\omega_{i+1} \xrightarrow{\beta\texttt{<>}} \omega_{i+1}.\beta\}$, and the final $q_n \overset{\vec{\alpha}_n}{\dashrightarrow}_{/\rightleftharpoons} \omega_{n+1} = q$ **via** Π_n can be extended to $q_n \overset{\vec{\alpha}_n.\beta}{\dashrightarrow}_{/\rightleftharpoons} \omega$ **via** $\Pi_n \cup \Pi'$. But a corresponding extension of the bridge's ownership path-pairs $\langle \pi_i', \pi_i \rangle = \omega_i \overset{\hat{\mu}_i}{\dashleftarrow} \mathbf{s} \overset{\hat{\mu}_i}{\dashrightarrow} q_i.\vec{\alpha}_i$ depends on the correlations in $\hat{\mu}_i$:

- There can never be correlations $\beta\texttt{=co<>}$ to $\texttt{co<>}$.
- There can be no correlation to `read` in the mode $\hat{\mu}_i$ of extensions π_i' and π_i of $h_\mathbf{r}$ or $h_\mathbf{o}'$: $\Gamma_n, \kappa_n \vdash_\epsilon \hat{e} : \hat{\tau}$ ensures that the base-mode $\mu_\mathbf{r} \circ \mu_\mathbf{o}(\vec{\beta})$ of extensions of the sent handle are `read` only if corresponding extension of the received handle have base-mode $\mu_\mathbf{o}(\vec{\beta}) = \texttt{read}$ too. This excludes the case of mode $\mu_\mathbf{o} = \texttt{co<>}$ from which no `read` mode can be extracted. In case of $\mu_\mathbf{o}(\vec{\beta}) \in \mathbb{A}$ for some $\vec{\beta}$, π_i has shape $\mathbf{s} \xrightarrow{\mu_\mathbf{r} \circ \mu_\mathbf{o}} \mathbf{o} \xrightarrow{\texttt{co}}{}^{*} \bullet \overset{\vec{\alpha}_i}{\dashrightarrow} \bullet$ with $\mu_\mathbf{o}(\vec{\alpha}) = \beta \in \mathbb{A}$ for some decomposition $\vec{\alpha}_i = \vec{\alpha}.\vec{\gamma}$. This means that the received handle's extension along a $\vec{\alpha}$-sequence of association paths is a path of mode $\beta\texttt{<>}$, which cannot be extended further to a `read` path.
- If $\hat{\mu}_i$ has a correlation $\beta\texttt{=}\hat{\mu}_i'$ to a `free` or `rep` mode, then π_i' and π_i are extended by dummy edges $h_i = \omega_i \xrightarrow{\beta\texttt{<>}} \omega_i.\beta$ and $h_i' = q_i.\vec{\alpha}_i \xrightarrow{\beta\texttt{<>}} q_i.\vec{\alpha}_i.\beta$ to the ownership path-pair $\langle \hat{\pi}_i, \hat{\pi}_i \rangle = \omega_i.\beta \overset{\hat{\mu}_i'}{\dashleftarrow} \mathbf{s} \overset{\hat{\mu}_i'}{\dashrightarrow} q_i.\vec{\alpha}_i.\beta$ for the $\vec{\alpha}_0.\beta$-bridge from q_0 to ω.
- If $\hat{\mu}_i$ contains a correlation $\beta\texttt{=}\alpha_i\texttt{<>}$ to an association mode or no β-correlation at all, i.e., if $\hat{\mu}_i(\beta) \in \mathbb{A}_\perp$, then ownership path-pair $\langle \pi_i', \pi_i \rangle$ establishes the region-couplings $\omega_i.\beta \rightleftharpoons \mathbf{s}.\alpha_i \rightleftharpoons (q_i.\vec{\alpha}_i).\beta$ or $\omega_i.\beta \rightleftharpoons \mathbf{s}.\perp \rightleftharpoons (q_i.\vec{\alpha}_i).\beta$, respectively. That is, $\omega_i.\beta \rightleftharpoons q_i.\vec{\alpha}_i.\beta$ **via** $\{\pi_i', \pi_i\}$. Additionally, $\rightleftharpoons$-path $q_i \overset{\vec{\alpha}_i}{\dashrightarrow}_{/\rightleftharpoons} \omega_{i+1}$ **via** Π_i implies $q_i.\vec{\alpha}_i.\beta \rightleftharpoons \omega_{i+1}.\beta$ **via** Π_i. Hence we have $\omega_i.\beta \rightleftharpoons \omega_{i+1}.\beta$ **via** $\Pi_i' = \Pi_i \cup \{\pi_i', \pi_i\}$. Observe that the bridge's path-base Π contained π_i', π_i and Π_i.

If there are consecutive modes $\hat{\mu}_i, \ldots, \hat{\mu}_j$ with $\hat{\mu}_k(\beta) \in \mathbb{A}_\perp$ they together imply $\omega_i.\beta \rightleftharpoons \omega_{j+1}.\beta$ **via** $\Pi_i \cup \{\pi_i', \pi_i\} \cup \ldots \cup \Pi_j \cup \{\pi_j', \pi_j\}$. It, and dummy edge $h_{j+1} = \omega_{j+1} \xrightarrow{\beta\texttt{<>}} \omega_{j+1}.\beta$ (if $j < n$) or extension $\rightleftharpoons$-path $\omega_{j+1} = q \overset{\beta}{\dashrightarrow}_{/\rightleftharpoons} \omega$ (if $j = n$) imply $\omega_i \overset{\beta}{\dashrightarrow}_{/\rightleftharpoons} \omega_{j+1}.\beta$ or ω. It extends the $\rightleftharpoons$-path $q_{i-1} \overset{\vec{\alpha}_{i-1}}{\dashrightarrow}_{/\rightleftharpoons} \omega_i$ **via** Π_{i-1} to $q_{i-1} \overset{\vec{\alpha}_{i-1}.\beta}{\dashrightarrow}_{/\rightleftharpoons} \omega_{j+1}.\beta$ or ω, respectively. It closes the gap between q_0 (if $i = 0$) or the preceding extended ownership path-pair (if $i > 0$) to the left, and ω (if $j = n$) or the following extended ownership path-pair (if $j < n$) to the right in the $\vec{\alpha}_0.\beta$-bridge from q_0 to ω. Its path-base is $\Pi_{i-1} \cup \Pi_i \cup \{\pi_i', \pi_i\} \cup \ldots \cup \Pi_j \cup \{\pi_j', \pi_j\} \cup \Pi_{j+1}'$ with $\Pi_{j+1}' = \{h_{j+1}\}$ or Π', respectively.

All in all, we have a $\vec{\alpha}_0.\beta$-bridge from q_0 to ω whose path-base Π'' contains besides Π' some dummy edges h_i and h_i', all paths from Π either directly or in extended form: the Π_i of the $\vec{\alpha}_0$-bridge's $\rightleftharpoons$-paths, the ownership paths π_i' and π_i with $\hat{\mu}_i{\cdot}\alpha \notin \{\texttt{free}, \texttt{rep}\}$ as well as those with $\hat{\mu}_i{\cdot}\alpha \in \{\texttt{free}, \texttt{rep}\}$. ∎

6.3.3 Change Modulo Region-Couplings

Reasoning *with* the reserved ownership assumption makes the proof of Unique Owner and Unique Head possible. But reasoning *about* it requires additional work: Not only the new ownership paths have to be considered, but also new $\rightleftharpoons$-paths established by new association paths and new region-couplings. Hence the next logical step, before proving the reserved ownership assumption, is to determine what changes at the level of $\rightleftharpoons$-paths in the relevant (substeps of) reductions with {new}, {upd}, {ret}, and {call}. We will find that in case of {upd} and {ret}, there are no new region-couplings and no new association paths modulo region-coupling.

Lemma 15 Consider object creation, i.e., the addition of a `free` edge $h_\mathbf{o} = \mathbf{r} \xrightarrow{\mu_\mathbf{o}} \mathbf{o}$ to a fresh object: $\mathfrak{g}' = \mathfrak{g} \oplus \mathbf{r} \xrightarrow{\mu_\mathbf{o}} \mathbf{o}$.

Define the substitution $\sigma(q) =_{\text{df}} \begin{cases} \mathbf{r}.\mu_\mathbf{o}(\vec{\alpha}).\vec{\alpha}' & \text{if } q = \mathbf{o}.\vec{\alpha}.\vec{\alpha}' \text{ with } \mu_\mathbf{o}(\vec{\alpha}) \in \mathbb{A}_\perp \\ q & \text{otherwise} \end{cases}$

a) In $\mathfrak{g}'^{\circledast}$, there are three kinds of *non-trivial* $\rightleftharpoons$-paths $\varphi = o \overset{\vec{\gamma}}{\dashrightarrow}_{/\rightleftharpoons} \omega$ **via** Π:

- φ has a counterpart $\sigma(o) \overset{\vec{\gamma}}{\dashrightarrow}_{/\rightleftharpoons} \sigma(\omega)$ **via** Π' in $\mathfrak{g}^{\circledast}$ and neither o nor ω are region objects $\mathbf{o}.\beta_1 \ldots \beta_k$ of $\mathbf{o}$ such that for all $i \leq k$, $\mu_\mathbf{o}(\beta_1 \ldots \beta_i) \notin \mathbb{A}_\perp$.
- φ is a $\rightleftharpoons$-path $o = \mathbf{o}.\vec{\beta} \overset{\vec{\gamma}'}{\dashrightarrow}_{/\rightleftharpoons} \mathbf{o}.\vec{\beta}.\vec{\gamma}' = \omega$ **via** dummy edges from one region object of $\mathbf{o}$ to another one such that for all prefixes $\vec{\beta}'$ of $\vec{\beta}$ we have $\mu_\mathbf{o}(\vec{\beta}') \notin \mathbb{A}_\perp$. That is, $\Pi \subseteq \Pi_{\circledast}$, which is defined below.
- φ is the combination $\mathbf{o}.\vec{\beta} \overset{\vec{\gamma}_1}{\dashrightarrow}_{/\rightleftharpoons} \mathbf{o}.\vec{\beta}.\vec{\gamma}_1 \overset{\vec{\gamma}_2}{\dashrightarrow}_{/\rightleftharpoons} \omega$, with $\vec{\gamma}_1.\vec{\gamma}_2 = \vec{\gamma}$, of an (unchangeable) $\rightleftharpoons$-path of the second kind and a $\rightleftharpoons$-path of the first kind, with $\mu_\mathbf{o}(\vec{\beta}.\vec{\gamma}_1) \in \mathbb{A}_\perp$.

b) In $\mathfrak{g}'^{\circledast}$, all *new* region-couplings $\omega.\vec{\gamma} \rightleftharpoons o.\vec{\alpha}$ **via** Π have a counterpart $\sigma(\omega.\vec{\gamma}) \rightleftharpoons \sigma(o.\vec{\alpha})$ **via** Π' in $\mathfrak{g}^{\circledast}$. If $\omega.\vec{\gamma} = \mathbf{o}.\beta_1 \ldots \beta_n$ or $o.\vec{\alpha} = \mathbf{o}.\beta_1 \ldots \beta_n$ then there is an $i \leq n$ such that $\mu_\mathbf{o}(\beta_1 \ldots \beta_i) \in \mathbb{A}_\perp$.

The path-base Π of first-kind $\rightleftharpoons$-paths and of new region-couplings contains, apart from initially new ownership paths $\pi \in \Pi_{\mathcal{N}}$ and apart from dummy edges $h' \in \Pi_{\circledast}$ of $\mathbf{o}$ and of its region objects $\mathbf{o}.\vec{\alpha}$ on the way to $\mu_\mathbf{o}(\vec{\alpha}.\alpha.\vec{\alpha}') \in \mathbb{A}_\perp$ only edges with counterparts in their counterpart's path-base Π', and contains them all: $\sigma(\Pi \setminus \Pi_{\mathcal{N}} \setminus \Pi_{\circledast}) \equiv \Pi'$,
where $\Pi_{\mathcal{N}} =_{\text{df}} \{\mathbf{r} \xrightarrow{\mu_\mathbf{o}} \mathbf{o} \xrightarrow{\vec{\alpha}} \mathbf{o}.\vec{\alpha} \mid \mu_\mathbf{o}(\vec{\alpha}) \in \{\texttt{free}, \texttt{rep}\} \wedge \exists \vec{\alpha}' \boldsymbol{.}\ \mu_\mathbf{o}(\vec{\alpha}.\vec{\alpha}') \in \mathbb{A}_\perp\}$
$\Pi_{\circledast} =_{\text{df}} \{\mathbf{o}.\vec{\alpha} \xrightarrow{\alpha} \mathbf{o}.\vec{\alpha}.\alpha \mid \mu_\mathbf{o}(\vec{\alpha}) \notin \mathbb{A}_\perp \wedge \exists \vec{\alpha}' \boldsymbol{.}\ \mu_\mathbf{o}(\vec{\alpha}.\alpha.\vec{\alpha}') \in \mathbb{A}_\perp\}$

Proof by simultaneous induction on the definition of $\rightleftharpoons$-paths and region-couplings: Lemma 6 guarantees that all potential access paths in $\mathfrak{g}'^{\circledast}$ are unchanged, initially new, or internally new. Notice that the witnesses $\sigma(\pi)$ of initially new paths from o to ω go from $\sigma(o)$ to $\sigma(\omega)$. The base case, a non-trivial $\rightleftharpoons$-path $o \overset{\beta}{\dashrightarrow}_{/\rightleftharpoons} \omega$ **via** $\{\pi\}$ is based on a path $\pi \in PAP_{\mathfrak{g}'^{\circledast}}(o, \beta\text{<>}, \omega)$, and a new region-coupling $\omega.\beta \rightleftharpoons o.\mu(\beta)$ **via** $\{\pi\}$ is established by an ownership path $\pi \in PAP_{\mathfrak{g}'^{\circledast}}(o, \mu, \omega)$ with $\mu(\beta) \in \mathbb{A}_\perp$.

- If π is initially new or internally new, then $o \neq \mathbf{o}.\vec{\beta}$ and $\omega = \mathbf{o}.\vec{\alpha}.\vec{\alpha}'$ with $\mu_\mathbf{o}(\vec{\alpha}) \in \mathbb{A}$. Hence $\sigma(o) = o$ and $\sigma(\omega) = \sigma(\mathbf{o}.\vec{\alpha}.\vec{\alpha}') = \mathbf{r}.\mu_\mathbf{o}(\vec{\alpha}).\vec{\alpha}'$. If π has a witness $\sigma(\pi)$, it connects $\sigma(o)$ and $\sigma(\omega)$. Hence in case of $\rightleftharpoons$-paths, $\sigma(\pi)$ is the necessary counterpart $\sigma(o) \overset{\beta}{\dashrightarrow}_{/\rightleftharpoons} \sigma(\omega)$ **via** $\{\sigma(\pi)\}$ for a first-kind $\rightleftharpoons$-path. And in case of new region-coupling and internally new π, $\sigma(\pi)$ is an ownership path that established $\mathbf{r}.\mu_{\mathbf{o}}(\vec{\alpha}).\vec{\alpha}'.\beta = \sigma(\omega).\beta = \sigma(\omega.\beta) \rightleftharpoons o.\mu(\beta) = \sigma(o).\mu(\beta) = \sigma(o.\mu(\beta))$ **via** $\{\sigma(\pi)\}$. If π is an initially new ownership path $\pi_{\vec{\beta}}$ then $o = \mathbf{r}$, $\omega = \mathbf{o}.\vec{\beta}$, and $\mu(\beta) = \mu_{\mathbf{o}}(\vec{\beta}.\beta)$. Hence trivial $\sigma(\mathbf{o}.\vec{\beta}.\beta) = \mathbf{r}.\mu_{\mathbf{o}}(\vec{\beta}.\beta) = \mathbf{r}.\mu(\beta) \rightleftharpoons \mathbf{r}.\mu(\beta)$ **via** $\Pi' = \emptyset$ is the witness for $\mathbf{o}.\vec{\beta}.\beta \rightleftharpoons \mathbf{r}.\mu(\beta)$ **via** $\Pi = \{\pi_{\vec{\beta}}\}$. Since $\pi_{\vec{\beta}} \in \Pi_{\mathfrak{N}}$, $\sigma(\Pi \setminus \Pi_{\mathfrak{N}}) = \emptyset \equiv \Pi' = \emptyset$.

- If π is unchanged with source $o \neq \mathbf{o}.\vec{\beta}$ then it cannot target any object $\mathbf{o}.\vec{\beta}'$. Hence $\sigma(o) = o$, $\sigma(\omega) = \omega)$ and $\sigma(o.\vec{\alpha}) = o.\vec{\alpha}$ and $\sigma(\omega.\vec{\gamma}) = \omega.\vec{\gamma}$. In case of $\rightleftharpoons$-paths, $\sigma(\pi) = \pi$ is the necessary counterpart $\sigma(o) \overset{\beta}{\dashrightarrow}_{/\rightleftharpoons} \sigma(\omega)$ **via** $\Pi' = \{\sigma(\pi)\}$ for a first-kind $\rightleftharpoons$-path **via** $\Pi = \{\pi\}$. And in case of new region-coupling, $\sigma(\pi) = \pi$ established the counterpart $\sigma(\omega.\beta) = \omega.\beta \rightleftharpoons o.\mu(\beta) = \sigma(o.\mu(\beta))$ **via** $\Pi' = \{\sigma(\pi)\}$.
- If π is unchanged with source $o = \mathbf{o}.\vec{\beta}$, then it can only be a dummy edge $\mathbf{o}.\vec{\beta} \xrightarrow{\beta} \mathbf{o}.\vec{\beta}.\beta$. Hence the region-coupling case does not apply. In case of $\vec{\beta} = \vec{\alpha}.\vec{\alpha}'$ with $\mu_{\mathbf{o}}(\vec{\alpha}) \in \mathbb{A}_{\perp}$, the dummy edge $\sigma(\pi) = \mathbf{r}.\mu_{\mathbf{o}}(\vec{\alpha}).\vec{\alpha}' \xrightarrow{\beta} \mathbf{r}.\mu_{\mathbf{o}}(\vec{\alpha}).\vec{\alpha}'.\beta$ is the necessary counterpart $\sigma(o) \overset{\beta}{\dashrightarrow}_{/\rightleftharpoons} \sigma(\omega)$ **via** $\Pi' = \{\sigma(\pi)\}$ for a first-kind $\rightleftharpoons$-path **via** $\Pi = \{\pi\}$. If there is no $\mu_{\mathbf{o}}(\vec{\alpha}) \in \mathbb{A}_{\perp}$ then φ is a second-kind, unchanged $\rightleftharpoons$-path.

In the induction step for part (a), consider first the case of $o \overset{\vec{\gamma}}{\dashrightarrow}_{/\rightleftharpoons} \omega$ **via** $\Pi = \Pi_1 \cup \Pi_2$ because of $\vec{\gamma} = \vec{\gamma}_1 \bullet \vec{\gamma}_2$ and two $\rightleftharpoons$-paths $o \overset{\vec{\gamma}_1}{\dashrightarrow}_{/\rightleftharpoons} q \overset{\vec{\gamma}_2}{\dashrightarrow}_{/\rightleftharpoons} \omega$ **via** Π_1 and Π_2, respectively. If the second $\rightleftharpoons$-path is of the second or third kind, then $q = \mathbf{o}.\beta_1 \ldots \beta_k$ and the first $\rightleftharpoons$-path can only be of the second kind. Consequently, $o \overset{\vec{\gamma}}{\dashrightarrow}_{/\rightleftharpoons} \omega$ is of, respectively, second or third kind. If $q \overset{\vec{\gamma}_2}{\dashrightarrow}_{/\rightleftharpoons} \omega$ is of the first kind and $o \overset{\vec{\gamma}_1}{\dashrightarrow}_{/\rightleftharpoons} q$ too, then the combination $\sigma(o) \overset{\vec{\gamma}_1}{\dashrightarrow}_{/\rightleftharpoons} \sigma(q) \overset{\vec{\gamma}_2}{\dashrightarrow}_{/\rightleftharpoons} \sigma(\omega)$ of their counterparts **via** Π_1' and Π_2', respectively, is a counterpart $o \overset{\vec{\gamma}}{\dashrightarrow}_{/\rightleftharpoons} \omega$ **via** $\Pi' = \Pi_1' \cup \Pi_2'$. Since $\sigma(\Pi_1 \setminus \Pi_{\mathfrak{N}} \setminus \Pi_{\circledast}) \equiv \Pi_1'$ and $\sigma(\Pi_2 \setminus \Pi_{\mathfrak{N}} \setminus \Pi_{\circledast}) \equiv \Pi_2'$, also $\sigma(\Pi \setminus \Pi_{\mathfrak{N}} \setminus \Pi_{\circledast}) = \sigma(\Pi_1 \cup \Pi_2 \setminus \Pi_{\mathfrak{N}} \setminus \Pi_{\circledast}) \equiv \Pi_1' \cup \Pi_2' = \Pi'$. If $o \overset{\vec{\gamma}_1}{\dashrightarrow}_{/\rightleftharpoons} q$ is of the second kind, then the combination $o \overset{\vec{\gamma}_1}{\dashrightarrow}_{/\rightleftharpoons} q \overset{\vec{\gamma}_2}{\dashrightarrow}_{/\rightleftharpoons} \omega$ to $o \overset{\vec{\gamma}}{\dashrightarrow}_{/\rightleftharpoons} \omega$ is obviously of the third kind. If $o \overset{\vec{\gamma}_1}{\dashrightarrow}_{/\rightleftharpoons} q$ is of the third kind, then it decomposes into a second-kind $o \overset{\vec{\gamma}_1'}{\dashrightarrow}_{/\rightleftharpoons} q'$ and a first-kind $q' \overset{\vec{\gamma}_2'}{\dashrightarrow}_{/\rightleftharpoons} q$. Hence $o \overset{\vec{\gamma}}{\dashrightarrow}_{/\rightleftharpoons} \omega$ can be decomposed into a second-kind kind $o \overset{\vec{\gamma}_1'}{\dashrightarrow}_{/\rightleftharpoons} q'$ and a first-kind $q' \overset{\vec{\gamma}_2'}{\dashrightarrow}_{/\rightleftharpoons} q \overset{\vec{\gamma}_2}{\dashrightarrow}_{/\rightleftharpoons} \omega$: it is a third-kind $\rightleftharpoons$-path again.

If $o \overset{\vec{\gamma}}{\dashrightarrow}_{/\rightleftharpoons} \omega$ **via** $\Pi = \Pi_1 \cup \Pi_2$ because $o.\vec{\gamma} \rightleftharpoons q.\vec{\gamma}'$ **via** Π_1 and $\varphi_2 = q \overset{\vec{\gamma}'}{\dashrightarrow}_{/\rightleftharpoons} \omega$ **via** Π_2 then the induction hypothesis guarantees a counterpart $\sigma(o.\vec{\gamma}) \rightleftharpoons \sigma(q.\vec{\gamma}')$ **via** Π_1' and that $q \overset{\vec{\gamma}'}{\dashrightarrow}_{/\rightleftharpoons} \omega$ belongs to one of three cases:

- If φ_2 has a counterpart $\sigma(q) \overset{\vec{\gamma}'}{\dashrightarrow}_{/\rightleftharpoons} \sigma(\omega)$ **via** Π_2' then either $\sigma(q) = q$, and thus $\sigma(q.\vec{\gamma}') = q.\vec{\gamma}'$, or $q = \mathbf{o}.\vec{\alpha}.\vec{\alpha}'$, and thus $\sigma(q) = \mathbf{r}.\mu_{\mathbf{o}}(\vec{\alpha}).\vec{\alpha}'$ and $\sigma(q.\vec{\gamma}') = \mathbf{r}.\mu_{\mathbf{o}}(\vec{\alpha}).\vec{\alpha}'.\vec{\gamma}'$. In both cases, counterparts $\sigma(q) \overset{\vec{\gamma}'}{\dashrightarrow}_{/\rightleftharpoons} \sigma(\omega)$ and $\sigma(o.\vec{\gamma}) \rightleftharpoons \sigma(q.\vec{\gamma}')$ combine to a first-kind $\rightleftharpoons$-path's counterpart $\sigma(o) \overset{\vec{\gamma}}{\dashrightarrow}_{/\rightleftharpoons} \sigma(\omega)$ **via** $\Pi' = \Pi_1' \cup \Pi_2'$ with $\sigma(\Pi \setminus \Pi_{\mathfrak{N}} \setminus \Pi_{\circledast}) \equiv \Pi'$.
- φ_2 can be a $\rightleftharpoons$-path $q = \mathbf{o}.\vec{\beta} \overset{\vec{\gamma}'}{\dashrightarrow}_{/\rightleftharpoons} \mathbf{o}.\vec{\beta}.\vec{\gamma}' = \omega$ **via** $\Pi_2 \subseteq \Pi_{\circledast}$. On one hand, $\mu_{\mathbf{o}}(\vec{\beta}') \notin \mathbb{A}_{\perp}$ for all proper prefixes $\vec{\beta}'$ of $\vec{\beta}$. On the other hand, for some prefix

$\vec{\beta}'$ the region-coupling guarantees $\mu_{\mathbf{o}}(\vec{\beta}') \in \mathbb{A}_{\perp}$. Hence it must be $\mu_{\mathbf{o}}(\vec{\beta}) \in \mathbb{A}_{\perp}$. Therefore $\sigma(q.\vec{\gamma}') = \sigma(\omega) = \mathbf{r}.\mu_{\mathbf{o}}(\vec{\beta})$. Hence the region-coupling's counterpart $\sigma(o.\vec{\gamma}) \rightleftharpoons \sigma(q.\vec{\gamma}') = \mathbf{r}.\mu_{\mathbf{o}}(\vec{\beta})$ and trivial $\mathbf{r}.\mu_{\mathbf{o}}(\vec{\beta}) \overset{\epsilon}{\dashrightarrow}_{\rightleftharpoons} \mathbf{r}.\mu_{\mathbf{o}}(\vec{\beta})$ **via** $\emptyset$ combine to a first-kind $\rightleftharpoons$-path's counterpart $\sigma(o) \overset{\vec{\gamma}}{\dashrightarrow}_{\rightleftharpoons} \sigma(\omega)$ **via** $\Pi' = \Pi'_1 \cup \emptyset$. Since $\Pi_2 \subseteq \Pi_{\circledast}$, $\Pi \backslash \Pi_{\mathcal{N}} \backslash \Pi_{\circledast} = \Pi_1$, so that $\sigma(\Pi_1 \backslash \Pi_{\mathcal{N}} \backslash \Pi_{\circledast}) \equiv \Pi'_1$ means $\sigma(\Pi \backslash \Pi_{\mathcal{N}} \backslash \Pi_{\circledast}) \equiv \Pi'$.

- φ_2 can be the combination of a second-kind $\rightleftharpoons$-path $\varphi_3 = \mathbf{o}.\vec{\beta} \overset{\vec{\gamma}'}{\dashrightarrow}_{\rightleftharpoons} \mathbf{o}.\vec{\beta}.\vec{\gamma}'$ **via** $\Pi_3 \subseteq \Pi_{\circledast}$ and a first-kind $\rightleftharpoons$-path $\varphi_4 = \mathbf{o}.\vec{\beta}.\vec{\gamma}' \overset{\vec{\gamma}''}{\dashrightarrow}_{\rightleftharpoons} \omega$ with $q = \mathbf{o}.\vec{\beta}$ such that $\vec{\gamma} = \vec{\gamma}'.\vec{\gamma}''$ and $\mu_{\mathbf{o}}(\vec{\beta}.\vec{\gamma}') \in \mathbb{A}$, and with $\Pi_2 = \Pi_3 \cup \Pi_4$. Hence the region-coupling's counterpart $\sigma(o.\vec{\gamma}) \rightleftharpoons \sigma(q.\vec{\gamma}') = \mathbf{r}.\mu_{\mathbf{o}}(\vec{\beta})$ and φ_3's counterpart $\sigma(q.\vec{\gamma}) \overset{\vec{\gamma}''}{\dashrightarrow}_{\rightleftharpoons} \sigma(\omega)$ **via** Π'_3 combine to a first-kind $\rightleftharpoons$-path's counterpart $\sigma(o) \overset{\vec{\gamma}}{\dashrightarrow}_{\rightleftharpoons} \sigma(\omega)$ **via** $\Pi'_1 \cup \Pi'_3$. Since $\Pi_3 \subseteq \Pi_{\circledast}$, $\Pi \setminus \Pi_{\mathcal{N}} \setminus \Pi_{\circledast} = \Pi_1 \cup \Pi_4$, so that $\sigma(\Pi_1 \setminus \Pi_{\mathcal{N}} \setminus \Pi_{\circledast}) \equiv \Pi'_1$ and $\sigma(\Pi_4 \setminus \Pi_{\mathcal{N}} \setminus \Pi_{\circledast}) \equiv \Pi'_4$ means $\sigma(\Pi \setminus \Pi_{\mathcal{N}} \setminus \Pi_{\circledast}) \equiv \Pi'$.

In the induction step of part (b), the symmetry case of new region-couplings is trivial.

In another case, $o.\vec{\alpha} \rightleftharpoons \omega.\vec{\gamma}$ **via** $\Pi = \Pi_1 \cup \Pi_2$ because $o.\vec{\alpha} \rightleftharpoons q.\vec{\gamma}'$ **via** Π_1 and $q.\vec{\gamma}' \rightleftharpoons \omega.\vec{\gamma}$ **via** Π_2. If both are new, then the induction hypothesis guarantees counterparts $\sigma(o.\vec{\alpha}) \rightleftharpoons \sigma(q.\vec{\gamma}')$ **via** Π'_1, and $\sigma(q.\vec{\gamma}') \rightleftharpoons \sigma(\omega.\vec{\gamma})$ **via** Π'_2, which combine to counterpart $\sigma(o.\vec{\alpha}) \rightleftharpoons \sigma(\omega.\vec{\gamma})$ **via** $\Pi' = \Pi'_1 \cup \Pi'_2$ with $\sigma(\Pi \setminus \Pi_{\mathcal{N}} \setminus \Pi_{\circledast}) \equiv \Pi'$. The same works with an unchanged first region-coupling if $\sigma(o.\vec{\alpha}) = o.\vec{\alpha}$ and $\sigma(q.\vec{\gamma}') = q.\vec{\gamma}'$, and works with an unchanged second region-coupling if $\sigma(q.\vec{\gamma}') = q.\vec{\gamma}'$ and $\sigma(\omega.\vec{\gamma}) = \omega.\vec{\gamma}$. Consider the case that the first region-coupling is new and the second is old with $q.\vec{\gamma}' = \mathbf{o}.\vec{\beta}$ or $\omega.\vec{\gamma} = \mathbf{o}.\vec{\beta}$ (the reverse case follows by symmetry). Since there are no old ownership paths to fresh $\mathbf{o}$ and its region objects, an *old* region-coupling connecting their regions can only be a trivial region-coupling $q.\vec{\gamma}' = (\mathbf{o}.\alpha_1 \ldots \alpha_i).\alpha_{i+1} \ldots \alpha_n \rightleftharpoons (\mathbf{o}.\alpha_1 \ldots \alpha_j).\alpha_{j+1} \ldots \alpha_n = \omega.\vec{\gamma}$ **via** Π_2 obtained from dummy association paths $\mathbf{o}.\vec{\alpha} \overset{\vec{\gamma}}{\dashrightarrow}_{\rightleftharpoons} \mathbf{o}.\vec{\alpha}.\vec{\gamma}$ **via** $\Pi_2 = \{\mathbf{o}.\vec{\alpha} \xrightarrow{\gamma_1} \mathbf{o}.\vec{\alpha}.\gamma_1, \mathbf{o}.\vec{\alpha}.\gamma_1 \xrightarrow{\gamma_2} \mathbf{o}.\vec{\alpha}.\gamma_1.\gamma_2, \ldots\}$. But then $\sigma(q.\vec{\gamma}') = \sigma(\omega.\vec{\gamma})$, so that the new region-coupling's counterpart guaranteed by the induction hypothesis is the desired $\sigma(o.\vec{\alpha}) \rightleftharpoons \sigma(q.\vec{\gamma}') = \sigma(\omega.\vec{\gamma})$ **via** $\Pi' = \Pi'_1$. Since $\Pi_2 \subseteq \Pi_{\circledast}$, $\sigma(\Pi \setminus \Pi_{\mathcal{N}} \setminus \Pi_{\circledast}) = \sigma(\Pi_1 \setminus \Pi_{\mathcal{N}} \setminus \Pi_{\circledast}) \equiv \Pi'_1 = \Pi'$.

If $o.\vec{\alpha}.\vec{\gamma} \rightleftharpoons \omega.\vec{\gamma}$ **via** Π, follows from $\varphi = o \overset{\vec{\alpha}}{\dashrightarrow}_{\rightleftharpoons} \omega$ **via** Π then the induction hypothesis guarantees that φ belongs to one of three cases:

- If φ has a counterpart $\sigma(o) \overset{\vec{\alpha}}{\dashrightarrow}_{\rightleftharpoons} \sigma(\omega)$ **via** Π'. It entails the necessary counterpart $\sigma(o).\vec{\alpha}.\vec{\gamma} = \sigma(o.\vec{\alpha}.\vec{\gamma}) \rightleftharpoons \sigma(\omega).\vec{\gamma} = \sigma(\omega.\vec{\gamma})$ **via** Π'.
- φ can be a $\rightleftharpoons$-path $o = \mathbf{o}.\vec{\beta} \overset{\vec{\gamma}'}{\dashrightarrow}_{\rightleftharpoons} \mathbf{o}.\vec{\beta}.\vec{\gamma}' = \omega$ based on dummy edges in $\Pi_{\circledast}$. Then it, and consequently $o.\vec{\alpha}.\vec{\gamma} \rightleftharpoons \omega.\vec{\gamma}$ is not new.
- $o \overset{\vec{\alpha}}{\dashrightarrow}_{\rightleftharpoons} \omega$ can be combined from a second-kind $\rightleftharpoons$-path $\varpi_1 = o \overset{\vec{\alpha}_1}{\dashrightarrow}_{\rightleftharpoons} o.\vec{\alpha}_1$ **via** Π_1 and a first-kind $\rightleftharpoons$-path $\varphi_2 = o.\vec{\alpha}_1 \overset{\vec{\alpha}_2}{\dashrightarrow}_{\rightleftharpoons} \omega$ **via** Π_2 with $\vec{\alpha} = \vec{\alpha}_1.\vec{\alpha}_2$, and $o = \mathbf{o}.\vec{\beta}$ such that $\mu_{\mathbf{o}}(\vec{\beta}.\vec{\alpha}_1) \in \mathbb{A}$. In this case, φ_2's counterpart $\sigma(o.\vec{\alpha}_1) \overset{\vec{\alpha}_2}{\dashrightarrow}_{\rightleftharpoons} \sigma(\omega)$ established the right region-coupling $\sigma(o.\vec{\alpha}_1).\vec{\alpha}_2.\vec{\gamma} = \sigma(o.\vec{\alpha}_1.\vec{\alpha}_2.\vec{\gamma}) = \sigma(o.\vec{\alpha}.\vec{\gamma}) \rightleftharpoons \sigma(\omega).\vec{\gamma} = \sigma(\omega.\vec{\gamma})$ **via** $\Pi' = \Pi'_2$ for new $o.\vec{\alpha}.\vec{\gamma} \rightleftharpoons \omega.\vec{\gamma}$ **via** Π. Since $\Pi_1 \subseteq \Pi_{\circledast}$, $\sigma(\Pi \setminus \Pi_{\mathcal{N}} \setminus \Pi_{\circledast}) = \sigma(\Pi_2 \setminus \Pi_{\mathcal{N}} \setminus \Pi_{\circledast}) \equiv \Pi'_2 = \Pi'$. ■

Lemma 16 Consider elementary mode conversion $\mathfrak{g}' = \mathfrak{g} \oplus \mathbf{c} \xrightarrow{\mu} \mathbf{o} \ominus \mathbf{c} \xrightarrow{\mu'} \mathbf{o}$, i.e., the substitution of an edge $h'_\mathbf{o} = \mathbf{c} \xrightarrow{\mu'} \mathbf{o}$ for an edge $h_\mathbf{o} = \mathbf{c} \xrightarrow{\mu} \mathbf{o}$ with $\mu \leq^1_\mathrm{m} \mu'$. In $\mathfrak{g}'^{\circledast}$,

a) all $\rightleftharpoons$-paths $o \overset{\vec{\gamma}}{\dashrightarrow}_{/\rightleftharpoons} \omega$ **via** Π have a precursor $o \overset{\vec{\gamma}}{\dashrightarrow}_{/\rightleftharpoons} \omega$ **via** $\Pi[h_\mathbf{o}/h'_\mathbf{o}]$, and
b) all region-couplings $\omega.\vec{\gamma} \rightleftharpoons o.\vec{\alpha}$ **via** Π have a precursor $\omega.\vec{\gamma} \rightleftharpoons o.\vec{\alpha}$ **via** $\Pi[h_\mathbf{o}/h'_\mathbf{o}]$.

Proof: Lemma 7 guarantees that all potential access paths $\pi \in PAP_{\mathfrak{g}'^{\circledast}}(o, \mu, \omega)$ in $\mathfrak{g}'^{\circledast}$ are unchanged, internally new, or initially new. They all have a precursor $\pi' = \pi[h_\mathbf{o}/h'_\mathbf{o}] \in PAP_{\mathfrak{g}^{\circledast}}(o, \mu', \omega)$ of the same mode $\mu' = \mu$ or of a directly compatible mode $\mu' \leq^1_\mathrm{m} \mu$. By the definition of $\leq^1_\mathrm{m}$, a mode $\mu \geq^1_\mathrm{m} \mu'$ cannot be an association mode. Hence all association paths π have a precursor π' of the same mode. The base case of new $\rightleftharpoons$-paths φ differs at most in the path-base: φ is now **via** $\{\pi\}$ instead of **via** $\{\pi'\}$. Also by the definition of $\leq^1_\mathrm{m}$, a mode $\mu \geq^1_\mathrm{m} \mu'$ can be a `free` or `rep` mode only in case of $\mu' =$ `free<`δ`>` $\leq^1_\mathrm{m}$ `rep<`δ`>` $= \mu$. Hence all ownership paths π have a precursor π' that was already an ownership path and had the same correlations δ. Hence the base case of new region-couplings differs at most in the path-base: $o.\mathfrak{r} \rightleftharpoons \omega.\vec{\gamma}$ is **via** π now instead of **via** π'. By induction therefore all $\rightleftharpoons$-paths and region-coupling **via** π have a precursor **via** $\Pi[h_\mathbf{o}/h'_\mathbf{o}]$. ■

Lemma 17 Consider result return $\mathfrak{g}' = \mathfrak{g} \ominus \mathbf{r} \xrightarrow{\mu_\mathbf{o}} \mathbf{o} \ominus \mathbf{s} \xrightarrow{\mu_\mathbf{r}} \mathbf{r} \oplus \mathbf{s} \xrightarrow{\mu_\mathbf{r} \circ \mu_\mathbf{o}} \mathbf{o}$, i.e., the substitution of the imported edge $h'_\mathbf{o} = \mathbf{s} \xrightarrow{\mu_\mathbf{r} \circ \mu_\mathbf{o}} \mathbf{o}$ for the exported edge $h_\mathbf{o} = \mathbf{r} \xrightarrow{\mu_\mathbf{o}} \mathbf{o}$ and the call-link $h_\mathbf{r} = \mathbf{s} \xrightarrow{\mu_\mathbf{r}} \mathbf{r}$. In $\mathfrak{g}'^{\circledast}$,

a) all $\rightleftharpoons$-paths $o \overset{\vec{\gamma}}{\dashrightarrow}_{/\rightleftharpoons} \omega$ **via** Π have a precursor $o \overset{\vec{\gamma}}{\dashrightarrow}_{/\rightleftharpoons} \omega$ **via** $\sigma(\Pi)$, and
b) all region-couplings $\omega.\vec{\gamma} \rightleftharpoons o.\vec{\alpha}$ **via** Π have a precursor $\omega.\vec{\gamma} \rightleftharpoons o.\vec{\alpha}$ **via** $\sigma(\Pi)$,

where substitution σ is $[h_\mathbf{r} \bullet h_\mathbf{o}/h'_\mathbf{o}, h_\mathbf{o}^{-1} \bullet h_\mathbf{r}^{-1}/h'^{-1}_\mathbf{o}]$ in case of $\mu_\mathbf{r} \circ \mu_\mathbf{o} =$ `co<>`, and $[h_\mathbf{r} \bullet h_\mathbf{o}/h'_\mathbf{o}]$ otherwise.

Proof: Lemma 8 guarantees that all potential access paths in $\mathfrak{g}'^{\circledast}$ are unchanged, internally new or initially new. If these are *association* paths $\pi \in PAP_{\mathfrak{g}'^{\circledast}}(o, \beta\texttt{<>}, \omega)$, they have a precursor $\pi' = \sigma(\pi) \in PAP_{\mathfrak{g}^{\circledast}}(o, \beta\texttt{<>}, \omega)$. Hence the base case of new $\rightleftharpoons$-paths φ differs at most in the path-base: φ is now **via** $\{\pi\}$ instead of **via** $\{\sigma(\pi)\}$. Similarly most ownership paths π that establish a region-coupling **via** $\{\pi\}$ have a precursor $\sigma(\pi)$ that established it already in $\mathfrak{g}^{\circledast}$ **via** $\{\sigma(\pi)\}$. The base case of a really new region-coupling could only come from an initially new ownership path $\pi \in PAP_{\mathfrak{g}'^{\circledast}}(o, \mu, \omega)$ without precursor $\sigma(\pi)$, i.e., whose counterpart $\mathfrak{expo}(\pi)$'s mode μ' with $\mu = \mu_\mathbf{r} \circ \mu'$ is not a `co`- or association mode, but a `free` mode. Then π is a `free` path which establishes the region-coupling $\omega.\gamma \rightleftharpoons \mathbf{s}.\mu(\gamma)$ in $\mathfrak{g}'^{\circledast}$ in two cases:

- $\mu(\gamma) = \alpha$, i.e., μ has a correlation γ=α`<>`. Since π's mode μ is the adaption $\mu_\mathbf{r} \circ \mu'$ of its counterpart's non-`co` and non-association mode μ', it can contain the correlation γ=α`<>` only because $\mathfrak{expo}(\pi)$'s mode μ' contains γ=β`<>` and $\mu_\mathbf{r}$ contains β=α`<>`. Correlations in μ' mean that it cannot be a `co`- or association mode, and thus $\mu_\mathbf{r} \circ \mu'$ can be a `free` or `rep` mode only if μ' is a `free` or `rep` mode. That is, $\mu' = \texttt{free<}\ldots,\gamma{=}\beta\texttt{<>},\ldots\texttt{>}$ or $\texttt{rep<}\ldots,\gamma{=}\beta\texttt{<>},\ldots\texttt{>}$. Hence, on one hand, `free` counterpart $\mathfrak{expo}(\pi)$ established $\omega.\gamma \rightleftharpoons \mathbf{r}.\beta$ **via** $\{\mathfrak{expo}(\pi)\}$ in $\mathfrak{g}^{\circledast}$. On the other hand, a *legal* `return` step means that in this case $\mu_{\mathbf{r}}$ is a `free` or `rep` mode. That is, $\mu_{\mathbf{r}} = \texttt{free<}\ldots,\beta{=}\alpha\texttt{<>},\ldots\texttt{>}$ or $\texttt{rep<}\ldots,\beta{=}\alpha\texttt{<>},\ldots\texttt{>}$. Hence $h_{\mathbf{r}}$ established $\mathbf{r}.\beta \rightleftharpoons \mathbf{s}.\alpha$ **via** $\{h_{\mathbf{r}}\}$ in $\mathfrak{g}^{\circledast}$. In combination, we have $\omega.\gamma \rightleftharpoons \mathbf{s}.\mu(\gamma) = \mathbf{s}.\alpha$ **via** $\Pi' = \{h_{\mathbf{r}}, \mathfrak{expo}(\pi)\}$.

- $\mu(\gamma) = \bot$, i.e., μ has no correlation $\gamma{=}\mu''$. Since π's mode μ is the adaption $\mu_{\mathbf{r}} \circ \mu'$ of its counterpart's non-`co` and non-association mode μ', it can contain no correlation $\gamma{=}\mu''$ only if $\mathfrak{expo}(\pi)$'s mode μ' contains no correlation $\gamma{=}\mu'''$. Consequently, `free` counterpart $\mathfrak{expo}(\pi)$ established $\omega.\gamma \rightleftharpoons \mathbf{r}.\bot$ **via** $\{\mathfrak{expo}(\pi)\}$ in $\mathfrak{g}^{\circledast}$. But since also $\mathbf{r}.\bot \rightleftharpoons \mathbf{s}.\bot$ **via** $\emptyset$, this means by transitivity $\omega.\gamma \rightleftharpoons \mathbf{s}.\bot = \mathbf{s}.\mu(\gamma)$ **via** $\Pi' = \{\mathfrak{expo}(\pi)\}$.

Hence there is a precursor for $\omega.\gamma \rightleftharpoons \mathbf{s}.\bot = \mathbf{s}.\mu(\gamma)$ **via** $\Pi = \{\pi\}$ such that $\Pi' \Subset \sigma(\Pi)$ since $h_{\mathbf{r}} \bullet \mathfrak{expo}(\pi) = \sigma(\pi)$. ■

Again, the case of {call} is the most complex. Several technical lemmas will be used for the proof; they are supplemented in the next subsection. In §6.3.1, $\vec{\alpha}$-bridges were used for the description of the new potential access paths. They consisted of an initial $\rightleftharpoons$-path $o \overset{\vec{\alpha}}{\dashrightarrow}\!/_{\rightleftharpoons}\ \omega_0$ followed by a series of triples $\langle \pi_i, \pi_i', \varphi_i' \rangle$ made of a pair of ownership paths π_i and π_i' and another $\rightleftharpoons$-path φ_i'. The description of the new $\rightleftharpoons$-paths will require a generalization to bridges where an initial $\rightleftharpoons$-path $o \overset{\vec{\alpha}}{\dashrightarrow}\!/_{\rightleftharpoons}\ \omega_0$ is followed by a series of *quadruples* $\langle \varphi_i', \pi_i', \pi_i, \varphi_i \rangle$, which will be called a "$\vec{\alpha}$-qbridge." And the description of the new region-couplings will require bridges made just of a series of *quadruples* $\langle \varphi_i', \pi_i', \pi_i, \varphi_i \rangle$, which will be called a "reservation qbridge."

Definition 15 A ***reservation qbridge*** from o to ω in supply substeps is a series of quadruples $\langle \varphi_i', \pi_i', \pi_i, \varphi_i \rangle$ from $i = 1$ to $n \geq 1$, each combining a backward ownership reservation $\langle \pi_i', \varphi_i' \rangle$ and a forward ownership reservations $\langle \pi_i, \varphi_i \rangle$ where $\pi_i' = \mathbf{s} \overset{\hat{\mu}_i}{\dashrightarrow} q_i'.\vec{\alpha}_i'$ and $\varphi_i' = q_i' \overset{\vec{\alpha}_i'}{\dashrightarrow}\!/_{\rightleftharpoons}\ \omega_i$, and where $\pi_i = \mathbf{s} \overset{\hat{\mu}_i}{\dashrightarrow} q_i.\vec{\alpha}_i$, and $\varphi_i = q_i \overset{\vec{\alpha}_i}{\dashrightarrow}\!/_{\rightleftharpoons}\ \omega_{i+1}$ such that $\omega_1 = o$ and $\omega_{n+1} = \omega$. The qbridge is **via** $\bigcup_{i=1}^{n} \Pi_i \cup \Pi_i' \cup \{\pi_i, \pi_i'\}$ if the φ_i is **via** Π_i and the φ_i' is **via** Π_i'.

An $\vec{\alpha}$-***qbridge*** from o to ω is an initial $\rightleftharpoons$-path $o \overset{\vec{\alpha}}{\dashrightarrow}\!/_{\rightleftharpoons}\ \omega_1$ **via** Π_0 with $\omega_1 = \omega$ or a reservation bridge from ω_1 to ω **via** Π'. from $i = 1$ to some $n \geq 0$, such that $\omega_{n+1} = \omega$. The qbridge is, respectively, **via** Π_0 or **via** $\Pi_0 \cup \Pi'$.

In case of $\mu_{\mathbf{o}} = \texttt{co<>}$, for each quadruple in both kinds of qbridges there is a $\vec{\alpha}$ such that each of the two ownership paths π_i and π_i' in it has shape $\mathbf{s} \xrightarrow{\mu_{\mathbf{r}} \circ \mu_{\mathbf{o}}} \mathbf{o} \xrightarrow{\texttt{co}}{}^{*} \bullet \overset{\vec{\alpha}}{\dashrightarrow} \bullet$ or $\mathbf{s} \xrightarrow{\mu_{\mathbf{r}}} \mathbf{r} \xrightarrow{\texttt{co}}{}^{*} \bullet \overset{\vec{\alpha}}{\dashrightarrow} \bullet$. If $\mu_{\mathbf{o}} \neq \texttt{co<>}$, one of the two ownership paths π_i and π_i' in each quadruple in the qbridges has shape $\mathbf{s} \xrightarrow{\mu_{\mathbf{r}} \circ \mu_{\mathbf{o}}} \mathbf{o} \xrightarrow{\texttt{co}}{}^{*} \bullet \overset{\vec{\alpha}.\vec{\gamma}}{\dashrightarrow} \bullet$ for some $\vec{\alpha}$ and $\vec{\alpha}'$ with $\mu_{\mathbf{o}}(\vec{\alpha}) \in \mathbb{A}$, while the other has shape $\mathbf{s} \xrightarrow{\mu_{\mathbf{r}}} \mathbf{r} \xrightarrow{\texttt{co}}{}^{*} \bullet \overset{\mu_{\mathbf{o}}(\vec{\alpha}).\vec{\gamma}}{\dashrightarrow} \bullet$.

Lemma 18 Assume the reserved ownership assumption in $\mathfrak{g}^{\circledast}$ and $\mathfrak{g} = ogr(e, \vec{\eta}, \mathfrak{s})$. Consider parameter supply $\mathfrak{g}' = \mathfrak{g} \oplus \mathbf{r} \xrightarrow{\mu_{\mathbf{o}}} \mathbf{o} \ominus \mathbf{s} \xrightarrow{\mu_{\mathbf{r}} \circ \mu_{\mathbf{o}}} \mathbf{o}$, i.e., the substitution of a received handle $h_{\mathbf{o}} = \mathbf{r} \xrightarrow{\mu_{\mathbf{o}}} \mathbf{o}$ for a sent handle $h'_{\mathbf{o}} = \mathbf{s} \xrightarrow{\mu_{\mathbf{r}} \circ \mu_{\mathbf{o}}} \mathbf{o}$ in the presence of a call-link $h_{\mathbf{r}} = \mathbf{s} \xrightarrow{\mu_{\mathbf{r}}} \mathbf{r}$.

a) For each *non-trivial* $\rightleftharpoons$-path $o \dashrightarrow^{\vec{\gamma}}_{\rightleftharpoons} \omega$ **via** Π in $\mathfrak{g}'^{\circledast}$, there was a $\vec{\gamma}$-qbridge from o to ω in $\mathfrak{g}^{\circledast}$.
b) For each *new* region-coupling $o.\vec{\alpha} \rightleftharpoons \omega.\vec{\gamma}$ **via** Π in $\mathfrak{g}'^{\circledast}$, there was a reservation qbridge from $o.\vec{\alpha}$ to $\omega.\vec{\gamma}$ in $\mathfrak{g}^{\circledast}$.

Both qbridges are **via** Π' such that $\Pi' \cup \Pi_{\mathbf{o}} \equiv \Pi \cup \Pi_{\Lambda} \cup \Pi_{\circledast}$ where $\Pi_{\circledast}$ is some set of dummy edges, where $\Pi_{\mathbf{o}} \subseteq \{h_{\mathbf{o}}, h_{\mathbf{o}}^{-1}\}$, and where $\Pi_{\Lambda} = \{h_{\mathbf{r}}, h'_{\mathbf{o}}\}$ if $\mu_{\mathbf{o}}(\vec{\alpha}) \in \mathbb{A}$ for some $\vec{\alpha}$, and $\Pi_{\Lambda} = \{h_{\mathbf{r}}, h'_{\mathbf{o}}\}$ or $\{h_{\mathbf{r}}^{-1}, h'^{-1}_{\mathbf{o}}\}$ or $\{h_{\mathbf{r}}, h_{\mathbf{r}}^{-1}, h'_{\mathbf{o}}, h'^{-1}_{\mathbf{o}}\}$ if $\mu_{\mathbf{o}} = \mu_{\mathbf{r}} = \mathsf{co}\langle\rangle$, and $\Pi_{\Lambda} = \emptyset$ otherwise.

Proof by simultaneous induction on the definition of $\rightleftharpoons$-paths and region-couplings: Lemma 9 guarantees that all potential access paths in $\mathfrak{g}'^{\circledast}$ are unchanged, internally new, initially new, or co-closure paths. In the base case of part (a), an *association* path $\pi \in PAP_{\mathfrak{g}'^{\circledast}}(o, \beta\langle\rangle, \omega)$ established the $\rightleftharpoons$-path $o \dashrightarrow^{\vec{\gamma}}_{\rightleftharpoons} \omega$ **via** $\Pi = \{\pi\}$ with $\vec{\gamma} = \beta$. And in the base case of part (b), an *ownership* path $\pi \in PAP_{\mathfrak{g}'^{\circledast}}(o, \mu, \omega)$ with $\mu(\beta) \in \mathbb{A}_{\perp}$ established the new region-coupling $o.\mu(\beta) \rightleftharpoons \omega.\beta$ **via** $\Pi = \{\pi\}$.

(1) If π is an *unchanged* or *internally-only new* path then it has a precursor $\pi' = \pi$ or $\pi' = \pi[h_{\mathbf{r}}^{-1} \bullet h'_{\mathbf{o}}/h_{\mathbf{o}}, h'^{-1}_{\mathbf{o}} \bullet h_{\mathbf{r}}/h_{\mathbf{o}}^{-1}]$. In part (a), π' is a $\rightleftharpoons$-path $o \dashrightarrow^{\vec{\gamma}}_{\rightleftharpoons} \omega$, i.e., a $\beta = \vec{\gamma}$-qbridge without reservation qbridge, but path-base $\Pi' = \{\pi'\}$. In part (b), π' established region-coupling $\omega.\beta \rightleftharpoons o.\mu(\beta)$ **via** $\Pi' = \{\pi'\}$. Obviously, $\Pi' \cup \Pi_{\mathbf{o}} \equiv \Pi \cup \Pi_{\Lambda}$ for a subset Π_{Λ} of $\{h_{\mathbf{r}}, h_{\mathbf{r}}^{-1}, h'_{\mathbf{o}}, h'^{-1}_{\mathbf{o}}\}$ and a subset $\Pi_{\mathbf{o}}$ of $\{h_{\mathbf{o}}, h_{\mathbf{o}}^{-1}\}$.

(2) If π is *internally really new* then $\pi = \pi_1 \bullet \pi_2$, there is a witness $\mathfrak{wit}(\pi) = \pi_1 \bullet \pi'_2 \in PAP_{\mathfrak{g}^{\circledast}}(o, \mu, q_0.\vec{\alpha}_0)$ and a $\vec{\alpha}_0$-bridge to ω **via** Π'. In part (a), $\mathfrak{wit}(\pi) = q \dashrightarrow^{\beta\langle\rangle} q_0.\vec{\alpha}_0$ and the initial $\rightleftharpoons$-path $q_0 \dashrightarrow^{\vec{\alpha}_0}_{\rightleftharpoons} \omega_0$ **via** Π'_0 of the $\vec{\alpha}_0$-bridge implied the $\rightleftharpoons$-path $o \dashrightarrow^{\beta}_{\rightleftharpoons} \omega_0$ **via** $\Pi'_0 \cup \{\mathfrak{wit}(\pi)\}$ (Lemma 12). And the triples $\langle \pi'_i, \pi_i, \varphi_i \rangle$ of the $\vec{\alpha}_0$-bridge can be transformed into the quadruples $\langle \varphi'_i, \pi'_i, \pi_i, \varphi_i \rangle$ of a reservation qbridge by adding $\varphi'_i = \omega_{i-1} \dashleftarrow^{\epsilon}_{\rightleftharpoons} \omega_{i-1}.\epsilon$ **via** $\emptyset$ for each $\pi'_i = \mathbf{s} \dashrightarrow^{\hat{\mu}_i} \omega_{i-1}$. Both together are the desired $\vec{\gamma} = \beta$-qbridge from o to ω **via** $\Pi'' = \Pi' \cup \{\mathfrak{wit}(\pi)\} \equiv \Pi' \cup \{\pi_1, \pi'_2\}$. Since internally really new π guarantees $\Pi' \cup \Pi_{\mathbf{o}} \equiv \{\pi_2\} \cup \Pi_{\Lambda} \cup \Pi_{\circledast}$ and since $\pi = \pi_1 \bullet \pi_2$, we have $\Pi'' \cup \Pi_{\mathbf{o}} \equiv \{\pi\} \cup \Pi_{\Lambda} \cup \Pi_{\circledast} \cup \{\pi'_2\}$. That is, $\Pi'' \cup \Pi_{\mathbf{o}} \equiv \Pi \cup \Pi_{\Lambda} \cup (\Pi_{\circledast} \cup \{\pi'_2\})$.

In part (b), $\mu(\beta) = \gamma \in \mathbb{A}$. The $\vec{\alpha}_0$-bridge is extended along dummy edge $h_\beta = \omega \xrightarrow{\beta} \omega.\beta$ to a $\vec{\alpha}_0.\beta$-bridge from q_0 to $\omega.\beta$ **via** $\Pi'' \equiv \Pi' \cup \{h_\beta\} \cup \Pi_\beta$ (Lemma 14). Its initial $\rightleftharpoons$-path $q'_0 \dashrightarrow^{\vec{\alpha}_0.\beta}_{\rightleftharpoons} \omega'_0$ **via** Π''_0 implies $(q'_0.\vec{\alpha}_0.\beta).\epsilon \rightleftharpoons \omega'_0.\epsilon$ **via** Π''_0. The ownership path $\mathfrak{wit}(\pi) = q \dashrightarrow^{\mu} q_0.\vec{\alpha}_0$ implies the matching $o.\gamma \rightleftharpoons (q_0.\vec{\alpha}_0).\beta$ **via** $\{\mathfrak{wit}(\pi)\}$. In the case that the $\vec{\alpha}_0.\beta$-bridge contains no triple series, i.e., $\omega'_0 = \omega.\beta$, these couplings mean $o.\gamma \rightleftharpoons (\omega.\beta).\epsilon$ **via** $\Pi''_0 \cup \{\mathfrak{wit}(\pi)\} = \Pi'' \cup \{\mathfrak{wit}(\pi)\}$. Otherwise, dummy edge $h_o = o \xrightarrow{\gamma} o.\gamma$ modulo these couplings is $\varphi' = o.\gamma \dashleftarrow^{\epsilon}_{\rightleftharpoons} \omega'_0$ **via** $\Pi''_0 \cup \{\mathfrak{wit}(\pi), h_o\}$. It extends the first triple $\langle \pi'_1, \pi_1, \varphi_1 \rangle$ in the $\vec{\alpha}_0.\beta$-bridge to a quadruple. In conjunction with the remaining triples transformed to quadruples, we get the desired reservation qbridge from $o.\gamma$ to $\omega.\beta$ **via** $\Pi''' = \Pi'' \cup \{\mathfrak{wit}(\pi), h_o\} \equiv \Pi' \cup \{h_\beta\} \cup \Pi_\beta \cup \{\mathfrak{wit}(\pi), h_o\}$.

Since internally really new π guarantees $\Pi' \cup \Pi_{\mathbf{o}} \equiv \{\pi_2\} \cup \Pi_\Lambda \cup \Pi_\circledast$ and since $\mathfrak{wit}(\pi) = \pi_1 \bullet \pi_2'$ and $\pi = \pi_1 \bullet \pi_2$, we have $\Pi''' \cup \Pi_{\mathbf{o}} \equiv \{\pi_2\} \cup \Pi_\Lambda \cup \Pi_\circledast \cup \{h_\beta\} \cup \Pi_\beta \cup \{\mathfrak{wit}(\pi), h_o\} = \{\pi_2\} \cup \Pi_\Lambda \cup (\Pi_\circledast \cup \{h_\beta, h_o\} \cup \Pi_\beta) \cup \{\pi_1 \bullet \pi_2'\} \equiv \{\pi\} \cup \Pi_\Lambda \cup (\Pi_\circledast \cup \{h_\beta, \pi_2', h_o\} \cup \Pi_\beta)$.

(3) If π is *initially new*, then $o = \mathbf{r}$ and there is a witness $\mathfrak{wit}(\pi) = h_{\mathbf{o}}' \bullet \pi_1 \bullet \pi_2' = \mathbf{s} \overset{\mu_{\mathbf{r}} \circ \beta <>}{\dashrightarrow} q_0.\vec{\alpha}_0$ and a $\vec{\alpha}_0$-bridge to ω **via** Π'. In part (a), Lemma 13 allows two cases for initially new *association* path π:

- There is a $\rightleftharpoons$-path $\mathbf{r} \overset{\beta}{\dashrightarrow}_{/\rightleftharpoons} \omega_0$ **via** $\Pi_0' \cup \{h_{\mathbf{r}}, \mathfrak{wit}(\pi)\}$ to the object ω_0 from which the series of triples in π's $\vec{\alpha}_0$-bridge leads to ω. This $\rightleftharpoons$-path, and the triple series transformed to quadruples as above, constitute a β-qbridge from $o = \mathbf{r}$ to ω **via** $\Pi''' = \Pi' \cup \{h_{\mathbf{r}}, \mathfrak{wit}(\pi)\} \equiv \Pi' \cup \{h_{\mathbf{r}}, h_{\mathbf{o}}' \bullet \pi_1 \bullet \pi_2'\} \equiv \Pi' \cup \{\pi_1\} \cup \Pi_\Lambda \cup \Pi_\circledast'$ with $\Pi_\Lambda = \{h_{\mathbf{r}}, h_{\mathbf{o}}'\}$ and $\Pi_\circledast' = \{\pi_2'\}$.
- There is an ownership path-pair $\langle h_{\mathbf{r}} \bullet h_\beta, \mathfrak{wit}(\pi)\rangle = \mathbf{r}.\beta \overset{\hat{\mu}}{\dashleftarrow} \mathbf{s} \overset{\hat{\mu}}{\dashrightarrow} q_0.\vec{\alpha}_0$ with $h_\beta = \mathbf{r} \xrightarrow{\beta} \mathbf{r}.\beta$ in front of π's $\vec{\alpha}_0$-bridge from q_0 to ω. Add the initial $\rightleftharpoons$-path $\varphi_0 = q_0 \overset{\vec{\alpha}_0}{\dashrightarrow}_{/\rightleftharpoons} \omega_0$ **via** Π_0' of the $\vec{\alpha}_0$-bridge to these two in order to get another triple $\langle h_{\mathbf{r}} \bullet h_\beta, \mathfrak{wit}(\pi), \varphi_0\rangle$, and transform this triple and the $\vec{\alpha}_0$-bridge's triples to quadruples as above. This gives us a reservation qbridge from $\mathbf{r}.\beta$ to ω **via** $\Pi'' = \Pi' \cup \{h_{\mathbf{r}}, h_\beta, \mathfrak{wit}(\pi)\}$. Add h_β as initial $\rightleftharpoons$-path, and we get the desired β-qbridge from $o = \mathbf{r}$ to ω **via** $\Pi''' = \Pi'' \cup \{h_\beta\} \equiv \Pi' \cup \{h_{\mathbf{r}}, h_\beta, h_{\mathbf{o}}' \bullet \pi_1 \bullet \pi_2'\} \equiv \Pi' \cup \{\pi_1\} \cup \Pi_\Lambda \cup \Pi_\circledast'$ with $\Pi_\Lambda = \{h_{\mathbf{r}}, h_{\mathbf{o}}'\}$ and $\Pi_\circledast' = \{h_\beta, \pi_2'\}$.

Since initially new π guarantees $\Pi' \cup \{h_{\mathbf{o}}\} \equiv \{\pi_2\} \cup \Pi_\Lambda \cup \Pi_\circledast$, we have $\Pi''' \cup \{h_{\mathbf{o}}\} \equiv \Pi' \cup \{\pi_1\} \cup \Pi_\Lambda \cup \Pi_\circledast' \cup \{h_{\mathbf{o}}\} \equiv \{h_{\mathbf{o}}, \pi_1, \pi_2\} \cup \Pi_\Lambda \cup (\Pi_\circledast \cup \Pi_\circledast') \equiv \{\pi\} \cup \Pi_\Lambda \cup (\Pi_\circledast \cup \Pi_\circledast')$.

In part (b), on one hand, π's $\vec{\alpha}_0$-bridge can be extended along dummy edge $h_\beta = \omega \xrightarrow{\beta} \omega.\beta$ to a $\vec{\alpha}_0.\beta$-bridge from q_0 to $\omega.\beta$ **via** $\Pi'' \equiv \Pi' \cup \{h_\beta\} \cup \Pi_\beta$ (Lemma 14). Lemma 21 allows two cases for initially new *ownership* path π with $\mu(\beta) \in \mathbb{A}_\perp$:

- Region-coupling $o.\mu(\beta) \rightleftharpoons \omega.\beta$ already existed in $\mathfrak{g}^\circledast$ **via** $\Pi''' = \Pi'' \cup \{\mathfrak{wit}(\pi)\} \cup \Pi_\Lambda''' \equiv \Pi' \cup \{h_\beta, \mathfrak{wit}(\pi)\} \cup \Pi_\beta \cup \Pi_\Lambda''' = \Pi' \cup \{h_{\mathbf{o}}', \tilde{\pi}'\} \cup \Pi_\Lambda''' \cup \Pi_\circledast'' = \Pi' \cup \{\tilde{\pi}'\} \cup \Pi_\Lambda'' \cup \Pi_\circledast''$ with $\Pi_\Lambda''' = \emptyset$ or $\{h_{\mathbf{r}}\}$, with $\Pi_\circledast'' = \Pi_\beta \cup \{h_\beta\}$ and with $\Pi_\Lambda'' = \{h_{\mathbf{o}}'\} \cup \Pi_\Lambda'''$.
- $\mu(\beta) \in \mathbb{A}$ and there was an ownership path-pair $\langle h_{\mathbf{r}} \bullet h_\beta', \mathfrak{wit}(\pi) \bullet h_\beta''\rangle = o.\mu(\beta) \overset{\hat{\mu}}{\dashleftarrow} \mathbf{s} \overset{\hat{\mu}}{\dashrightarrow} q_0.\vec{\alpha}_0.\beta$. These two are completed to a quadruple by trivial $\varphi' = o.\mu(\beta) \overset{\epsilon}{\dashleftarrow}_{/\rightleftharpoons} o.\mu(\beta)$ **via** $\emptyset$ to the left, and the $\vec{\alpha}_0.\beta$-bridge's initial $\rightleftharpoons$-path $\varphi_0' = q_0 \overset{\vec{\alpha}_0.\beta}{\dashrightarrow}_{/\rightleftharpoons} \omega_0'$ **via** Π_0'' to the right. It and the $\vec{\alpha}_0.\beta$-bridge's triples transformed to quadruples as above, constitute the desired reservation qbridge from $o.\mu(\beta)$ to $\omega.\beta$ **via** $\Pi''' = \Pi'' \cup \{h_{\mathbf{r}} \bullet h_\beta', \mathfrak{wit}(\pi) \bullet h_\beta''\} \equiv \Pi' \cup \{h_\beta, h_{\mathbf{r}}, h_\beta', h_{\mathbf{o}}', \tilde{\pi}', h_\beta''\} \cup \Pi_\beta = \Pi' \cup \{\tilde{\pi}'\} \cup \Pi_\Lambda'' \cup \Pi_\circledast''$ where $\Pi_\circledast'' = \Pi_\beta \cup \{h_\beta, h_\beta', h_\beta''\}$ and $\Pi_\Lambda'' = \{h_{\mathbf{r}}, h_{\mathbf{o}}'\}$.

Since initially new π guarantees $\Pi' \cup \{h_{\mathbf{o}}, \tilde{\pi}'\} \equiv \{\pi\} \cup \Pi_\Lambda \cup \Pi_\circledast$, we have $\Pi''' \cup \{h_{\mathbf{o}}\} \equiv \Pi' \cup \{\tilde{\pi}'\} \cup \Pi_\Lambda'' \cup \Pi_\circledast'' \cup \{h_{\mathbf{o}}\} \equiv \{\pi\} \cup \Pi_\Lambda \cup (\Pi_\circledast \cup \Pi_\circledast'')$.

In the induction step, we can ignore trivial $\rightleftharpoons$-paths $o \overset{\epsilon}{\dashrightarrow}_{/\rightleftharpoons} o$, since they produce nothing new.

For part (a), consider first the case of $o \overset{\vec{\gamma}}{\dashrightarrow}_{/\rightleftharpoons} \omega$ **via** Π because of $\vec{\gamma} = \vec{\gamma}_1 \bullet \vec{\gamma}_2$ and two $\rightleftharpoons$-paths $o \overset{\vec{\gamma}_1}{\dashrightarrow}_{/\rightleftharpoons} q \overset{\vec{\gamma}_2}{\dashrightarrow}_{/\rightleftharpoons} \omega$ **via** Π_1 and Π_2, respectively. The induction hypothesis guarantees a $\vec{\gamma}_1$-qbridge from o to q **via** Π'_1, and a $\vec{\gamma}_2$-qbridge from q to ω **via** Π'_2. These two qbridges combine to a $\vec{\gamma}_1.\vec{\gamma}_2$-qbridge from o to ω **via** $\Pi' \equiv \Pi'_1 \cup \Pi'_2 \cup \Pi_{\vec{\gamma}_2}$ (Lemma 22). This is the desired $\vec{\gamma}$-qbridge with $\Pi' \cup \Pi_{\mathbf{o}} \equiv \Pi'_1 \cup \Pi'_2 \cup \Pi_{\vec{\gamma}_2} \cup \Pi_{\mathbf{o}} \equiv (\Pi_1 \cup \Pi_\Lambda \cup \Pi'_{\circledast}) \cup (\Pi_2 \cup \Pi_\Lambda \cup \Pi''_{\circledast}) \cup \Pi_{\vec{\gamma}_2} = \Pi \cup (\Pi_\Lambda) \cup (\Pi'_{\circledast} \cup \Pi''_{\circledast} \cup \Pi_{\vec{\gamma}_2})$.

If $o \overset{\vec{\alpha}}{\dashrightarrow}_{\rightleftharpoons} \omega$ because $o.\vec{\alpha} \rightleftharpoons q.\vec{\gamma}$ **via** Π_1 and $q \overset{\vec{\gamma}}{\dashrightarrow}_{\rightleftharpoons} \omega$ **via** Π_2 then the induction hypothesis guarantees a $\vec{\gamma}$-qbridge from q to ω **via** Π'_2, and two cases for the region-coupling:

- Either $o.\vec{\alpha} \rightleftharpoons q.\vec{\gamma}$ is unchanged. Then it and the $\vec{\gamma}$-qbridge's initial $\rightleftharpoons$-path $q \overset{\vec{\gamma}}{\dashrightarrow}_{\rightleftharpoons} \omega_1$ **via** $\Pi'_{2,0}$ imply the $\rightleftharpoons$-path $o \overset{\vec{\alpha}}{\dashrightarrow}_{\rightleftharpoons} \omega_1$ **via** $\Pi_1 \cup \Pi'_{2,0}$ (Lemma 12). It, together with the $\vec{\gamma}$-qbridge's reservation qbridge from ω_1 to ω constitutes the desired $\vec{\alpha}$-qbridge from o to ω **via** $\Pi' = \Pi_1 \cup \Pi'_2$.
- Or there is a reservation qbridge from $o.\vec{\alpha}$ to $q.\vec{\gamma}$ **via** Π'_1. Its final $\rightleftharpoons$-path $o_n \overset{\vec{\alpha}_n}{\dashrightarrow}_{\rightleftharpoons} q.\vec{\gamma}$ **via** $\Pi'_{1,n}$ and the $\vec{\gamma}$-qbridge's initial $\rightleftharpoons$-path $q \overset{\vec{\gamma}}{\dashrightarrow}_{\rightleftharpoons} \omega_1$ **via** $\Pi'_{2,0}$ imply the $\rightleftharpoons$-path $o_n \overset{\vec{\alpha}_n}{\dashrightarrow}_{\rightleftharpoons} \omega_1$ **via** $\Pi'_{1,n} \cup \Pi'_{2,0}$ (Lemma 12). Substituting it for the reservation qbridge's final $\rightleftharpoons$-path links it with the $\vec{\gamma}$-qbridge's reservation qbridge to a reservation qbridge from $o.\vec{\alpha}$ to ω **via** $\Pi'_1 \cup \Pi'_2$. Prefixing it with $\pi_\alpha = o \xrightarrow{\vec{\alpha}} o.\vec{\alpha}$ as a basic $\rightleftharpoons$-path produces the desired $\vec{\alpha}$-qbridge from o to ω **via** $\Pi' = \Pi'_1 \cup \Pi'_2 \cup \{\pi_\alpha\}$.

Since $\Pi'_2 \cup \Pi''_{\mathbf{o}} \equiv \Pi_2 \cup \Pi_\Lambda \cup \Pi''_{\circledast}$, in the former case $\Pi' \cup \Pi''_{\mathbf{o}} \equiv \Pi_1 \cup (\Pi'_2 \cup \Pi''_{\mathbf{o}}) \equiv \Pi_1 \cup (\Pi_2 \cup \Pi_\Lambda \cup \Pi''_{\circledast}) = \Pi \cup \Pi_\Lambda \cup \Pi''_{\circledast}$. In the latter case also $\Pi'_1 \cup \Pi'_{\mathbf{o}} \equiv \Pi_1 \cup \Pi_\Lambda \cup \Pi'_{\circledast}$, so that $\Pi' \cup (\Pi'_{\mathbf{o}} \cup \Pi''_{\mathbf{o}}) \equiv \Pi'_1 \cup \Pi'_2 \cup \{\pi_\alpha\} \cup (\Pi'_{\mathbf{o}} \cup \Pi''_{\mathbf{o}}) \equiv (\Pi_1 \cup \Pi_\Lambda \cup \Pi'_{\circledast}) \cup (\Pi_2 \cup \Pi_\Lambda \cup \Pi''_{\circledast}) \cup \{\pi_\alpha\} = \Pi \cup \Pi_\Lambda \cup (\Pi'_{\circledast} \cup \Pi''_{\circledast} \cup \{\pi_\alpha\})$.

In one case of the induction step of part (b), $o.\vec{\alpha} \rightleftharpoons \omega.\vec{\gamma}$ **via** Π is new because the inverse $\omega.\vec{\gamma} \rightleftharpoons o.\vec{\alpha}$ **via** Π is new. Then the induction hypothesis guarantees a reservation qbridge from $\omega.\vec{\gamma}$ to $o.\vec{\alpha}$ **via** Π'. Note that reservation qbridges are symmetric in structure: If read in reverse, this reservation qbridge is a reservation qbridge from $o.\vec{\alpha}$ to $\omega.\vec{\gamma}$.

In another case, $o.\vec{\alpha} \rightleftharpoons \omega.\vec{\gamma}$ because $o.\vec{\alpha} \rightleftharpoons q.\vec{\beta}$ **via** Π_1 and $q.\vec{\beta} \rightleftharpoons \omega.\vec{\gamma}$ **via** Π_2. If none of the two region-couplings is new, they combined to $o.\vec{\alpha} \rightleftharpoons \omega.\vec{\gamma}$ **via** $\Pi_1 \cup \Pi_2 = \Pi$ already in $\mathfrak{g}^{\circledast}$. If both are new, then the induction hypothesis's reservation qbridges from $o.\vec{\alpha}$ to $q.\vec{\beta}$ and from $q.\vec{\beta}$ to $\omega.\vec{\gamma}$ concatenate to one reservation qbridge from $o.\vec{\alpha}$ to $\omega.\vec{\gamma}$ **via** $\Pi' = \Pi'_1 \cup \Pi'_2$. If the first region-coupling is new and the second old, then the induction hypothesis's reservation qbridges from $o.\vec{\alpha}$ to $q.\vec{\beta}$ **via** Π'_1 in $\mathfrak{g}^{\circledast}$ has a last $\rightleftharpoons$-path $o_n \overset{\vec{\alpha}_n}{\dashrightarrow}_{\rightleftharpoons} q.\vec{\beta}$ **via** $\Pi'_{1,n}$. In conjunction with old $q.\vec{\beta} \rightleftharpoons \omega.\vec{\gamma}$ in $\mathfrak{g}^{\circledast}$, this entails $o_n \overset{\vec{\alpha}_n}{\dashrightarrow}_{\rightleftharpoons} \omega.\vec{\gamma}$ **via** $\Pi'_{1,n} \cup \Pi_2$ (Lemma 22). It, together with the rest of the reservation qbridge from $o.\vec{\alpha}$ to $q.\vec{\beta}$ is a reservation qbridge from $o.\vec{\alpha}$ to $\omega.\vec{\gamma}$ **via** $\Pi' = \Pi'_1 \cup \Pi_2$. The case of unchanged first and new second region-coupling follows by symmetry.

If new $o.\vec{\alpha}.\vec{\gamma} \rightleftharpoons \omega.\vec{\gamma}$ is implied by new $o \overset{\vec{\alpha}}{\dashrightarrow}_{\rightleftharpoons} \omega$ **via** Π, then the induction hypothesis guarantees an $\vec{\alpha}$-qbridge from o to ω **via** Π' in $\mathfrak{g}^{\circledast}$. It can be extended by $\omega \xrightarrow{\vec{\gamma}} \omega.\vec{\gamma}$ (as a $\rightleftharpoons$-path **via** $\Pi_\omega = \{\omega \xrightarrow{\gamma_1} \omega.\gamma_1, \ \ldots, \ \omega.\gamma_1 \ldots \gamma_{n-1} \xrightarrow{\gamma_n} \omega.\vec{\gamma}\}$) to a $\vec{\alpha}.\vec{\gamma}$-qbridge from o to $\omega.\vec{\gamma}$ **via** $\Pi'' \equiv \Pi' \cup \Pi_\omega \cup \Pi_{\vec{\gamma}}$ (Lemma 22). That qbridge's initial $\rightleftharpoons$-path $o \dashrightarrow^{\vec{\alpha}.\vec{\gamma}}_{/\rightleftharpoons} \omega'_1$ **via** Π''_0 means $o.\vec{\alpha}.\vec{\gamma}.\epsilon \rightleftharpoons \omega'_1.\epsilon$. Hence dummy edge path $o \xrightarrow{\vec{\alpha}.\vec{\gamma}} o.\vec{\alpha}.\vec{\gamma}$ (as a $\rightleftharpoons$-path **via** $\Pi_{\vec{\alpha}.\vec{\gamma}} = \{o \xrightarrow{\alpha_1} o.\alpha_1, \ \ldots, \ o.\alpha_1 \ldots \alpha_{n-1} \xrightarrow{\alpha_n} o.\vec{\alpha}\}$) implies the $\rightleftharpoons$-path $\omega'_1 \dashrightarrow^{\epsilon}_{/\rightleftharpoons} o.\vec{\alpha}.\vec{\gamma}$ **via** $\Pi''_0 \cup \Pi_{\vec{\alpha}.\vec{\gamma}}$. It can extend the $\rightleftharpoons$-path $\varphi'_i = o'_1 \dashrightarrow^{\vec{\alpha}'_1}_{/\rightleftharpoons} \omega'_1$ **via** Π''_1 of the $\vec{\alpha}.\vec{\gamma}$-qbridge's first quadruple $\langle \varphi'_1, \pi'_1, \pi_1, \varphi_1 \rangle$ to $\tilde{\varphi}_i = o'_1 \dashrightarrow^{\vec{\alpha}'_1.\epsilon}_{/\rightleftharpoons} o.\vec{\alpha}.\vec{\gamma}$ **via** $\Pi''_0 \cup \Pi_{\vec{\alpha}.\vec{\gamma}} \cup \Pi''_1$. By substituting $\tilde{\varphi}_i$ for φ'_i, the $\vec{\alpha}.\vec{\gamma}$-qbridge's reservation qbridge from ω'_1 to $\omega.\vec{\gamma}$ is transformed to the desired reservation qbridge from $o.\vec{\alpha}.\vec{\gamma}$ to $\omega.\vec{\gamma}$ **via** $\Pi''' = \Pi'' \cup \Pi_{\vec{\alpha}.\vec{\gamma}}$.

By induction hypothesis, $\Pi' \cup \Pi_{\mathbf{o}} \equiv \Pi \cup \Pi_\Lambda \cup \Pi_\circledast$. Hence $\Pi''' \cup \Pi_{\mathbf{o}} \equiv (\Pi' \cup \Pi_\omega \cup \Pi_{\vec{\gamma}}) \cup \Pi_{\vec{\alpha}.\vec{\gamma}} \cup \Pi_{\mathbf{o}} \equiv \Pi \cup \Pi_\Lambda \cup (\Pi_\circledast \cup \Pi_\omega \cup \Pi_{\vec{\gamma}} \cup \Pi_{\vec{\alpha}.\vec{\gamma}})$. ■

6.3.4 Technical Lemmas for the Region-Coupling Level

The following technical lemmas are all in the context of parameter supply substeps during a *legal* {call}-reduction step, i.e., with a typing $\Gamma_n, \kappa_n \vdash_\epsilon \hat{e} : \hat{\tau}$ for the operation call expression redex $\hat{e}$. They concern the graph $\mathfrak{g}'$ after addition of a new received handle $h_{\mathbf{o}} = \mathbf{r} \xrightarrow{\mu_{\mathbf{o}}} \mathbf{o}$ and removal of old sent handle $h'_{\mathbf{o}} = \mathbf{s} \xrightarrow{\mu_{\mathbf{r}} \circ \mu_{\mathbf{o}}} \mathbf{o}$ from the previous graph $\mathfrak{g}$, i.e., $\mathfrak{g}' = \mathfrak{g} \oplus \mathbf{r} \xrightarrow{\mu_{\mathbf{o}}} \mathbf{o} \ominus \mathbf{s} \xrightarrow{\mu_{\mathbf{r}} \circ \mu_{\mathbf{o}}} \mathbf{o}$.

The first three lemmas are about initially new ownership paths. The final lemma is about qbridges.

Lemma 19 Assume $\mathfrak{g} \models \mathsf{UH}$ and $\mathfrak{g} = ogr(e, \vec{\eta}, \mathfrak{s})$. Then a sent handle that is `free` cannot coincide with the call-link.

Proof: On one hand, if sent handle and call-link were one and the same edge h, then the operation call expression $\hat{e}$ would contain the handle h twice: As value $\langle \mathbf{s}, \mu_{\mathbf{r}}, \mathbf{r} \rangle$ of the receiver expression and as value $\langle \mathbf{s}, \mu_{\mathbf{r}} \circ \mu_{\mathbf{o}}, \mathbf{o} \rangle$ of an argument expression. Hence its multiplicity in $\mathfrak{g}$ should be more than one by $\mathfrak{g} = ogr(e, \vec{\eta}, \mathfrak{s})$. On the other hand, if the sent handle is `free`, $\mathfrak{g} \models \mathsf{UH}$ ensures that its multiplicity is one in $\mathfrak{g}$. A contradiction. ■

Lemma 20 Assume the reserved ownership assumption in $\mathfrak{g}^\circledast$ and $\mathfrak{g} = ogr(e, \vec{\eta}, \mathfrak{s})$. Consider an initially new ownership path $\pi \in PAP_{\mathfrak{g}'^\circledast}(o, \mu, \omega)$. Its witness $\mathfrak{wit}(\pi) = \mathbf{s} \dashrightarrow^{\mu_{\mathbf{r}} \circ \mu} q_0.\vec{\alpha}_0$ is related with ω through a single $\rightleftharpoons$-path $q_0 \dashrightarrow^{\vec{\alpha}_0}_{/\rightleftharpoons} \omega_0 = \omega$; this path is the $\vec{\alpha}_0$-bridge from q_0 to ω; there is no series of triples.

Proof: Initially new π is an extension of $h_{\mathbf{o}}$. Hence by $\Gamma_n, \kappa_n \vdash_\epsilon \hat{e} : \hat{\tau}$, it can be an ownership path only if it is `free`, i.e., $\mu =$ `free<...>`. But then $\mathfrak{wit}(\pi)$ is `free` too: $\mu_{\mathbf{r}} \circ \mu = \mu_{\mathbf{r}} \circ$`free<...>` = `free<...>`. Since it is $h'_{\mathbf{o}}$'s extension and the nesting constraint excludes correlations to `free` modes, $h'_{\mathbf{o}}$ must be `free`, so that it cannot coincide with the call-link $h_{\mathbf{r}} = \mathbf{s} \xrightarrow{\mu_{\mathbf{r}}} \mathbf{r}$: This follows from Lemma 19 since the reserved ownership assumption obviously implies $\mathfrak{g} \models \mathsf{UH}$.

If there is a triple in π's the $\vec{\alpha}_0$-bridge there would be a first ownership path-pair $\omega_0 \xleftarrow{\hat{\mu}_1} \mathbf{s} \xrightarrow{\hat{\mu}_1} q_1.\vec{\alpha}_1$. The left one starts with $h_\mathbf{r}$ by the definition of initially new paths (since $\mu_\mathbf{o} \neq \mathsf{co<>}$). The triple of witness $\mathfrak{wit}(\pi) = \mathbf{s} \xrightarrow{\mu_\mathbf{r} \circ \mu} q_0.\vec{\alpha}_0$, initial $\rightleftharpoons$-path $q_0 \dashrightarrow^{\vec{\alpha}_0}_{\not\rightleftharpoons} \omega_0$, and $\omega_0 \xleftarrow{\hat{\mu}_1} \mathbf{s}$ would by the reserved ownership assumption imply that $\mathfrak{wit}(\pi)$ also starts with $h_\mathbf{r}$. $\mathfrak{wit}(\pi)$, however, by definition starts with $h'_\mathbf{o}$. But `free` $h'_\mathbf{o}$ cannot coincide with $h_\mathbf{r}$. ■

Lemma 21 Assume the reserved ownership assumption in $\mathfrak{g}^{\circledast}$ and $\mathfrak{g} = ogr(e, \vec{\eta}, \mathfrak{s})$. Consider an initially new ownership path $\pi \in PAP_{\mathfrak{g}'^{\circledast}}(\mathbf{r}, \mu, \omega)$ whose $\vec{\alpha}_0$-bridge from q_0 to ω **via** Π in $\mathfrak{g}^{\circledast}$ starts with $\rightleftharpoons$-path $q_0 \dashrightarrow^{\vec{\alpha}_0}_{\not\rightleftharpoons} \omega_0$ **via** Π_0. For any β with $\mu(\beta) \in \mathbb{A}_\perp$, there was in $\mathfrak{g}^{\circledast}$,

- a region-coupling $\mathbf{r}.\mu(\beta) \rightleftharpoons \omega.\beta$ **via** $\Pi \cup \{\mathfrak{wit}(\pi)\}$ or $\Pi \cup \{\mathfrak{wit}(\pi), h_\mathbf{r}\}$, or
- an ownership path-pair $\langle h_\mathbf{r} \bullet h'_\beta, \mathfrak{wit}(\pi) \bullet h_\beta \rangle = \mathbf{r}.\mu(\beta) \xleftarrow{\hat{\mu}} \mathbf{s} \xrightarrow{\hat{\mu}} (q_0.\vec{\alpha}_0).\beta$ extending $h_\mathbf{r}$ and $\mathfrak{wit}(\pi)$, respectively, by dummy edges h'_β and h_β while $\mu(\beta) \in \mathbb{A}$.

Proof: Let $\omega' = q_0.\vec{\alpha}_0$ be short for the target of π's witness $\mathfrak{wit}(\pi) \in PAP_{\mathfrak{g}^{\circledast}}(\mathbf{s}, \mu, q_0.\vec{\alpha}_0)$. Notice that there are no triples in initially new ownership path π's $\vec{\alpha}_0$-bridge, i.e., $\omega_0 = \omega$ and $\Pi_0 = \Pi$ (Lemma 20). Initially new π is an extension of $h_\mathbf{o}$. Hence by $\Gamma_n, \kappa_n \vdash_\epsilon \hat{e} : \hat{\tau}$, it can be an ownership path only if it is `free`. But then witness $\mathfrak{wit}(\pi)$ of mode $\mu_\mathbf{r} \circ \mu$ is `free` too and correlates the same association roles which π correlates.

If $\mu(\beta) = \perp$, this means that μ lacks a correlation for β. Then also $\mu_\mathbf{r} \circ \mu$ lacks it, so that `free` $\mathfrak{wit}(\pi) \in PAP_{\mathfrak{g}^{\circledast}}(\mathbf{s}, \mu, \omega')$ established $\mathbf{s}.\perp \rightleftharpoons \omega'.\beta$ **via** $\{\mathfrak{wit}(\pi)\}$. But since $\mathbf{r}.\perp \rightleftharpoons \mathbf{s}.\perp$, this entails $\mathbf{r}.\perp = \mathbf{r}.\mu(\beta) \rightleftharpoons \omega'.\beta = q_0.\vec{\alpha}_0.\beta$ **via** $\{\mathfrak{wit}(\pi)\}$. And since $q_0 \dashrightarrow^{\vec{\alpha}_0}_{\not\rightleftharpoons} \omega_0 = \omega$ means $q_0.\vec{\alpha}_0.\beta \rightleftharpoons \omega.\beta$ **via** $\Pi_0 = \Pi$ we have $\mathbf{r}.\mu(\beta) \rightleftharpoons \omega.\beta$ **via** $\Pi \cup \{\mathfrak{wit}(\pi)\}$.

If $\mu(\beta) = \alpha \in \mathbb{A}$, this means that μ contains the correlation β=α<>. But then $\mu_\mathbf{o}$, which contains μ, contains α<>. Therefore the existence of $h'_\mathbf{o} = \mathbf{s} \xrightarrow{\mu_\mathbf{r} \circ \mu_\mathbf{o}} \mathbf{o}$ presupposes a corresponding correlation α=$\hat{\mu}$ to some $\hat{\mu}$ in the call-link's mode $\mu_\mathbf{r}$. It implies that $\mu_\mathbf{r} \circ \mu$ contains the correlation β=$\hat{\mu}$.

- $\hat{\mu}$ cannot be `co<>` since there are no correlations to `co`; neither can it be a `read` mode Since $\mu(\beta) = \mu_\mathbf{o}(\vec{\alpha}_0.\beta) = \alpha$, $\Gamma_n, \kappa_n \vdash_\epsilon \hat{e} : \hat{\tau}$ ensures that $\mu_\mathbf{r} \circ \mu_\mathbf{o}(\vec{\alpha}_0.\beta) = \mu_\mathbf{r} \circ \mu(\beta) = \hat{\mu}(\epsilon)$ is not `read`.
- If $\hat{\mu}$ is a `free` or `rep` mode then, on one side, call-link $h_\mathbf{r} = \mathbf{s} \xrightarrow{m\texttt{<}\ldots,\alpha=\hat{\mu},\ldots\texttt{>}} \mathbf{r}$ is extended by dummy edge $h'_\beta = \mathbf{r} \xrightarrow{\alpha\texttt{<>}} \mathbf{r}.\alpha$ to the ownership path $\hat{\pi}' = \mathbf{s} \xrightarrow{\hat{\mu}} \mathbf{r}.\alpha = \mathbf{r}.\mu(\beta)$. On the other side, $\mathfrak{wit}(\pi)$ of mode $\mu_\mathbf{r} \circ \mu = \texttt{free<}\ldots,\beta=\hat{\mu},\ldots\texttt{>}$ is extended by dummy edge $h_\beta = q_0.\vec{\alpha}_0 \xrightarrow{\beta\texttt{<>}} q_0.\vec{\alpha}_0.\beta$ to the ownership path $\hat{\pi} = \mathbf{s} \xrightarrow{\hat{\mu}} q_0.\vec{\alpha}_0.\beta$
- If $\hat{\mu}$ is an association mode γ<> then, first, call-link $h_\mathbf{r} = \mathbf{s} \xrightarrow{m\texttt{<}\ldots,\alpha=\gamma\texttt{<>},\ldots\texttt{>}} \mathbf{r}$ established $\mathbf{s}.\gamma \rightleftharpoons \mathbf{r}.\alpha$ **via** $\{h_\mathbf{r}\}$ since it is `free` or `rep`: As a mode with a correlation α=$\hat{\mu}$, $\mu_\mathbf{r}$ cannot be a `co`- or association mode; and a `read` mode is excluded by $\Gamma_n, \kappa_n \vdash_\epsilon \hat{e} : \hat{\tau}$ since $\mu(\beta) = \mu_\mathbf{o}(\vec{\alpha}_0.\beta) \in \mathbb{A}$. Second, $\mathfrak{wit}(\pi)$ of mode $\mu_\mathbf{r} \circ \mu = \texttt{free<}\ldots,\beta=\gamma\texttt{<>},\ldots\texttt{>}$ established $\mathbf{s}.\gamma \rightleftharpoons (q_0.\vec{\alpha}_0).\beta$ **via** $\{\mathfrak{wit}(\pi)\}$. Third, first and only $\rightleftharpoons$-path $q_0 \dashrightarrow^{\vec{\alpha}_0}_{\not\rightleftharpoons} \omega_0 = \omega$ established $q_0.\vec{\alpha}_0.\beta \rightleftharpoons \omega.\beta$ **via** $\Pi_0 = \Pi$. All three region-couplings together mean $\mathbf{r}.\mu(\beta) \rightleftharpoons \omega.\beta$ **via** $\Pi \cup \{h_\mathbf{r}, \mathfrak{wit}(\pi)\}$. ■

Lemma 22 (Qbridges)

a) If there is a $\vec{\alpha}$-qbridge from o to ω **via** Π and $o.\vec{\alpha} \rightleftharpoons o'.\vec{\alpha}'$ **via** Π' then there is a $\vec{\alpha}'$-qbridge from o' to ω **via** $\Pi \cup \Pi'$.

b) A $\vec{\alpha}$-qbridge from o to ω **via** Π can be extended along $\omega \overset{\beta}{\dashrightarrow}_{/\rightleftharpoons} \omega'$ **via** Π' to a $\vec{\alpha}.\beta$-qbridge from o to ω' **via** $\Pi' \equiv \Pi \cup \Pi' \cup \Pi_\beta$.

c) A $\vec{\alpha}$-qbridge from o to q **via** Π_1 and a $\vec{\gamma}$-qbridge from q to ω **via** Π_2 concatenate to a $\vec{\alpha}.\vec{\gamma}$-qbridge from o to ω **via** $\Pi' \equiv \Pi_1 \cup \Pi_2 \cup \Pi_{\vec{\gamma}}$.

Proof: Part (a). The $\vec{\alpha}$-qbridge starts with a $\rightleftharpoons$-path $o \overset{\vec{\alpha}}{\dashrightarrow}_{/\rightleftharpoons} \omega_1$ **via** Π_0. It, and $o.\vec{\alpha} \rightleftharpoons o'.\vec{\alpha}'$ imply the $\rightleftharpoons$-path $o' \overset{\vec{\alpha}'}{\dashrightarrow}_{/\rightleftharpoons} \omega_1$ **via** $\Pi_0 \cup \Pi'$. It, together with the $\vec{\alpha}$-qbridge's reservation qbridge from ω_1 to ω, constitutes a $\vec{\alpha}'$-qbridge from o' to ω **via** $\Pi \cup \Pi'$.

Part (b). The proof is a generalization of that for the extension of bridges with triple-series (Lemma 14): The $\rightleftharpoons$-paths $o = q_0 \overset{\vec{\alpha}_0}{\dashrightarrow}_{/\rightleftharpoons} \omega_1$ **via** Π_0, and each intermediate $\omega_i \overset{\vec{\alpha}'_i}{\dashleftarrow}_{/\rightleftharpoons} q'_i$ **via** Π'_i and $q_i \overset{\vec{\alpha}_i}{\dashrightarrow}_{/\rightleftharpoons} \omega_{i+1}$ **via** Π_i in the quadruples, can obviously be extended to $q_0 \overset{\vec{\alpha}_0.\beta}{\dashrightarrow}_{/\rightleftharpoons} \omega_1.\beta$ **via** $\Pi_0 \cup \{\omega_1 \xrightarrow{\beta} \omega_1.\beta\}$, to $\omega_i.\beta \overset{\vec{\alpha}'_i.\beta}{\dashleftarrow}_{/\rightleftharpoons} q'_i$ **via** $\Pi'_i \cup \{\omega_i \xrightarrow{\beta} \omega_i.\beta\}$, and to $q_i \overset{\vec{\alpha}_i.\beta}{\dashrightarrow}_{/\rightleftharpoons} \omega_{i+1}$ **via** $\Pi_i \cup \{\omega_{i+1} \xrightarrow{\beta} \omega_{i+1}.\beta\}$, respectively. The final $\rightleftharpoons$-path $q_n \overset{\vec{\alpha}_n}{\dashrightarrow}_{/\rightleftharpoons} \omega$ **via** Π_i extends to $q_n \overset{\vec{\alpha}_n.\beta}{\dashrightarrow}_{/\rightleftharpoons} \omega'$ **via** $\Pi_i \cup \Pi_\beta$. The corresponding ownership path-pair $q'_i.\vec{\alpha}'_i.\beta \overset{\hat{\mu}'_i}{\dashleftarrow} \mathbf{s} \overset{\hat{\mu}'_i}{\dashrightarrow} q_i.\vec{\alpha}_i.\beta$ can be obtained by extension of those quadruple's ownership path-pairs $q'_i.\vec{\alpha}'_i \overset{\hat{\mu}_i}{\dashleftarrow} \mathbf{s} \overset{\hat{\mu}_i}{\dashrightarrow} q_i.\vec{\alpha}_i$ *if* $\hat{\mu}_i$ has an β-correlation to a `free` or `rep` mode.

As explained in the proof of Lemma 14, ownership path pairs $\langle \pi'_i, \pi_i \rangle$ that cannot be extended instead establish $(q'_i.\vec{\alpha}'_i).\beta \rightleftharpoons \mathbf{s}.\hat{\mu}_i(\beta) \rightleftharpoons (q_i.\vec{\alpha}_i).\beta$ **via** $\{\pi'_i, \pi_i\}$. This is extended to the left by $\omega_i \overset{\vec{\alpha}'_i}{\dashleftarrow}_{/\rightleftharpoons} q'_{i+1}$ **via** Π'_i implying $\omega_i.\beta \rightleftharpoons q'_{i+1}.\vec{\alpha}'_i.\beta$, and to the right by $q_i \overset{\vec{\alpha}_i}{\dashrightarrow}_{/\rightleftharpoons} \omega_{i+1}$ **via** Π_i implying $q_i.\beta \rightleftharpoons \omega_{i+1}.\vec{\alpha}_i.\beta$, to the region-coupling $\omega_i.\beta \rightleftharpoons \omega_{i+1}.\beta$ **via** $\Pi'_i \cup \Pi_i \cup \{\pi'_i, \pi_i\}$. Hence, for any number of consecutive modes $\hat{\mu}_i, \ldots, \hat{\mu}_j$ with $\hat{\mu}_k(\beta) \in \mathbb{A}_\perp$, the gap between q_0 (if $i = 0$) or the preceding extended ownership path-pair (if $i > 0$), and ω (if $j = n$) or the following extended ownership path-pair (if $j < n$) in the $\vec{\beta}.\beta$-qbridge from o to ω is closed, as shown in Lemma 14, by a $\rightleftharpoons$-path $q_{i-1} \overset{\vec{\alpha}_{i-1}.\beta}{\dashrightarrow}_{/\rightleftharpoons} \omega_{j+1}.\beta$ or $q_{i-1} \overset{\vec{\alpha}_{i-1}.\beta}{\dashrightarrow}_{/\rightleftharpoons} \omega$, respectively.

Part (c). By part (b), the first qbridge can be incrementally extended along the second qbridge's initial $\rightleftharpoons$-path $q \overset{\vec{\alpha}_2}{\dashrightarrow}_{/\rightleftharpoons} \omega_1$ **via** $\Pi_{2,0}$ to a $\vec{\alpha}_1.\vec{\alpha}_2$-qbridge from o to ω_1 **via** $\Pi'_1 \equiv \Pi_1 \cup \Pi_{2,0} \cup \Pi'_\beta$. It is extended by the second qbridge's reservation qbridge from ω_1 to ω to a $\vec{\alpha}_1.\vec{\alpha}_2$-qbridge from o to ω **via** $\Pi' \equiv \Pi_1 \cup \Pi_2 \cup \Pi_{\gamma_1} \cup \ldots \cup \Pi_{\gamma_n}$. ■

6.3.5 The Structure of Reserved Ownership

The reasoning about the reserved ownership assumption is a generalization of the reasoning about the structure of object ownership in §6.3.1.

Lemma 23 If $e_0,\ \eta_0, \mathfrak{s}_0, om_0, \mathfrak{g}_0 \Longrightarrow^* e',\ \vec{\eta}', \mathfrak{s}', om', \mathfrak{g}'$ is a reduction defined relative to a program p with $\vdash p$ **start** e_0 then $\mathfrak{g}'^{\circledast}$ satisfies the reserved ownership assumption (Definition 13).

Proof by induction on the number N of reduction steps from e_0 to e': In the base case $N = 0$, $\mathfrak{g}'$ is the empty graph $\mathfrak{g}_0 = \emptyset$. Its extension $\mathfrak{g}'^{\circledast}$ by dummy edges of non-composition modes trivially satisfies the assumption. In the induction step $N \rightarrow N+1$, reduction $e_0,\ \eta_0, \mathfrak{s}_0, om_0, \mathfrak{g}_0 \Longrightarrow^* e,\ \vec{\eta}, \mathfrak{s}, om, \mathfrak{g}$ is continued $e,\ \vec{\eta}, \mathfrak{s}, om, \mathfrak{g} \Longrightarrow e',\ \vec{\eta}', \mathfrak{s}', om', \mathfrak{g}'$. By induction hypothesis, the assumption holds in $\mathfrak{g}^{\circledast}$. A look at the context rules shows that the changes to the object graph are absolutely independent of the term context around the redex $\hat{e}$. Hence we can move directly to a case analysis of the rule by which redex $\hat{e}$ is reduced.

In case of $\{\mathrm{var}_l\}$, $\{\mathrm{var}_f\}$, $\{\mathrm{rd}_{dst}\}$, and $\{\mathrm{null}\}$, the object graph is unchanged, so that the assumption is trivially preserved. As explained in the proof for Theorem 2, the case of $\{\mathrm{rd}_{cp}\}$ is harmless since the type-system prevents the replicated handle from having a `free` mode. In the remaining cases, we will have to exclude after each substep to a graph $\mathfrak{g}''$, a violation of the reserved ownership assumption by any combination $\langle \pi, \varphi, \varphi', \pi' \rangle$ made of two ownership paths $\pi \in PAP_{\mathfrak{g}''^{\circledast}}(o, \mu, q.\vec{\alpha})$ and $\pi' \in PAP_{\mathfrak{g}''^{\circledast}}(o', \mu', q'.\vec{\alpha}')$, and two $\rightleftharpoons$-paths $\varphi = q \overset{\vec{\alpha}}{\dashrightarrow}\!/_{\rightleftharpoons}\ \omega$ and $\varphi' = q' \overset{\vec{\alpha}'}{\dashrightarrow}\!/_{\rightleftharpoons}\ \omega$.

$\boxed{\{\mathrm{new}\}}$ The reduction of `new<`δ`>` c`()` adds an edge $h_{\mathbf{o}} = \mathbf{r} \xrightarrow{\mu_{\mathbf{o}}} \mathbf{o}$ to a *fresh* object $\mathbf{o}$, where $\mu_{\mathbf{o}} =$ `free<`δ`>`. The kinds of ownership paths π in $\mathfrak{g}'^{\circledast}$ guaranteed by Lemma 6 and the kinds of $\rightleftharpoons$-paths φ in $\mathfrak{g}'^{\circledast}$ guaranteed by Lemma 15 combine as follows:

- If π is an initially new ownership path then its target $q.\vec{\alpha}$ is some $(\mathbf{o}.\vec{\gamma}).\vec{\alpha}$ with $\mu_{\mathbf{o}}(\vec{\gamma}.\vec{\alpha}) \in \{$`free`, `rep`$\}$. Since no modes can be nested to an association mode, this implies $\mu_{\mathbf{o}}(\vec{\beta}) \notin \mathbb{A}$ for all prefixes $\vec{\beta}$ of $\vec{\gamma}.\vec{\alpha}$. Hence the only $\rightleftharpoons$-path $q \overset{\vec{\alpha}}{\dashrightarrow}\!/_{\rightleftharpoons}\ \omega$ is the dummy edge path $\mathbf{o}.\vec{\gamma} \xrightarrow{\vec{\alpha}} \omega = \mathbf{o}.\vec{\gamma}.\vec{\alpha} = q.\vec{\alpha}$ (Lemma 15).
- If ownership path π is unchanged then $q.\vec{\alpha}$ cannot be $\mathbf{o}$ or one of its region objects. Hence $\sigma(q) = q$ and $q \overset{\vec{\alpha}}{\dashrightarrow}\!/_{\rightleftharpoons}\ \omega$ must be a $\rightleftharpoons$-path with a counterpart $\tilde{\varphi} = \sigma(q) \overset{\vec{\alpha}}{\dashrightarrow}\!/_{\rightleftharpoons}\ \sigma(\omega)$ in $\mathfrak{g}^{\circledast}$ (Lemma 15). π and $\tilde{\varphi}$ are half a quadruple $\langle \pi, \tilde{\varphi}, \ldots \rangle$ since $q = \sigma(q)$.
- If π is internally new then its target $q.\vec{\alpha}$ is some $\mathbf{o}.\vec{\gamma}.\vec{\gamma}'$ with $\mu_{\mathbf{o}}(\vec{\gamma}) \in \mathbb{A}$. That is, $q = \mathbf{o}.\vec{\alpha}'$ for the $\vec{\alpha}'$ with $\vec{\alpha}'.\vec{\alpha} = \vec{\gamma}.\vec{\gamma}'$. If $\vec{\alpha}'$ is a proper prefix of $\vec{\gamma}$ with $\vec{\gamma} = \vec{\alpha}'.\vec{\alpha}''$ then $q \overset{\vec{\alpha}}{\dashrightarrow}\!/_{\rightleftharpoons}\ \omega$ can be decomposed into a dummy edge path $\mathbf{o}.\vec{\alpha}' \xrightarrow{\vec{\alpha}''} \tilde{q} = \mathbf{o}.\vec{\alpha}'.\vec{\alpha}''$ and a $\rightleftharpoons$-path $\tilde{\varphi} = \tilde{q} \overset{\vec{\gamma}'}{\dashrightarrow}\!/_{\rightleftharpoons}\ \omega$ with counterpart $\sigma(\tilde{q}) \overset{\vec{\gamma}'}{\dashrightarrow}\!/_{\rightleftharpoons}\ \sigma(\omega)$ (Lemma 15). Hence this case of $\pi \in PAP_{\mathfrak{g}'^{\circledast}}(o, \mu, q.\vec{\alpha})$ and $\varphi = q \overset{\vec{\alpha}}{\dashrightarrow}\!/_{\rightleftharpoons}\ \omega$ can be reduced to the case of $\tilde{\pi} \in PAP_{\mathfrak{g}'^{\circledast}}(o, \mu, \tilde{q}.\vec{\gamma})$ and $\tilde{\varphi} = \tilde{q} \overset{\vec{\gamma}'}{\dashrightarrow}\!/_{\rightleftharpoons}\ \omega$. In the case that $\vec{\alpha}'$ is $\vec{\gamma}$ or an extension of it, $q \overset{\vec{\alpha}}{\dashrightarrow}\!/_{\rightleftharpoons}\ \omega$ must be a $\rightleftharpoons$-path with a counterpart $\tilde{\varphi} = \sigma(q) \overset{\vec{\alpha}}{\dashrightarrow}\!/_{\rightleftharpoons}\ \sigma(\omega)$ in $\mathfrak{g}^{\circledast}$ (Lemma 15). π's witness $\sigma(\pi)$ and $\tilde{\varphi}$ are half a quadruple $\langle \sigma(\pi), \tilde{\varphi}, \ldots \rangle$ since $\sigma(\pi)$'s target is $\mathbf{r}.\mu_{\mathbf{o}}(\vec{\gamma}).\vec{\gamma}' = \sigma(q)$.

This shows that if ω is some $\mathbf{o}.\vec{\gamma}$ with $\mu_{\mathbf{o}}(\vec{\gamma}) \notin \mathbb{A}$ for all prefixes $\vec{\gamma}'$ of $\vec{\gamma}$ then, first, π and π' must both be initially new, and, second, $q.\vec{\alpha} = \omega = q'.\vec{\alpha}'$. Hence, π and

π' are one and the same potential access path $\pi = \pi_{\tilde{\gamma}} = \pi'$, and thus automatically have the same source, mode, and shape: The quadruple $\langle \pi, \varphi, \varphi', \pi' \rangle$ in $\mathfrak{g}'^{\circledast}$ satisfies the assumption.

In other cases of ω, π and π' are unchanged or internally new, and thus have counterparts $\tilde{\pi} = \pi$ or $\tilde{\pi} = \sigma(\pi)$, and $\tilde{\pi}' = \pi'$ or $\tilde{\pi}' = \sigma(\pi')$, respectively, which combine with the $\rightleftharpoons$-paths' counterparts $\tilde{\varphi}$ and $\tilde{\varphi}'$ to a quadruple $\langle \tilde{\pi}, \tilde{\varphi}, \tilde{\varphi}', \tilde{\pi}' \rangle$ in $\mathfrak{g}^{\circledast}$. Since π and π' have the same source, mode, and shape as $\tilde{\pi}$ and $\tilde{\pi}'$, respectively, quadruple $\langle \pi, \varphi, \varphi', \pi' \rangle$ in $\mathfrak{g}'^{\circledast}$ cannot violate the assumption if $\langle \tilde{\pi}, \tilde{\varphi}, \tilde{\varphi}', \tilde{\pi}' \rangle$ did not violate it in $\mathfrak{g}^{\circledast}$.

$\boxed{\{\text{upd}\}}$ Consider the reduction of a destructive assignment $\hat{e} = \ell = \langle \mathbf{c}, \mu, \mathbf{o} \rangle$ to a location ℓ containing the old handle $\langle \mathbf{c}, \mu', \omega \rangle$: In this case, $\mathfrak{g}' = \mathfrak{g} \ominus \mathbf{c} \xrightarrow{\mu'} \omega \ominus \mathbf{c} \xrightarrow{\mu} \mathbf{o} \oplus \mathbf{c} \xrightarrow{\mu'} \mathbf{o}$. The typeability of the redex $\hat{e}$ following from Theorem 6 ensures that $\mu \leq_{\mathrm{m}} \mu'$. Proceed by induction on the number k of elementary conversions from μ to μ', i.e., $\mu \leq^1_{\mathrm{m}} \tilde{\mu}_1 \leq^1_{\mathrm{m}} \tilde{\mu}_2 \ldots \tilde{\mu}_{k-1} \leq^1_{\mathrm{m}} \tilde{\mu}_k = \mu'$. In the base case, $\mu = \mu'$, so that the addition of $\mathbf{c} \xrightarrow{\mu'} \mathbf{o}$ is canceled out by the removal of $\mathbf{c} \xrightarrow{\mu} \mathbf{o}$. The removal $\mathfrak{g}' = \mathfrak{g} \ominus \mathbf{c} \xrightarrow{\mu'} \omega$ creates no new ownership paths, and thus no new region-couplings, and also no new association paths, and thus no new $\rightleftharpoons$-paths. Hence it trivially preserves the assumption.

In the induction step $k \to k+1$, the induction hypothesis guarantees that the assumption holds in $\mathfrak{g}_k = \mathfrak{g}_{k-1} \ominus \mathbf{c} \xrightarrow{\mu'} \omega \ominus \mathbf{c} \xrightarrow{\mu} \mathbf{o} \oplus \mathbf{c} \xrightarrow{\tilde{\mu}_k} \mathbf{o}$. After the final step from $\mathfrak{g}_k$ to $\mathfrak{g}' = \mathfrak{g}_k \ominus \mathbf{c} \xrightarrow{\tilde{\mu}_k} \mathbf{o} \oplus \mathbf{c} \xrightarrow{\mu_{k+1}} \mathbf{o}$, consider the quadruple $\langle \pi, \varphi, \varphi', \pi' \rangle$. Lemma 16 guarantees that there are no new $\rightleftharpoons$-paths φ and φ' in $\mathfrak{g}'^{\circledast}$. Hence the argument for the assumption's preservation by each quadruple $\langle \pi, \varphi, \varphi', \pi' \rangle$ goes exactly the same way like the argument given for the preservation of UO and UH by two ownership paths π and π' in §6.3.1 (see there).

$\boxed{\{\text{ret}\}}$ For the reduction of $\ll$`return` $\langle \mathbf{r}, \mu_{\mathbf{o}}, \mathbf{o} \rangle$`;`$\gg$ in the context of top-level environment with call-link $h_{\mathbf{r}} = \langle \mathbf{s}, \mu_{\mathbf{r}}, \mathbf{r} \rangle$, consider first the replacement $\mathfrak{g}'' = \mathfrak{g} \ominus \mathbf{r} \xrightarrow{\mu_{\mathbf{o}}} \mathbf{o} \ominus \mathbf{s} \xrightarrow{\mu_{\mathbf{r}}} \mathbf{r} \oplus \mathbf{s} \xrightarrow{\mu_{\mathbf{r}} \circ \mu_{\mathbf{o}}} \mathbf{o}$ of the exported handle and the call-link by the imported handle. Lemma 17 guarantees that there are no new $\rightleftharpoons$-paths φ and φ' in $\mathfrak{g}'^{\circledast}$. Hence the argument for the assumption's preservation by each quadruple $\langle \pi, \varphi, \varphi', \pi' \rangle$ goes exactly the same way like the argument given for the preservation of UO and UH by two ownership paths π and π' in §6.3.1 (see there). The final step to $\mathfrak{g}'$, which only removes handles, namely those from the environment's locations, obviously preserves the assumption.

$\boxed{\{\text{call}\}}$ For an operation call expression $\hat{e} = \langle \mathbf{s}, \mu_{\mathbf{r}}, \mathbf{r} \rangle \Leftarrow f(\langle \mathbf{s}, \mu'_1, \mathbf{o}_1 \rangle, \ldots, \langle \mathbf{s}, \mu'_k, \mathbf{o}_k \rangle)$ we need the typeability of the redex $\hat{e}$ and the type-consistency of the object-map following from Theorem 6. The former ensures via the latter that if the modes of the parameters in the receiver's method f are $\mu_1, \ldots, \mu_k$, then each mode μ'_i of a sent handle is compatible to the adaption $\mu_{\mathbf{r}} \circ \mu_i$ of the mode of the corresponding method's parameter relative to the call-link $h_{\mathbf{r}}$: $\mu'_i \leq_{\mathrm{m}} \mu_{\mathbf{r}} \circ \mu_i$.

Proceed by induction on the number k of (non-nil) sent handles $h''_{\mathbf{o}} = \langle \mathbf{s}, \mu'_i, \mathbf{o}_i \rangle$. In the base case $k = 0$, $\mathfrak{g}' = \mathfrak{g} \oplus \mathbf{r} \xrightarrow{\texttt{co<>}} \mathbf{r}$. The only potentially new edge in the new extended graph $\mathfrak{g}'^{\star}$, is $\mathbf{r} \xrightarrow{\texttt{co<>}} \mathbf{r}$. (Note that it is identical to its own inverse $\mathbf{r} \xleftarrow{\texttt{co<>}} \mathbf{r}$.) All new ownership paths and association paths π in $\mathfrak{g}'$ must contain $\mathbf{r} \xrightarrow{\texttt{co<>}} \mathbf{r}$ (at least once). They all have the obvious precursor π' in which all occurrences of the new edge have been cut out. These precursors have the same shape as π, so that these new ownership paths cannot cause a violation of the assumption.

In the induction step $k-1 \to k$, $\mathfrak{g}' = \mathfrak{g}_{k-1} \ominus \mathbf{s} \xrightarrow{\mu'_k} \mathbf{o}_k \oplus \mathbf{r} \xrightarrow{\mu_k} \mathbf{o}_k$, where the assumption holds for $\mathfrak{g}''^{\circledast}$ by induction hypothesis. The intermediate step $\mathfrak{g}'' = \mathfrak{g}_{k-1} \ominus \mathbf{s} \xrightarrow{\mu'_k} \mathbf{o}_k \oplus \mathbf{s} \xrightarrow{\mu_{\mathbf{r}} \circ \mu_k} \mathbf{o}_k$ of converting the k^{th} sent handle's mode μ'_k exactly to the adaption $\mu_{\mathbf{r}} \circ \mu_k$ of the parameter's mode is like the conversion before assignment. The preservation of the assumption under this change was already shown in the $\{\text{upd}\}$-case above. Note the preserved assumption entails in particular $\mathfrak{g}''^{\circledast} \models \mathsf{UH}$.

For the final substep, the actual supply $\mathfrak{g}' = \mathfrak{g}'' \ominus \mathbf{s} \xrightarrow{\mu_{\mathbf{r}} \circ \mu_k} \mathbf{o}_k \oplus \mathbf{r} \xrightarrow{\mu_k} \mathbf{o}_k$, observe first that the multiplicity of all `free` *edges* remains below two: The only edges whose multiplicity is increased in the extension $\mathfrak{g}'^{\star}$ of $\mathfrak{g}'$ are $h_{\mathbf{o}} = \mathbf{r} \xrightarrow{\mu_k} \mathbf{o}_k$ and, if $\mu_k = \texttt{co<>}$, its inverse $h_{\mathbf{o}}^{-1} = \mathbf{o}_k \xrightarrow{\texttt{co<>}} \mathbf{r}$. If μ_k is a `free` mode, then $\mu_{\mathbf{r}} \circ \mu_k$ is `free`. Hence sent handle $h'_{\mathbf{o}} = \mathbf{s} \xrightarrow{\mu_{\mathbf{r}} \circ \mu_k} \mathbf{o}_k$ ensures by the assumption that it is the only old `free` edge targeting $\mathbf{o}_k$ and that its multiplicity is one. But then, after decreasing its multiplicity, no old `free` edge remains in $\mathfrak{g}'$. The multiplicity of new `free` received handle $h_{\mathbf{o}} = \mathbf{r} \xrightarrow{\mu_k} \mathbf{o}_k$ is one. Since the constraint on valid modes allows only `free` edges to be extended to `free` paths, this means that the reason for a violation of the assumption cannot lie in the multiplicity of the initial edge in `free` paths.

Now, consider the quadruple $\langle \pi, \varphi, \varphi', \pi' \rangle$ in $\mathfrak{g}'^{\circledast}$. The ownership paths π and π' and $\rightleftharpoons$-paths φ and φ' possible in $\mathfrak{g}'^{\circledast}$ are described in Lemmas 9 and 18. In $\mathfrak{g}''^{\circledast}$ the source of π's and π''s precursor, witness, or witness, respectively was bridged by a series of n such quadruples. On the way from o to ω, there were h quadruples $\langle \pi_1, \varphi_1, \varphi'_1, \pi'_1 \rangle$, ..., $\langle \pi_{h-1}, \varphi_{h-1}, \varphi'_{h-1}, \pi'_{h-1} \rangle$ and the half-quadruple "$\langle \pi_h, \varphi_h$"; they are complemented by another half-quadruple "$\pi'_h, \varphi'_h \rangle$" and $n-h-1$ quadruples $\langle \pi_{h+1}, \varphi_{h+1}, \varphi'_{h+1}, \pi'_{h+1} \rangle$, ..., $\langle \pi_n, \varphi_n, \varphi'_n, \pi'_n \rangle$ on the way from ω to o':

- For φ there was a $\vec{\alpha}$-qbridge: initial $\tilde{\varphi}_j$, and quadruples $\langle \varphi'_j, \pi'_j, \pi_{j+1}, \varphi_{j+1} \rangle$, ..., $\langle \varphi'_{h-1}, \pi'_{h-1}, \pi_h, \varphi_h \rangle$.
- If π is unchanged or internally-only new, $j = 1$. $\pi_1 = \pi_j$ is π's equivalent precursor and $\varphi_j = \varphi_1$ is $\tilde{\varphi}_j$ from φ.
- If π is internally really new, there was a witness $\pi_1 =_{\text{df}} \mathfrak{wit}(\pi)$, an initial $\rightleftharpoons$-path φ_1, and a triple series transformable to a series of quadruples $\langle \varphi'_1, \pi'_1, \pi_2, \varphi_2 \rangle$, ..., $\langle \varphi'_{j-1}, \pi'_{j-1}, \pi_j, \hat{\varphi}_j \rangle$. In this case, the missing φ_j is the $\rightleftharpoons$-path $q_j \xrightarrow{\vec{\alpha}_j.\vec{\gamma}_j}\!\!/_{\rightleftharpoons}\ \omega_0$ implied by π's $\vec{\alpha}_j.\vec{\gamma}_j$-bridge's last $\rightleftharpoons$-path $\hat{\varphi}_j = q_j \xrightarrow{\vec{\alpha}_j.\vec{\gamma}_j}\!\!/_{\rightleftharpoons}\ q.\vec{\alpha}$ and by φ's initial $\rightleftharpoons$-path $\tilde{\varphi}_j = q \xrightarrow{\vec{\alpha}}\!\!/_{\rightleftharpoons}\ \omega_0$ (Lemma 12).
- If π is initially new, there was a witness $\pi_1 =_{\text{df}} \mathfrak{wit}(\pi)$, an initial $\rightleftharpoons$-path φ_1, and a triple series transformable to a series of quadruples $\langle \varphi'_1, \pi'_1, \pi_2, \varphi_2 \rangle$, ...,

$\langle\varphi'_{j-1}, \pi'_{j-1}, \pi_j, \tilde{\varphi}_j\rangle$. In this case, the missing φ_j is $q_j \xrightarrow{\vec{\alpha}_j.\vec{\gamma}_i}\!\!\not\leftrightharpoons \omega_0$ as in the internally really new case.

If this reservation qbridge has the length one, i.e., $\langle\pi_0, \varphi_0, \varphi'_0, \pi'_0\rangle = \langle\pi_n, \varphi_n, \varphi'_n, \pi'_n\rangle$, then the assumption for $\mathfrak{g}''^{\circledast}$ guarantees that π_0's source o_0 and mode $\hat{\mu}_0$ equals π'_n's source o_{n+1} and mode $\hat{\mu}_{n+1}$. And if $\hat{\mu}_0$ or $\hat{\mu}_{n+1}$ is `free`, then π_0 and π'_n have the same shape. Hence, first, if neither π nor π' are initially new, this means that they have the same source and, if one is `free`, the same initial edge: There no conflict with the assumption. Second, if *ownership* path π (or π') is initially new then its mode μ (or μ') is `free` by $\Gamma_n, \kappa_n \vdash_\epsilon \hat{e} : \hat{\tau}$, and thus the mode $\mu_\mathbf{r}\circ\mu = \hat{\mu}_0$ (or $\mu_\mathbf{r}\circ\mu' = \hat{\mu}_{n+1}$) of its witness $\pi_0 = \mathfrak{wit}(\pi)$ (or $\pi'_n = \mathfrak{wit}(\pi')$) is `free` too. But then both π_0 and π'_n by assumption start with the initial edge of initially new paths: $h'_\mathbf{o}$. If, while π is initially new, π' were unchanged, internally-only new, or internally really new (or vice versa), then π''s unchanged initial edge would by assumption coincide with the initial handle $h'_\mathbf{o}$ of π's witness $\mathfrak{wit}(\pi)$. But as initial edge of `free` path π, $h'_\mathbf{o}$'s multiplicity of one (guaranteed by the assumption) decreases to zero in $\mathfrak{g}'^{\circledast}$; it cannot be unchanged. Hence π and π' must be both initially new. Since their witnesss π_0 and π'_n by assumption have the same shape $\mathbf{s} \xrightarrow{\mu_\mathbf{r}\circ\mu_k} \mathbf{o}_k \xrightarrow{\texttt{co}}{}^* \bullet \dashrightarrow^{\vec{\alpha}} \bullet$, π and π' have the same shape $\mathbf{r} \xrightarrow{\mu_k} \mathbf{o}_k \xrightarrow{\texttt{co}}{}^* \bullet \dashrightarrow^{\vec{\alpha}} \bullet$, and thus the same source and mode.

If there is a longer series of quadruples $\langle\pi_i, \varphi_i, \varphi'_i, \pi'_i\rangle$, the assumption for $\mathfrak{g}''^{\circledast}$ guarantees for each that π_i's source o_i equal π'_i's source o_{i+1} and mode $\hat{\mu}_{i+1}$. That is, $o_0 = \mathbf{s} = o_{n+1}$ and $\hat{\mu}_0 = \hat{\mu}_i = \ldots = \hat{\mu}_{n+1}$. And if $\hat{\mu}_i$ or $\hat{\mu}_{i+1}$ is `free`, then π_i and π'_i have the same initial edge $u \xrightarrow{\tilde{\mu}} v$ by assumption, and thus the same shape $u \xrightarrow{\tilde{\mu}} v \xrightarrow{\texttt{co}}{}^* \bullet \dashrightarrow^{\epsilon} \bullet$ by the nesting constraint on modes which does not allow for extending paths by association paths to `free` paths. The way how π'_i's and π_{i+1}'s shape are related by Lemma 18, this means that only the case of $\mu_k = $ `co<>` is possible every ownership path π_i and π'_i has shape $\mathbf{s} \xrightarrow{\mu_\mathbf{r}\circ\mu_k} \mathbf{o}_k \xrightarrow{\texttt{co}}{}^* \bullet \dashrightarrow^{\epsilon} \bullet$ or $\mathbf{s} \xrightarrow{\mu_\mathbf{r}} \mathbf{r} \xrightarrow{\texttt{co}}{}^* \bullet \dashrightarrow^{\epsilon} \bullet$.

Hence, if neither π nor π' are initially new, all of this means that they have the same source. Additionally, if one of them is `free`, they both can only have the shape $\mathbf{s} \xrightarrow{\mu_\mathbf{r}} \mathbf{r} \xrightarrow{\texttt{co}}{}^* \bullet \dashrightarrow^{\epsilon} \bullet$, i.e., start with $h_\mathbf{r} = \mathbf{s} \xrightarrow{\mu_\mathbf{r}} \mathbf{r}$ since the alternative, `free` $h'_\mathbf{o} = \mathbf{s} \xrightarrow{\mu_\mathbf{r}\circ\mu_k} \mathbf{o}_k$ would not be unchanged but reduced to zero multiplicity in $\mathfrak{g}'^{\circledast}$. There is no conflict with the assumption. And if ownership path π (or π') were initially new, then $\mu_k \neq$ `co<>` and $\pi_0 = \mathfrak{wit}(\pi)$ (or $\pi'_n = \mathfrak{wit}(\pi')$) would be `free` and start with $h'_\mathbf{o}$. But this would contradict the $\mu_k = $ `co<>` necessary for `free` ownership paths π'_i's and π_{i+1} in an internal quadruple of the series (see above). ■

6.3.6 Conclusion

To conclude this section, we now after much work return to the ownership theorem that started this section, and obtain it as corollary from Lemma 23:

Theorem 7 If $e_0,\ \eta_0, \mathfrak{s}_0, om_0, \mathfrak{g}_0 \Longrightarrow^* e',\ \vec{\eta}', \mathfrak{s}', om', \mathfrak{g}'$ is a reduction defined relative to a program p with $\vdash p$ **start** e_0 then

$$\mathfrak{g}' \models \mathsf{UH}, \mathsf{UO}$$

Proof: Lemma 23 guarantees the reserved ownership assumption about quadruples $\langle\tilde{\pi}, \tilde{\varphi}, \tilde{\varphi}', \tilde{\pi}'\rangle$ in $\mathfrak{g}'^{\circledast}$ (cf. Definition 13). UO and UH are included as the special case where $\tilde{\pi} \in PAP_{\mathfrak{g}'}(w, \tilde{\mu}, v)$ and $\tilde{\pi}' \in PAP_{\mathfrak{g}'}(w', \tilde{\mu}', v)$, and where $\tilde{\varphi}$ and $\tilde{\varphi}'$ are the trivial $\rightleftharpoons$-path $v \overset{\epsilon}{\dashrightarrow}/_{\rightleftharpoons}\ v$. ■

6.4 Structural Integrity of Mutator Access

Since the ownership paths through which mutators can be called are essentially the same as in base-JaM—a `free` or `rep` handle followed by `co`-handles—no new proof needs to be developed for properties Mutator Control Path and Mutator Control.

Theorem 8 If $e_0,\ \eta_0, \mathfrak{s}_0, om_0, \mathfrak{g}_0 \Longrightarrow^* e',\ \vec{\eta}', \mathfrak{s}', om', \mathfrak{g}'$ is a reduction defined relative to a program p with $\vdash p$ **start** e_0 then

$$\mathfrak{g}', \vec{\eta}' \models \mathsf{MCP}, \mathsf{MC}$$

Proof: The same potential access paths are—modulo correlations—ownership paths as in base-JaM, and mutator calls are subject to the—modulo correlations—same condition $\mu_\mathbf{r} \in \mathbf{Wr}(\kappa)$ as in base-JaM. Hence it easy to convince oneself that the proof for MCP in base-JaM (Theorem 3) literally is a proof for MCP in JaM. The obvious exception is that one has to use, instead of base-JaM's Proposition 2 and Theorem 1, the respective Proposition 5 and Theorem 6 of JaM. Then one only has to read "if $\mu_\mathbf{r}$ is `free` or `rep`" as "if $\mu_\mathbf{r}$ has base-mode `free` or `rep`," and "$\mu_\mathbf{r}$ can be `co`" as "$\mu_\mathbf{r}$ can be `co<>`."

The sanctuary and $\mathbf{Wr}(\kappa)$ are—modulo correlations—defined the same as in base-JaM. The proof for MC in base-JaM (Theorem 4) nearly is a proof for MC in JaM: Instead of base-JaM's Theorem 3 and Proposition 2, one uses JaM's Theorem 8 and Proposition 5. And one expands "$PAP(o, \texttt{rep}, \omega)$" and "$PAP(o, \texttt{free}, \omega)$" to "$PAP(o, \texttt{rep<...>}, \omega)$" and "$PAP(o, \texttt{free<...>}, \omega)$," and expands "$\mu_\mathbf{r}$ cannot be `free`," "$\mu_\mathbf{r} = \texttt{rep}$," and "$\mu_\mathbf{r} = \texttt{co}$" to "$\mu_\mathbf{r}$ cannot be `free<...>`," "$\mu_\mathbf{r} = \texttt{rep<...>}$," and "$\mu_\mathbf{r} = \texttt{co<>}$," respectively. ■

6.5 Composite State Encapsulation

This section shows that the representative's control over mutator executions (mutator control) does indeed entail the desired control over any change of the composite state (composite state encapsulation). The main structure of the proof of composite state encapsulation in JaM is the same as in base-JaM. Lemmas on shallow state encapsulation and coherence are used, which are developed further below (Lemmas 24 and 25). The coherence aspect is a much more complicated affair in JaM than it was in base-JaM: The `rep` paths in the field subgraph, which define the membership in the state representation, may pass, via captured `read` handles, through objects that do not belong to the state representation. A large technical lemma (Lemma 26) is necessary to prove that all these intermediate objects are at least members of the sanctuary or they are immutable. This means that a change of the composite state can be affected not only by objects in the state representation but by any object in the sanctuary. Still, this weaker form of coherence in JaM is sufficient for composite state encapsulation since all members of the sanctuary are mutator controlled. The technical lemma's proof will make use of one constraint that has been specifically introduced to this end and has played no role up to now: Non-destructive read of `free` handles out of variables is permitted only if the variable is local in an observer. In conjunction with it, the proof will for the first time exploit the fact that `return` steps decrease the multiplicity of the handles in the terminated environment.

Theorem 9 (Composite state encapsulation) If $e_0,\ \eta_0, \mathfrak{s}_0, om_0, \mathfrak{g}_0 \Longrightarrow^* e,\ \vec{\eta}, \mathfrak{s}, om, \mathfrak{g} \Longrightarrow e',\ \vec{\eta}', \mathfrak{s}', om', \mathfrak{g}'$ is a reduction defined relative to a program p with $\vdash p\ \ \mathtt{start}\ e_0$ then for all $o \in \mathrm{dom}(om)$,

$$CState_{\mathfrak{s},om}(o) \neq CState_{\mathfrak{s}',om'}(o) \ \Rightarrow\ \exists i \leq n.\ \mathbf{r}_i = o\ \wedge\ \kappa_i = \mathtt{mut}$$

where $\vec{\eta} = {\eta_1}_{h_1}^{\kappa_1}, \ldots, {\eta_n}_{h_n}^{\kappa_n}$ with $h_i = \langle \mathbf{s}_i, \mu_i, \mathbf{r}_i \rangle$.

Proof: $CState_{\mathfrak{s},om}(o) \neq CState_{\mathfrak{s}',om'}(o)$

$$\begin{array}{ll}
\overset{\text{Lemma 25}}{\Longrightarrow} & \exists \omega \in \{o\} \cup Sanc_{\mathfrak{g}}(o).\ \mathfrak{s}\,|_{flds_{om}(\omega)} \neq \mathfrak{s}'\,|_{flds_{om}(\omega)} \\
\overset{\text{Lemma 24}}{\Longrightarrow} & \exists \omega \in \{o\} \cup Sanc_{\mathfrak{g}}(o).\ \mathbf{r}_n = \omega \quad \wedge\ \kappa_n = \mathtt{mut} \\
\Longrightarrow & \mathbf{r}_n \in \{o\} \cup Sanc_{\mathfrak{g}}(o) \qquad\qquad \wedge\ \kappa_n = \mathtt{mut} \\
\Longrightarrow & (\mathbf{r}_n = o \wedge \kappa_n = \mathtt{mut})\ \vee\ (\mathbf{r}_n \in Sanc_{\mathfrak{g}}(o)\ \wedge\ \kappa_n = \mathtt{mut}) \\
\overset{\text{Theorem 8}}{\Longrightarrow} & (\mathbf{r}_n = o \wedge \kappa_n = \mathtt{mut})\ \vee\ (\exists i \leq n.\ \mathbf{r}_i = o\ \wedge\ \kappa_i = \mathtt{mut}) \\
\Longrightarrow & \exists i \leq n.\ \mathbf{r}_i = o\ \wedge\ \kappa_i = \mathtt{mut}
\end{array}$$

■

Lemma 24 If $e_0,\ \eta_0, \mathfrak{s}_0, om_0, \mathfrak{g}_0 \Longrightarrow^* e,\ \vec{\eta}, \mathfrak{s}, om, \mathfrak{g} \Longrightarrow e',\ \vec{\eta}', \mathfrak{s}', om', \mathfrak{g}'$ is a reduction defined relative to a program p with $\vdash p\ \ \mathtt{start}\ e_0$ then for $\vec{\eta} = {\eta_1}_{h_1}^{\kappa_1}, \ldots, {\eta_n}_{h_n}^{\kappa_n}$ with $h_n = \langle \mathbf{s}_n, \mu_n, \mathbf{r}_n \rangle$, and for all $\omega \in \mathrm{dom}(om)$,

$$\mathfrak{s}\,|_{flds_{om}(\omega)} \neq \mathfrak{s}'\,|_{flds_{om}(\omega)} \ \Rightarrow\ \mathbf{r}_n = \omega\ \wedge\ \kappa_n = \mathtt{mut}$$

Proof: The proof is literally the same as the proof of its base-JaM version (Lemma 4), with the obvious exception that JaM's Theorem 6 has to be used instead of base-JaM's Theorem 1. ■

Lemma 25 (Coherence) If $e_0, \eta_0, \mathfrak{s}_0, om_0, \mathfrak{g}_0 \Longrightarrow^* e, \vec{\eta}, \mathfrak{s}, om, \mathfrak{g} \Longrightarrow e', \vec{\eta}', \mathfrak{s}', om', \mathfrak{g}'$ is a reduction defined relative to a program p with $\vdash p$ **start** e_0 then

$$CState_{\mathfrak{s},om}(o) \neq CState_{\mathfrak{s}',om'}(o) \;\Rightarrow\; \exists \omega \in \{o\} \cup Sanc_{\mathfrak{g}}(o) \bullet\ \mathfrak{s}\,|_{\mathit{flds}_{om}(\omega)} \neq \mathfrak{s}'\,|_{\mathit{flds}_{om}(\omega)}$$

So far only the source, mode and target of a potential access path or $\rightleftharpoons$-path have been of interest. For the proof of the coherence lemma, one has to look at all the intermediate handles' objects since each of them can destroy or create the potential access path or $\rightleftharpoons$-path by capturing or overwriting a handle in a field. The difficult case are the new class of `rep` paths in JaM that are extensions like $\mathsf{d} \xrightarrow{\texttt{read<dest=rep<...>>}} \mathsf{i} \overset{\texttt{dest<>}}{\dashrightarrow} \texttt{e1}$ of paths by association paths since these may have prefixes that are not `rep` paths. There is not even a guarantee that in the field subgraph any path from o to intermediate object i constitutes a sequence of ownership paths. For such objects it has to be shown that their handles that are, or can be extended to, association paths are used by third objects for their `rep` paths only in ways safe for coherence.

An object o has its sub-objects, i.e., objects reachable from o by sequences of ownership paths, for storing `rep` handles or handles extensible to `rep` handles. (Actually passing them down requires appropriate correlations on the ownership paths, so that only certain sub-objects can really be used for this—this refinement will become relevant on page 176.) The storage is *safe* w.r.t. coherence if it is through a `rep` path since their `rep` sub-objects are protected in o's sanctuary. The safety of storage in a `free` sub-object depends on the initial `free` handle: It should be captured in a field. Then, in order to call the mutator on the `free` sub-object, the initial handle has to be taken out of the field by destructive read, which requires a mutator in the source. We can filter out these safe sub-objects if we index the sub-object set $Sub(o)$ with the set H of the initial handles of the used `free` paths. The subset of o's sub-objects safe for the storage of `rep` paths then is $Sub_H(o)$ for some $H \subseteq fgr_{om}(\mathfrak{s})$.

$$\begin{aligned} Sub_H(o) =_{\mathrm{df}} & \bigcup_{PAP_{\mathfrak{g}}(o,\texttt{rep<}\delta\texttt{>},q)\neq\emptyset} && \{q\} \cup Sub_H(q) \\ \cup\ & \bigcup_{\hbar\in H} \bigcup_{\hbar\bullet\,\tilde{\pi}\in PAP_{\mathfrak{g}}(o,\texttt{free<}\delta\texttt{>},q)} && \{q\} \cup Sub_H(q) \end{aligned}$$

Moreover, one can easily see that it is safe if `rep` paths pass through objects that are effectively immutable (in a shallow sense), meaning that they never change their fields. There are different kinds of reasons for an object to be (im)mutable: It can be a consequence of the methods offered, or of the program executions possible. We are concerned here only with immutability due to the lack of permission for calling mutators (as captured in the mutator access properties), and for accessing the field containing the handle for making the call: To be ***legally mutable*** in JaM, $lmut(\omega)$, requires to have a legally mutable owner o, or to be reachable by a `free` path with an initial handle h not captured in a field (then it does not matter whether owner o is legally mutable or not):

$$\begin{aligned} \mathit{lmut}(\omega) \Leftrightarrow_{\mathrm{df}} & \exists o\text{.}\ \mathit{Osh}_{\mathfrak{g}}(o,\omega) \neq \emptyset \ \wedge\ \mathit{lmut}(o) \\ & \vee\ \exists o, \delta, h, \tilde{\pi}\text{.}\ \hbar \bullet \tilde{\pi} \in \mathit{PAP}_{\mathfrak{g}}(o, \texttt{free<}\delta\texttt{>}, \omega)\ \wedge\ \hbar \notin \mathit{fgr}_{om}(\mathfrak{s}) \end{aligned}$$

Proof of the lemma: The beginning of the proof are like in base-JaM (Lemma 3): A composite state change means a change of a restriction of the store:

$$\mathfrak{s}\,|_{\bigcup_{\omega \in \mathit{StRep}_{\mathfrak{s},om}(o)} \mathit{flds}_{om}(\omega)} \neq \mathfrak{s}'\,|_{\bigcup_{\omega \in \mathit{StRep}_{\mathfrak{s}',om'}(o)} \mathit{flds}_{om'}(\omega)}$$

If the domain of the restriction is unchanged then the composite state change means that the store changed somewhere in $\bigcup_{\omega \in \mathit{StRep}_{\mathfrak{s},om}(o)} \mathit{flds}_{om}(\omega) = \bigcup_{\omega \in \mathit{StRep}_{\mathfrak{s}',om'}(o)} \mathit{flds}_{om'}(\omega)$. It changed at some $\ell \in \mathit{flds}_{om}(\omega)$ of some $\omega \in \mathit{StRep}_{\mathfrak{s},om}(o)$: $\mathfrak{s}(\ell) \neq \mathfrak{s}'(\ell)$. Hence $\mathfrak{s}\,|_{\mathit{flds}_{om}(\omega)} \neq \mathfrak{s}'\,|_{\mathit{flds}_{om}(\omega)}$ for this ω.

Next consider a change $\bigcup_{\omega \in \mathit{StRep}_{\mathfrak{s},om}(o)} \mathit{flds}_{om}(\omega) \neq \bigcup_{\omega \in \mathit{StRep}_{\mathfrak{s}',om'}(o)} \mathit{flds}_{om'}(\omega)$. Since the set of field locations of each "old" object $\omega \in \mathit{StRep}_{\mathfrak{s},om}(o)$ remains unchanged in any reduction step ($\mathit{flds}_{om}(\omega) = \mathit{flds}_{om'}(\omega)$), such a change presupposes a change in the set of state-representing implementation objects. That is, $\mathit{StRep}_{\mathfrak{s},om}(o) \neq \mathit{StRep}_{\mathfrak{s}',om'}(o)$. This expands to

$$\{o\} \cup \bigcup_{\mathit{PAP}_{\mathit{fgr}_{om}(\mathfrak{s})}(o,\texttt{rep},q) \neq \emptyset} \mathit{StRep}_{\mathfrak{s},om}(q) \neq \{o\} \cup \bigcup_{\mathit{PAP}_{\mathit{fgr}_{om'}(\mathfrak{s}')}(o,\texttt{rep},q) \neq \emptyset} \mathit{StRep}_{\mathfrak{s}',om'}(q)$$

There must be an object q that is reachable from o by a non-empty sequence $o = o_0 \overset{\texttt{rep}}{\dashrightarrow} o_1 \overset{\texttt{rep}}{\dashrightarrow} \dots \overset{\texttt{rep}}{\dashrightarrow} o_n = \omega$ of `rep` paths in field subgraph $\mathit{fgr}_{om}(\mathfrak{s})$ but no such sequence in $\mathit{fgr}_{om'}(\mathfrak{s}')$, or vice versa. Paths $\pi = o_j \overset{\texttt{rep}}{\dashrightarrow} o_{j+1}$ in the field subgraph are created by assigning one of π's handles into a field (capture), and are destroyed by updating a field containing one of its handles by assignment or destructive read (overwrite). Each of these objects must belong to the composite object o (as state representation or otherwise), or be effectively immutable. In terms of *lmut* and *Sub* developed above,

$$\begin{aligned} & q_0 \xrightarrow{\mu_0} q_1 \dots q_k \xrightarrow{\mu_k} q_{k+1} \in \mathit{PAP}_{\mathit{fgr}_{om}(\mathfrak{s})}(o_j, \texttt{rep<}\delta\texttt{>}, o_{j+1}) \\ & \Rightarrow \forall i \in \{1, \dots, k\}\text{.}\ \neg \mathit{lmut}(q_i)\ \vee\ \exists H \subseteq \mathit{fgr}_{om}(\mathfrak{s})\text{.}\ q_i \in \mathit{Sub}_H(o_j) \end{aligned}$$

This property will be shown below in Corollary 2 to be an invariant of legal JaM program executions and thus hold in particular in $\mathfrak{s}, om, \mathfrak{g}$. An object q_i that is not legally mutable, $\neg\mathit{lmut}(q_i)$, has no owner or is in the sanctuary of an object without owner. Hence it cannot execute any mutators (Mutator Control Path, Theorem 8) and thus its fields cannot change (Shallow State Encapsulation, Lemma 24). An object q_i that is reachable through a `free` path $\tilde{\pi}$ with initial handle $\hbar$ stored in a field cannot occur as call-link in the environment stack since $\hbar$ is unique (Theorem 7). But then q_i cannot be executing any mutators (Mutator Control Path) and thus its fields cannot change (Shallow State Encapsulation). This leaves only objects q_i in o_j's sanctuary, or o_j itself, to capture or overwrite a handle $q_i \xrightarrow{\mu_i} q_{i+1}$ in assignment and destructive read steps. Only they can create or destroy rep paths $\pi = q_0 \xrightarrow{\mu_0} q_1 \ldots q_k \xrightarrow{\mu_k} q_{k+1} \in PAP_{fgr_{om}(\mathfrak{s})}(o_j, \texttt{rep<...>}, o_{j+1})$ in the field subgraph. Consequently, the object q_i whose field change $(\mathfrak{s}\,|_{flds_{om}(q_i)} \neq \mathfrak{s}'\,|_{flds_{om}(q_i)})$ creates or destroys a path $\pi = o_j \dashrightarrow^{\texttt{rep}} o_{j+1}$ in sequence $o = o_0 \dashrightarrow^{\texttt{rep}} o_1 \dashrightarrow^{\texttt{rep}} \ldots \dashrightarrow^{\texttt{rep}} o_n = \omega$ of rep paths in the field subgraph is in o's sanctuary $(q_i \in Sanc(o_j) \wedge o_j \in Sanc(o) \Rightarrow q_i \in Sanc(o))$. ■

The invariant used above needs to be refined to strengthen it for a preservation proof.

Step 1. Provision has to be taken for showing the preservation of the invariant's "$q_i \in Sub_H(o)$" case when one of the captured free handles $\hbar \in H \subseteq fgr_{om}(\mathfrak{s})$ is read destructively out of its field: Since this makes q_i mutable without o's control, this is safe only if the destructive read simultaneously interrupts the rep path $\pi \in PAP_{fgr_{om}(\mathfrak{s})}(o, \texttt{rep<}\delta\texttt{>}, \omega)$ that passed through q_i *in the field subgraph view.* To this end, field-captured rep paths π through sub-objects q_i in field-captured free sub-objects $\hat{q}$ are required to contain the corresponding free path's initial handle. They have to pass through all the free handles $\hbar \in H$ on which $q_i \in Sub_H(o)$ was based: The condition $H \Subset \{\pi\}$ has to be added. Actually, since all handles in π are in the field subgraph, it can *replace* the condition $H \subseteq fgr_{om}(\mathfrak{s})$.

Step 2. In order to prepare for step *3* with rep paths not only in the field subgraph and uncaptured free handles (that can be exchanged as parameter and result), we have to be more precise about the free sub-objects in which rep handles and handles extensible to rep paths can actually be stored: o can store them through a free path in its direct free sub-objects only if the free path's mode $\hat{\mu}$ contains a (nested) correlation of rep mode, i.e., $\hat{\mu}(\vec{\gamma}) = \texttt{rep}$ for some $\vec{\gamma}$. And o's sub-objects q can store such handles in their free sub-objects only if the handles are association handles or can be extended to association paths and if the free path's mode $\hat{\mu}$ contains a correlation to the corresponding association mode, i.e., $\hat{\mu}(\vec{\gamma}) = \beta \in \mathbb{A}$.

That is, as non-immutable intermediate objects in rep paths in the field subgraph only those objects $u \in Sub_H(o)$ should be accepted where all free handles $\hbar \in H$ on the way from o to u have a mode $\hat{\mu}$ with $\hat{\mu}(\vec{\gamma}) \in \{\texttt{rep}\} \cup \mathbb{A}$. Using a corresponding predicate $repdn(u \xrightarrow{\hat{\mu}} v) \Leftrightarrow_{\mathrm{df}} \exists \vec{\alpha}\bullet\ \hat{\mu}(\vec{\gamma}) \in \{\texttt{rep}\} \cup \mathbb{A}$, the invariant for all objects q_i in o's rep paths π in $fgr_{om}(\mathfrak{s})$ now reads

$$\neg lmut(q_i) \ \vee\ \exists H \Subset \{\pi\}\bullet\ q_i \in Sub_H(o) \wedge \forall \hbar \in H\bullet\, repdn(\hbar)$$

Step 3. Showing preservation of the invariant about rep paths in the field subgraph under the capturing of handles in fields requires us to know already something similar about *all* rep paths $\pi \in PAP_{\mathfrak{g}}(o, \mu, \omega)$ in $\mathfrak{g}$. Also these are safe if they pass through only objects u safe for rep paths in the field subgraph. But we have to deal with an additional possibility specific to uncaptured rep paths π: Naturally, an *uncaptured* rep path π should be able to pass through sub-objects q_i of o reachable not only through captured but also *uncaptured* free paths, and through sub-objects q_i of immutable $\hat{q}$ reachable not only through captured `free` paths (which would make q_i immutable too) but also through *uncaptured* `free` paths (which means q_i is a mutable non-sub-object of o). All this can be safely allowed if the uncaptured initial edges $\hbar$ of the `free` paths are edges of π: Then π can only only show up in the field subgraph if all the `free` initial edges were captured—so that we are back to the condition on `rep` paths in the field subgraph.

That is, the condition on the objects q_i in any `rep` path π should be

$$\begin{aligned}&\neg lmut(q_i)\\ &\vee\ \exists H \Subset \{\pi\}\bullet\ q_i \in Sub_H(o) \wedge \forall \hbar \in H\bullet repdn(\hbar)\\ &\vee\ \exists \hat{q}\bullet\ \neg lmut(\hat{q})\ \wedge\ \exists H \Subset \{\pi\}\bullet\ q_i \in Sub_H(\hat{q}) \wedge \forall \hbar \in H\bullet repdn(\hbar)\end{aligned}$$

But this is not all.

Step 4. We also have to deal with the possibility that π contains not the uncaptured initial handle $\hbar$ leading to q_i, but its `read` copy h created by non-destructive read or handles h to objects to which the copy was passed. Such a `rep` path π is safe if h is an uncaptured edge local to an observer invocation, because then h blocks π's showing up in the field subgraph. In order to handle return steps that return h from an observer back into a mutator, it has to be clarified that h and $\hbar$ are "*observer bounded*," $obsbd(h, \hbar)$, in the following sense: h and $\hbar$ are uncaptured handles; h is local to call-levels l in which an observer is executing; and (the unique `free` handle) $\hbar$ is local to a call-level l' and either l' is above l, or all call-levels from l' up to l are executing observers. Then when h is returned into a mutator, $\hbar$ must be in the terminated invocation, so that its destruction by return leaves q_i as an immutable object. The uncaptured `free` handle $\hbar$ may be passed through another `free` handle $\hbar'$ to an object and captured there. But then we will have $obsbd(h, \hbar')$ instead.

That is, the condition $\exists H \Subset \{\pi\}\bullet\ q_i \in Sub_H(o) \wedge \forall \hbar \in H\bullet repdn(\hbar)$ needs to be expanded to $\exists H\bullet\ q_i \in Sub_H(o) \wedge \forall \hbar \in H\bullet repdn(\hbar) \wedge (\{\hbar\} \Subset \{\pi\} \vee \exists h\bullet \{h\} \Subset \{\pi\} \wedge obsbd(h, \hbar))$, and the same for the $q_i \in Sub_H(\hat{q})$ case. To make the extended invariant easier to handle, its formulation will be restructured: For each object u in any `rep` path π of o there is an H such that

$$\begin{aligned}&\neg lmut(u)\ \vee\ u \in Sub_H(o)\ \vee\ \exists \hat{q}\bullet\ \neg lmut(\hat{q}) \wedge u \in Sub_H(\hat{q})\\ &\wedge\ \forall \hbar \in H\bullet\ repdn(\hbar)\ \wedge\ \big(\{\hbar\} \Subset \{\pi\}\ \vee\ \exists h\bullet\ \{h\} \Subset \{\pi\} \cup \Pi \wedge obsbd(h, \hbar)\big)\end{aligned}$$

where $obsbd(h, \hbar)$ is formalized as follows using the kinds κ_i of the methods executing at the various call-levels in the environment stack $\vec{\eta} = {\eta_1}_{h_1}^{\kappa_1} \ldots {\eta_n}_{h_n}^{\kappa_n}$, and using the receivers $\mathbf{r}_i$ and modes $\hat{\mu}_i$ of the call-links $h_i = \langle \mathbf{s}_i, \hat{\mu}_i, \mathbf{r}_i \rangle$ in it:

$$\begin{aligned}obsbd(o \xrightarrow{\mu} \omega, \hbar) &\Leftrightarrow_{\mathrm{df}} o \xrightarrow{\mu} \omega, \hbar \notin fgr_{om}(\mathfrak{s})\ \wedge\ \mu \neq \texttt{co<>}\\ &\qquad \wedge\ \forall l, l'\bullet\ atlevel(o \xrightarrow{\mu} \omega, l)\ \Rightarrow\ \kappa_l = \texttt{obs}\\ &\qquad\qquad \wedge\ atlevel(\hbar, l') \Rightarrow \forall i\bullet l' \leq i < l \Rightarrow \kappa_i = \texttt{obs}\\ atlevel(h, l) &\Leftrightarrow_{\mathrm{df}} h \in \mathrm{im}(\mathfrak{s}\,|_{\mathrm{im}(\eta_l)})\ \vee\ \mathcal{E}_l = \ldots h \ldots\ \vee\ (l < n \wedge h = h_{l+1})\end{aligned}$$

$$
\begin{array}{llll}
RSub_H(o) & =_{\mathrm{df}} & \bigcup_{res(o,\mathtt{rep<}\delta\mathtt{>},q)} & \{q\} \cup RSub_H(q) \\
 & \cup & \bigcup_{\hbar \in H} \bigcup_{res(\hbar,\mathtt{free<}\delta\mathtt{>},q)} & \{q\} \cup RSub_H(q) \\
RFree(h) & =_{\mathrm{df}} & \bigcup_{res(h,\mathtt{free<}\delta\mathtt{>},q)} & \{q\} \\
rmut(\omega) & \Leftrightarrow_{\mathrm{df}} & \exists o, \delta\bullet & rmut(o) \ \wedge \ (res(o, \mathtt{rep<}\delta\mathtt{>}, \omega) \vee res(o, \mathtt{free<}\delta\mathtt{>}, \omega)) \\
 & \vee & \exists h, \delta\bullet & h \notin fgr_{om}(\mathfrak{s}) \ \wedge \ res(h, \mathtt{free<}\delta\mathtt{>}, \omega) \\
res(o, \mu, \omega) & \Leftrightarrow_{\mathrm{df}} & \exists q, \vec{\gamma}\bullet & PAP_{\mathfrak{g}^{\circledast}}(o, \mu, q.\vec{\gamma}) \neq \emptyset \ \wedge \ q \overset{\vec{\gamma}}{\dashrightarrow}_{/\rightleftharpoons} \omega \\
res(h, \mu, \omega) & \Leftrightarrow_{\mathrm{df}} & \exists o, \tilde{\pi}, q, \vec{\gamma}\bullet & h \bullet \tilde{\pi} \in PAP_{\mathfrak{g}^{\circledast}}(o, \mu, q.\vec{\gamma}) \wedge q \overset{\vec{\gamma}}{\dashrightarrow}_{/\rightleftharpoons} \omega
\end{array}
$$

Figure 6.12: Auxiliary definitions for Lemma 26

The auxiliary predicate *atlevel*(h, l) symbolizes that handle h occurs at call-level l, either as the value of a local variable in environment η_i, or as a temporary value in the runtime term at method nesting level l, written $h \in \mathcal{E}_l$, or as the call-link h_{l+1} from call-level l to the next. The $\mathcal{E}_l$ are determined by the decomposition of current runtime term e using a series of reduction context $\mathcal{E}_1, \ldots, \mathcal{E}_{n-1} \in R_1^{\square}$ and an innermost runtime term $\mathcal{E}_n$ containing no inlined method body, so that $e = \mathcal{E}_1[\ll \ldots [\ll \mathcal{E}_{n-1}[\ll \mathcal{E}_n \gg] \gg] \ldots \gg]$.

Step 5. The final step is to move to the graphs $\mathfrak{g}^{\circledast}$ and $fgr_{om}(\mathfrak{s})^{\circledast}$ extended by region objects, and to generalize ownership paths paths π to *ownership paths reservations* $\langle \pi, \varphi \rangle$. This is necessary for showing the invariant's preservation under the supply of parameters. The generalization to reservations applies, on one hand, the considered **rep** path π—all objects in π *and* in φ's path-base Π are subjected to a condition—and, on the other hand, to the ownership paths π defining sub-object-relationships and (im)mutability in the condition. To this end, figure 6.12 defines the "reservation closure"-induced generalizations $rmut(\omega)$ of $lmut(\omega)$, $RSub_H(o)$ of $Sub_H(o)$, $res(o, \mu, \omega)$ of $PAP(o, \mu, \omega) \neq \emptyset$, and $res(h, \mu, \omega)$ of $h \bullet \tilde{\pi} \in PAP(o, \mu, \omega)$. $RFree(h)$ is also defined, which is needed below for the expression of the auxiliary invariant on ***free*** reservations.

Lemma 26 If $e_0, \eta_0, \mathfrak{s}_0, om_0, \mathfrak{g}_0 \Longrightarrow^* e, \vec{\eta}, \mathfrak{s}, om, \mathfrak{g}$ is a reduction defined relative to a program p with $\vdash p$ **start** e_0 then

$$
\begin{array}{l}
h_1 \bullet \tilde{\pi} \in PAP_{\mathfrak{g}^{\circledast}}(o, \mathtt{rep<}\ldots\mathtt{>}, q.\vec{\gamma}) \ \wedge \ q \overset{\vec{\gamma}}{\dashrightarrow}_{/\rightleftharpoons} \omega \ \mathbf{via}\ \Pi \\
\Rightarrow \forall \{v \xrightarrow{\mu} w\} \Subset \{\tilde{\pi}\} \cup \Pi\bullet\ \forall u \in \{v, w\}\bullet\ \exists H\bullet \\
\quad \neg rmut(u) \ \vee \ u \in RSub_H(o) \ \vee \ \exists \hat{q}\bullet\ \neg rmut(\hat{q}) \wedge u \in RSub_H(\hat{q}) \\
\quad \wedge \ \forall \hbar \in H\bullet\ repdn(\hbar) \ \wedge \ \Big(\{\hbar\} \Subset \{\pi\} \ \vee \ \exists h\bullet\ \{h\} \Subset \{\pi\} \cup \Pi \wedge obsbd(h, \hbar)\Big)
\end{array}
$$

The invariant as it was used in Lemma 25 is implied by this:

Corollary 2

$$
\begin{array}{l}
q_0 \xrightarrow{\mu_0} q_1 \ldots q_k \xrightarrow{\mu_k} q_{k+1} \in PAP_{fgr_{om}(\mathfrak{s})}(o, \texttt{rep<}\delta\texttt{>}, \omega) \\
\Rightarrow \forall i \in \{1, \ldots, k\}\bullet\ \neg lmut(q_i)\ \vee\ \exists H \subseteq fgr_{om}(\mathfrak{s})\bullet\ q_i \in Sub_H(o)
\end{array}
$$

Proof: Any rep path $\pi \in PAP_{fgr_{om}(\mathfrak{s})}(o, \texttt{rep<...>}, \omega)$ in the field subgraph is extended to a reservation $\langle\pi, \varphi\rangle$ by trivial $\rightleftharpoons$-path $\varphi = \omega \dashrightarrow^{\epsilon}_{\rightleftharpoons} \omega$ **via** $\emptyset$. Since $\{\pi\} \cup \Pi \Subset fgr_{om}(\mathfrak{s})$, the alternative with $obsbd(h, \hbar)$ never applies. All $\hbar \in H$ are in π, hence $H \Subset fgr_{om}(\mathfrak{s})$. Obviously, the lemma's $\neg rmut(u)$ implies $\neg lmut(u)$. $u \in RSub_H(o)$ or $u \in RSub_H(\hat{q})$ implies $u \in Sub_H(o)$ or $Sub_H(\hat{q})$, respectively, if there is a connection by ownership *paths*. Otherwise the bridge of ownership reservations on the way from o or $\hat{q}$ to u contains a right-most reservation that is not an ownership path. Its target $\hat{q}'$ is $\neg lmut(\hat{q}')$. If $\hat{q}'$ is u, then $\neg lmut(u)$. And if $u \in RSub_H(\hat{q})$ or $RSub_H(\hat{q}')$ with not legally mutable $\hat{q}$ or $\hat{q}'$, then also u is not legally mutable since $H \Subset fgr_{om}(\mathfrak{s})$. ■

The preservation of the lemma's invariant under conversion of handles from free to rep requires a similar invariant about free reservations $\langle\pi, \varphi\rangle$. These are harder to reason about than rep reservations because free handles can be exchanged between objects. Here it is crucial that the nesting constraint on modes excludes correlations to free modes on top of that to co<>. This lets free paths be like in base-JaM: a free edge followed by co-edges. Hence the intermediate objects u in free *paths* starting with h_1 are all targeted by free paths starting with h_1. For free *reservations*, this has to be generalized to the objects $u \in RFree(h_1)$ to which free reservations exist with initial handle h_1 (cf. figure 6.12), and to objects that are immutable:[8]

$$
\begin{array}{l}
h_1 \bullet \tilde{\pi} \in PAP_{\mathfrak{g}^{\circledast}}(o, \texttt{free<...>}, q.\vec{\gamma})\ \wedge\ q \dashrightarrow^{\vec{\gamma}}_{\rightleftharpoons} \omega\ \textbf{via}\ \Pi \\
\Rightarrow \forall\{v \xrightarrow{\mu} w\} \Subset \{\tilde{\pi}\} \cup \Pi\bullet\ \forall u \in \{v, w\}\bullet \\
\quad \neg rmut(u)\ \vee\ u \in RFree(h_1)
\end{array}
$$

Proof of the lemma: The lemma's invariant on rep paths and the auxiliary invariant on free paths are handled simultaneously by induction on the number N of reduction steps from e_0 to e. In the base case $N = 0$, $\mathfrak{g} = \mathfrak{g}_0 = \emptyset$ is empty. Hence there can be no ownership paths, so that the invariant holds trivially. In the induction step $N \to N+1$, reduction $e_0, \eta_0, \mathfrak{s}_0, om_0, \mathfrak{g}_0 \Longrightarrow^* e_N, \vec{\eta}_N, \mathfrak{s}_N, om_N, \mathfrak{g}_N$ is continued $e_N, \vec{\eta}_N, \mathfrak{s}_N, om_N, \mathfrak{g}_N \Longrightarrow e, \vec{\eta}, \mathfrak{s}, om, \mathfrak{g}$. In most steps the two invariants holding by induction hypothesis are obviously preserved. For the other steps we will make use of the *typeability* of redex $\hat{e}$ that follows from Theorem 6, and of the *reserved ownership assumption* that Lemma 23 guarantees (without mention of the theorem/lemma).

[8]Since no proper region object can be targeted by free paths made only of free edges and co-edges, φ must is an ϵ-path. One would expect it to be trivial and have an empty path-base $\Pi = \emptyset$. However, this can be shown not to be the case. But it suffices, and is more uniform to the treatment of rep reservations, to say that all objects in π and Π are in $RFree(h_1)$ or immutable.

$\boxed{\{\text{if}_t\},\ \{\text{if}_f\}}$ The only relevant change is the potential removal of the compared handles by a multiplicity decrease from one to zero. This cannot create new ownership paths nor $\rightleftharpoons$-paths, thus preserving all old case of $\neg \mathit{rmut}(u)$. The removal may however interrupt an old ownership path π or $\rightleftharpoons$-path φ, and destroy an ownership reservation $\langle \pi, \varphi \rangle$. If this was the last ownership reservation on the target object ω, then ω is now immutable. Together with ω, all objects u become immutable that are reachable from ω via sequences of reservations $\langle \pi_i, \varphi_i \rangle$ with `rep` path π_i or with `free` path π_i with field-captured initial handle. Consequently, whenever a relationship $u \in \mathit{RFree}(h)$ ceases to hold by the loss of ownership reservations, u is now immutable instead. And if a relationship $u \in \mathit{RSub}_H(o)$ ceases to hold, there must be an intermediate object v in the ownership reservation connection from o to u i.e., with $v \in \mathit{RSub}_{H''}(o)$ and $u = v$ or $u \in \mathit{RSub}_{H'}(v)$ for some for $H', H'' \subseteq H$, that is not reserved any more. This makes v immutable, and perhaps also u. The invariants are preserved.

$\boxed{\{\text{rd}_{dst}\}}$ Where a handle is stored is only relevant in the definition of *rmut*. Hence destructive read access might have an effect if a field location is read that contains a `free` handle h. But shallow state encapsulation (Lemma 24) guarantees that the destructive access to fields happens only within current object $\mathbf{r}$'s mutators. Hence the mutator access properties (Theorem 8) imply that $\mathbf{r}$ is mutable. Thus so are all its sub-objects, in particular, the targets of $\mathbf{r}$'s `free` paths based on h, and their sub-objects. The invariants are preserved.

$\boxed{\{\text{rd}_{cp}\}}$ The only thing that might change in the object graph by non-destructive read access is the addition of a `read` handle $h_\mathbf{o} = \hbar_\mathbf{o}[\texttt{read}/\texttt{free}]$ for a `free` handle $\hbar_\mathbf{o}$ in the variable. All new paths π' have a precursor $\pi = \pi'[\hbar_\mathbf{o}/h_\mathbf{o}]$ with the same base-mode or base-mode `read` instead of `free`. Hence all region-couplings and $\rightleftharpoons$-paths φ' **via** Π' have precursors with path-base $\Pi = \Pi'[\hbar_\mathbf{o}/h_\mathbf{o}]$. There is no change in $\mathit{RFree}(h)$, $\mathit{RSub}_H(o)$, nor $\mathit{rmut}(\omega)$. Hence for all objects u in the new reservation $\langle \pi', \varphi' \rangle$, the old reservation $\langle \pi, \varphi \rangle$ guarantees by induction hypothesis that u is immutable, or in $\mathit{RFree}(h_1)$, or in $\mathit{RSub}_H(\hat{q})$ of an object $\hat{q}$ that is o or immutable. From all $\hbar \in H$ we know that there is an h in $\langle \pi, \varphi \rangle$ with $h = \hbar \vee \mathit{obsbd}(h, \hbar)$. In $\langle \pi', \varphi' \rangle$, this h still exists if it is not $\hbar_\mathbf{o}$. And if h was $\hbar_\mathbf{o}$, then in $\langle \pi', \varphi' \rangle$ we have $h_\mathbf{o}$ instead. The typeability of redex $\hat{e}$ ensures that `read`-alias $h_\mathbf{o}$ is created only in an *observer* and only from handle $\hbar_\mathbf{o}$ in the location of a *local* variable: $\mathit{obsbd}(h_\mathbf{o}, \hbar_\mathbf{o})$. Hence the invariants are preserved.

$\boxed{\{\text{new}\}}$ In an object creation step, a `free` edge $h_\mathbf{o} = \mathbf{r} \xrightarrow{\mu_\mathbf{o}} \mathbf{o}$ to a fresh object is added: $\mathfrak{g} = \mathfrak{g}_N \oplus \mathbf{r} \xrightarrow{\mu_\mathbf{o}} \mathbf{o}$.

First. Before looking at the intermediate objects in ownership reservations, consider which reservations and which relevant relationships are new, and which are lost. The results are summarized in figure 6.13:

Since no edges are removed, all old reservations are preserved and thus all old relationships $u \in \mathit{RFree}(h)$ and $u \in \mathit{RSub}_H(o)$. The only really new ownership paths

in $\mathfrak{g}_N{}^{\circledast}$		in $\mathfrak{g}^{\circledast}$
$u \in \mathit{RFree}(h),\ u \in \mathit{RSub}_H(o)$	$\Rightarrow$	$u \in \mathit{RFree}(h),\ u \in \mathit{RSub}_H(o)$
$\neg\mathit{rmut}(u) \wedge u \neq \mathbf{o}.\vec{\alpha}$	$\Rightarrow$	$\neg\mathit{rmut}(u)$
$\neg\mathit{rmut}(\mathbf{r}.\mu_{\mathbf{o}}(\vec{\alpha}).\vec{\alpha}')$	$\Rightarrow$	$\neg\mathit{rmut}(\mathbf{o}.\vec{\alpha}.\vec{\alpha}')$
$\mathbf{r}.\mu_{\mathbf{o}}(\vec{\alpha}).\vec{\alpha}' \in \mathit{RFree}(h),\ \mathit{RSub}_H(o)$	$\Rightarrow$	$\mathbf{o}.\vec{\alpha}.\vec{\alpha}' \in \mathit{RFree}(h),\ \mathit{RSub}_H(o)$
$\mu_{\mathbf{o}}(\vec{\alpha}) = \mathtt{read}$	$\Rightarrow$	$\neg\mathit{rmut}(\mathbf{o}.\vec{\alpha})$

Figure 6.13: Change of relationships in creation steps

target some $\mathbf{o}.\vec{\alpha}$ (Lemma 6), and the only really new $\rightleftharpoons$-paths have some $\mathbf{o}.\vec{\alpha}$ as source or as target, but not both (Lemma 15). Since there can be no old ownership paths to fresh $\mathbf{o}$ and its region objects, unchanged π can combine with new φ only if it differs from its counterpart $\sigma(\varphi)$ by having target $\mathbf{o}.\vec{\alpha}$. New π can combine with an *unchanged* φ on $\mathbf{o}$'s side, but φ's target will always be the same one as π's, i.e., some $\mathbf{o}.\vec{\alpha}$. Internally new π can combine with a really new φ if it differs from its counterpart $\sigma(\varphi)$ in its source; but their counterparts can combine to a reservation $\langle\sigma(\pi), \sigma(\varphi)\rangle$ that precedes $\langle\pi, \varphi\rangle$. Hence objects u other than $\mathbf{o}$ and its region objects do not become reachable by really new ownership reservations. For these objects, $\neg\mathit{rmut}(u)$ is preserved.

All ownership reservations on a region object $\mathbf{o}.\vec{\alpha}.\vec{\alpha}'$ are witnessed by ownership reservations on $\mathbf{r}.\mu_{\mathbf{o}}(\vec{\alpha}).\vec{\alpha}' = \sigma(\mathbf{o}.\vec{\alpha}.\vec{\alpha}')$. Hence if sequences of old ownership reservations on $\mathbf{r}.\mu_{\mathbf{o}}(\vec{\alpha}).\vec{\alpha}'$ left it immutable or placed it in $\mathit{RFree}(h)$ or $\mathit{RSub}_H(o)$ in $\mathfrak{g}_N{}^{\circledast}$, then the corresponding new sequences of ownership reservations on $\mathbf{o}.\vec{\alpha}.\vec{\alpha}'$ respectively leave it immutable or place it in $\mathit{RFree}(h)$ or $\mathit{RSub}_H(o)$.

Consider the objects $\mathbf{o}.\vec{\alpha}$ where $\mu_{\mathbf{o}}(\vec{\alpha}) = \mathtt{read}$: The nesting constraint on valid modes like $\mu_{\mathbf{o}}$ ensures that no extraction $\mu_{\mathbf{o}}(\vec{\alpha}')$ for a prefix of $\vec{\alpha}$ can be an association mode. Hence the only non-dummy path targeting $\mathbf{o}.\vec{\alpha}$ is the initially new path $\pi_{\vec{\alpha}}$ of a $\mathtt{read}$ mode. $\mathbf{o}.\vec{\alpha}$ has no owner. Moreover, there is no chance to construct any ownership reservation on $\mathbf{o}.\vec{\alpha}$ (see above). $\mathbf{o}.\vec{\alpha}$ is immutable.

Second, consider the objects in reservations $\langle\pi, \varphi\rangle$ with unchanged ownership path π or unchanged $\rightleftharpoons$-path φ: An unchanged π cannot pass through fresh $\mathbf{o}$ and its region objects, and thus can combine only with unchanged $\rightleftharpoons$-paths, which also do not pass through fresh $\mathbf{o}$ and its region objects. Since nothing changed for these objects, reservations with unchanged π still satisfy the invariants.

The only unchanged $\rightleftharpoons$-paths that can combine with a new ownership path π are those based on dummy edge sequences $\mathbf{o}.\alpha_1 \ldots \alpha_k \xrightarrow{\alpha_{k+1}\cdots\alpha_n} \mathbf{o}.\vec{\alpha}$, since all new π end in some $\mathbf{o}.\vec{\alpha} = (\mathbf{o}.\alpha_1 \ldots \alpha_k).\alpha_{k+1} \ldots \alpha_n$. These have the same target as π and contain only edges already contained in $\pi = h_{\mathbf{o}} \bullet \mathbf{o} \xrightarrow{\vec{\alpha}} \mathbf{o}.\vec{\alpha}$, or $\pi = \pi_1 \bullet h_{\mathbf{o}} \bullet \mathbf{o} \xrightarrow{\vec{\alpha}} \mathbf{o}.\vec{\alpha}$. Hence these reservations are safe iff the new ownership path π is safe. We can ignore the objects in old ownership paths and old $\rightleftharpoons$-paths, and concentrate on the (objects in) in new ownership paths (in combination with new or old $\rightleftharpoons$-paths) and in (the path-bases of) new $\rightleftharpoons$-paths (in combination with new or old ownership paths).

Third. Consider the objects u in new ownership paths π in reservations $\langle\pi, \varphi\rangle$:

(a) π can be a *initially new* ownership path $\pi_{\vec{\alpha}} = h_{\mathbf{o}} \bullet \mathbf{o} \xrightarrow{\vec{\alpha}} \mathbf{o}.\vec{\alpha}$ with initial edge $h_1 = h_{\mathbf{o}}$: In the case `free` $\pi_{\vec{\alpha}}$, the nesting constraint on valid modes like $\mu_{\mathbf{o}}$ ensures that $\vec{\alpha} = \epsilon$. Hence there is only $\mathbf{o} = \mathbf{o}.\epsilon$ in $\pi_{\vec{\alpha}}$, which is obviously in $\mathit{RFree}(h_{\mathbf{o}})$. In the case of `rep` path $\pi_{\vec{\alpha}}$, $\mu_{\mathbf{o}}(\vec{\alpha}) =$ `rep` the nesting constraint ensures that all its non-trivial prefixes are initially new `free`, `rep`, or `read` paths $\pi_{\vec{\alpha}'}$ of modes $\mu_{\vec{\alpha}'}$ with $\mu_{\vec{\alpha}'}(\vec{\alpha}'') =$ `rep`. In the first two cases, we have $\mathbf{o}.\vec{\alpha}' \in \mathit{RSub}_{\emptyset}(\mathbf{r})$ for $\pi_{\vec{\alpha}}$'s intermediate object $\mathbf{o}.\vec{\alpha}'$, and for the latter case it was shown above that $\mathbf{o}.\vec{\alpha}'$ is immutable.

(b) If π is a *internally new* ownership path $\pi_1 \bullet \pi_{\vec{\alpha}} \bullet \pi_3$ with $\mu_{\mathbf{o}}(\vec{\alpha}) = \beta$, it has a witness $\sigma(\pi) = \pi_1 \bullet \pi_2'$ of the same mode with $\pi_2' = \mathbf{r} \xrightarrow{\beta.\vec{\alpha}'} \mathbf{r}.\beta.\vec{\alpha}'$. Since π contains non-initially the non-`co` handle $h_{\mathbf{o}}$, the nesting constraint on modes means that it cannot be `free`. π and $\sigma(\pi)$ must be a `rep` paths. Since in $\mathfrak{g}_N{}^{\circledast}$, there was the old reservation $\langle \sigma(\pi), \mathbf{r}.\beta.\vec{\alpha}' \overset{\epsilon}{\dashrightarrow}_{\not\rightleftharpoons} \mathbf{r}.\beta.\vec{\alpha}' \rangle$, the induction hypothesis covers all objects in $\sigma(\pi)$. Since the reservation is old, the objects in $\sigma(\pi)$ still satisfy the invariant in $\mathfrak{g}^{\circledast}$, as shown above. Observe that in the third and fourth case, the handle h guaranteed to be in $\sigma(\pi)$ must be in the π_1-segment common with π since the π_2'-segment consists of dummy edges that are neither `free` nor absent from the *extended* graph $\mathit{fgr}_{om_N}(\mathfrak{s}_N)^{\circledast}$.

1. Hence the objects u in the π_1-segment of π satisfy the invariant in $\mathfrak{g}^{\circledast}$.
2. The objects $u = \mathbf{o}.\vec{\alpha}'$ in the $\pi_{\vec{\alpha}}$ segment of π are immutable if $\mu_{\mathbf{o}}(\vec{\alpha}') =$ `read` (see above). The case of $\mu_{\mathbf{o}}(\vec{\alpha}') \in \{$`free`, `rep`$\}$ is covered by looking at the possibilities which the induction hypothesis guarantees for $\mathbf{r}$ in `rep` path $\sigma(\pi)$. There are no other cases of $\mu_{\mathbf{o}}(\vec{\alpha}')$: $\mu_{\mathbf{o}}$ must contain a (nested) correlation to β by $\mu_{\mathbf{o}}(\vec{\alpha}) = \beta$, so that the nesting constraint on valid mode $\mu_{\mathbf{o}}$ ensures that all extractions $\mu_{\mathbf{o}}(\vec{\alpha}')$ by a prefix $\vec{\alpha}'$ of $\vec{\alpha}$ are `free`, `rep`, or `read`.
   First, $\mathbf{r}$ may be immutable. Then in the `rep`-case also the target $u = \mathbf{o}.\vec{\alpha}'$ of its initially new `rep` path $\pi_{\vec{\alpha}'}$ is immutable. In the `free` case, we have $u \in \mathit{RSub}_{\{h_{\mathbf{o}}\}}(\mathbf{r})$ with immutable $\mathbf{r}$ and with $h_{\mathbf{o}}$ in π and $\mathit{repdn}(h_{\mathbf{o}})$ since $\mu_{\vec{\alpha}'}(\vec{\alpha}'') = \beta$. Second, $\mathbf{r} \in \mathit{RSub}_H(\hat{q})$. Hence initially new ownership path $\pi_{\vec{\alpha}'}$ from $\mathbf{r}$ to u of mode $\mu_{\vec{\alpha}'}$ means that also $u \in \mathit{RSub}_{H'}(\hat{q})$ with $H' = H$ in the `rep` case and, in the `free` case, $H' = H \cup \{h_{\mathbf{o}}\}$ with $h_{\mathbf{o}}$ in π and $\mathit{repdn}(h_{\mathbf{o}})$ since $\mu_{\vec{\alpha}'}(\vec{\alpha}'') = \beta$.
3. The objects $u = \mathbf{o}.\vec{\alpha}.\vec{\alpha}''$ in the π_3 segment of π are immutable, in $\mathit{RFree}(h_1)$, in $\mathit{RSub}_H(o)$ or in $\mathit{RSub}_H(\hat{q})$ since the objects $\sigma(u) = \mathbf{r}.\beta.\vec{\alpha}''$ in the π_2'-segment of $\sigma(\pi)$ are by induction hypothesis guaranteed to be immutable, in $\mathit{RFree}(h_1)$, in $\mathit{RSub}_H(o)$ or in $\mathit{RSub}_H(\hat{q})$ (see above). Whether free $\hbar \in H$ is in $\sigma(\pi)$ or there is an uncaptured handle h in $\sigma(\pi)$ with $\mathit{obsbd}(h, \hbar)$, this handle still exists in π, as shown above.

Fourth. Consider the objects u in the path-base Π of *new* $\rightleftharpoons$-path $\varphi = q \overset{\vec{\gamma}}{\dashrightarrow}_{\not\rightleftharpoons} \omega$ **via** Π. Observe that $h_{\mathbf{o}}$ must be in Π since otherwise φ could not be new: $\{h_{\mathbf{o}}\} \Subset \Pi$. It has a precursor $\sigma(\varphi) = \sigma(q) \overset{\vec{\gamma}}{\dashrightarrow}_{\not\rightleftharpoons} \sigma(\omega)$ **via** $\Pi' \equiv \sigma(\Pi \setminus \Pi_{\mathfrak{N}} \setminus \Pi_{\circledast})$. This precursor exists also with a path-base that includes the dummy edges from $\mathbf{r}$ to $\sigma(q)$ or $\sigma(\omega) = \mathbf{r}.\mu_{\mathbf{o}}(\vec{\alpha}).\vec{\alpha}'$, respectively: The dummy edge sequence $\mathbf{r} \xrightarrow{\mu_{\mathbf{o}}(\vec{\alpha}).\vec{\alpha}'} \mathbf{r}.\mu_{\mathbf{o}}(\vec{\alpha}).\vec{\alpha}'$ as a $\rightleftharpoons$-path means the region-coupling $\mathbf{r}.(\mu_{\mathbf{o}}(\vec{\alpha}).\vec{\alpha}').\epsilon \rightleftharpoons (\mathbf{r}.\mu_{\mathbf{o}}(\vec{\alpha}).\vec{\alpha}').\epsilon$ **via** $\Pi'' = \{\mathbf{r} \xrightarrow{\mu_{\mathbf{o}}(\vec{\alpha})} \mathbf{r}.\mu_{\mathbf{o}}(\vec{\alpha})$, $\mathbf{r}.\mu_{\mathbf{o}}(\vec{\alpha}) \xrightarrow{\alpha_1'} \mathbf{r}.\mu_{\mathbf{o}}(\vec{\alpha}).\alpha_1'$, $\ldots$, $\mathbf{r}.\mu_{\mathbf{o}}(\vec{\alpha}).\alpha_1' \ldots \alpha_{n-1}' \xrightarrow{\alpha_n'} \mathbf{r}.\mu_{\mathbf{o}}(\vec{\alpha}).\vec{\alpha}'\}$. This redundant region-coupling extends the path-base of trivial $\mathbf{r}.\mu_{\mathbf{o}}(\vec{\alpha}).\vec{\alpha}' \overset{\epsilon}{\dashrightarrow}_{/\rightleftharpoons} \mathbf{r}.\mu_{\mathbf{o}}(\vec{\alpha}).\vec{\alpha}'$ **via** $\emptyset$ to the $\rightleftharpoons$-path $\varphi_\epsilon = \mathbf{r}.(\mu_{\mathbf{o}}(\vec{\alpha}).\vec{\alpha}').\epsilon \overset{\epsilon}{\dashrightarrow}_{/\rightleftharpoons} \mathbf{r}.\mu_{\mathbf{o}}(\vec{\alpha}).\vec{\alpha}'$ **via** Π''. Its redundant concatenation with $\sigma(\varphi)$ to $\varphi' = \varphi_\epsilon \bullet \sigma(\varphi)$ or $\sigma(\varphi) \bullet \varphi_\epsilon$, respectively, extends the path-base to $\Pi' \cup \Pi''$.

(a) In the case of unchanged π, new φ must be new at its target, i.e., $\sigma(q) = q$ and $\omega = \mathbf{o}.\vec{\alpha}.\vec{\alpha}'$ with $\sigma(\omega) = \mathbf{r}.\mu_{\mathbf{o}}(\vec{\alpha}).\vec{\alpha}'$. Then π combined with counterpart $\varphi' = q \overset{\vec{\gamma}}{\dashrightarrow}_{/\rightleftharpoons} \mathbf{r}.\mu_{\mathbf{o}}(\vec{\alpha}).\vec{\alpha}'$ **via** $\Pi' \cup \Pi''$ to a reservation $\langle \pi, \varphi' \rangle$ in $\mathfrak{g}_N{}^{\circledast}$ to which the induction hypothesis applies.

(b) In the case of new π, π cannot be initially new since initially new ownership paths do not reach region objects $\mathbf{o}.\vec{\alpha}$ with $\mu_{\mathbf{o}}(\vec{\alpha}) \in \mathbb{A}$ or beyond and thus cannot combine with new $\rightleftharpoons$-paths. Internally new π can combine with new $\varphi = \mathbf{o}.\vec{\alpha}.\vec{\alpha}' \overset{\vec{\gamma}}{\dashrightarrow}_{/\rightleftharpoons} \omega$ **via** Π. If φ is of the third kind in Lemma 15, it is the combination of an old $\rightleftharpoons$-path $\varphi_1 = \mathbf{o}.\vec{\beta} \overset{\vec{\gamma}_1}{\dashrightarrow}_{/\rightleftharpoons} \mathbf{o}.\vec{\alpha}$ and a $\rightleftharpoons$-path $\varphi_2 = \mathbf{o}.\vec{\alpha} \overset{\vec{\gamma}_2}{\dashrightarrow}_{/\rightleftharpoons} \omega$ with counterpart $\varphi_2' = \mathbf{r}.\mu_{\mathbf{o}}(\vec{\alpha}).\vec{\alpha}' \overset{\vec{\gamma}}{\dashrightarrow}_{/\rightleftharpoons} \sigma(\omega)$ **via** $\Pi' \cup \Pi''$, where $\vec{\gamma} = \vec{\gamma}_1 \bullet \vec{\gamma}_2$. All edges of φ_1 must also be contained in π with target $\mathbf{o}.\vec{\alpha}.\vec{\alpha}' = \mathbf{o}.\vec{\beta}.\vec{\gamma}_1.\vec{\gamma}_2$. Hence their sources and targets are already covered. The reservation $\langle \pi, \varphi_2 \rangle$ with just φ_2 is equivalent to the original reservation $\langle \pi, \varphi \rangle$, and it contains no new edges: $\{\pi\} \cup \Pi_2 \equiv \{\pi\} \cup \Pi$. It thus suffices to look just at $\rightleftharpoons$-paths of the first kind: Its counterpart φ_2' combined with internally new π's witness $\sigma(\pi) = \sigma(o) \overset{\mu}{\dashrightarrow} \mathbf{r}.\mu_{\mathbf{o}}(\vec{\alpha}).\vec{\alpha}'.\vec{\gamma}$ to a reservation $\langle \sigma(\pi), \sigma(\varphi) \rangle$ in $\mathfrak{g}_N{}^{\circledast}$ to which the induction hypothesis applies.

In both cases, the $\rightleftharpoons$-path in the respective reservation $\langle \pi, \varphi' \rangle$ or $\langle \sigma(\pi), \varphi' \rangle$ has the same source o and initial edge h_1 like π. Consider what all this means for the objects u in new φ's path-base Π:

1. If u is the source or target of a handle $h \notin \Pi_{\mathbb{N}} \cup \Pi_{\circledast}$ then its precursor $\sigma(u)$ is the source or target, respectively, of $\sigma(h)$ in Π' and thus covered by the induction hypothesis. In the case of $u \neq \mathbf{o}.\vec{\alpha}$, $\sigma(u) = u$, so that the induction hypothesis covers u directly: u is immutable, in $\mathit{RFree}(h_1)$ or in $\mathit{RSub}_H(\hat{q})$. Otherwise, $u = \mathbf{o}.\vec{\alpha}.\vec{\alpha}'$ such that $\sigma(u) = \mathbf{r}.\mu_{\mathbf{o}}(\vec{\alpha}).\vec{\alpha}'$. But what the induction hypothesis guarantees for $\sigma(u)$ in $\mathfrak{g}_N{}^{\circledast}$, also holds for u in $\mathfrak{g}^{\circledast}$, as explained at the beginning of the {new}-case.
2. If u is the source $\mathbf{r}$ of the initial handle $h_{\mathbf{o}}$ in an initially new ownership path $\hat{\pi} \in \Pi_{\mathbb{N}}$, then it is covered as the source of $\mathbf{r} \xrightarrow{\mu_{\mathbf{o}}(\vec{\alpha})} \mathbf{r}.\mu_{\mathbf{o}}(\vec{\alpha})$ in φ''s path-base $\Pi' \cup \Pi''$.
3. The other objects in initially new ownership paths $\hat{\pi} \in \Pi_{\mathbb{N}}$ and dummy edges $\hat{\pi} \in \Pi_{\circledast}$ are region objects $u = \mathbf{o}.\vec{\alpha}'$ with $\mu_{\mathbf{o}}(\vec{\alpha}'.\vec{\alpha}'') = \beta$. As in the case of objects in internally new ownership paths, If $\mu_{\mathbf{o}}(\vec{\alpha}') = \texttt{read}$ then u is immutable. And if $\mu_{\mathbf{o}}(\vec{\alpha}') \in \{\texttt{free}, \texttt{rep}\}$, the possibilities which the induction hypothesis allows for $\mathbf{r}$ in Π'' (see above) imply that u is immutable like $\mathbf{r}$ or, in the `rep` case, in $\mathit{RSub}_H(\hat{q})$ like $\mathbf{r}$, or, in the `free` case, in $\mathit{RSub}_{H \cup \{h_{\mathbf{o}}\}}(\hat{q})$ with $\mathit{repdn}(h_{\mathbf{o}})$ if $\mathbf{r}$ is in $\mathit{RSub}_H(o)$.

Observe that in all cases of $u \in \mathit{RSub}_H(\hat{q})$ or $\mathit{RSub}_{H \cup \{h_{\mathbf{o}}\}}(\hat{q})$, the handle h guaranteed to be in the old reservation for every $\hbar \in H$ exists also in $\langle \pi, \varphi \rangle$: If h is in

in $\mathfrak{g}'^{\circledast}$		in $\mathfrak{g}''^{\circledast}$
$\neg rmut(u)$	$\Rightarrow$	$\neg rmut(u)$
$u \in RFree(h)$	$\Rightarrow$	$u \in RFree(h) \;\vee\; u \in RSub_{\emptyset}(\mathbf{c}) \;\vee\; \neg rmut(u)$
$u \in RSub_H(o)$	$\Rightarrow$	$u \in RSub_{H \setminus \{h_\mathbf{o}\}}(o) \;\vee\; \neg rmut(u)$ $\vee\; \exists v, H' \subseteq H \setminus \{h_\mathbf{o}\}\textbf{.}\; \neg rmut(v) \wedge u \in RSub_{H'}(v)$

Figure 6.14: Change of relationships in conversion steps

$\{\pi\} \cup \Pi' \cup \Pi''$ or $\{\sigma(\pi)\} \cup \Pi' \cup \Pi''$, respectively, it may first be in π or $\sigma(\pi)$, and thus in π, as shown above for $\sigma(\pi)$. Neither `free` $h = \hbar$ nor uncaptured h with $obsbd(h, \hbar)$ can be a dummy edge, so that it cannot be in Π''. Hence h can otherwise only be in $\Pi' \equiv \sigma(\Pi \setminus \Pi_{\mathbb{N}} \setminus \Pi_{\circledast})$, and thus in $\sigma(\Pi) \supseteq \sigma(\Pi \setminus \Pi_{\mathbb{N}} \setminus \Pi_{\circledast})$. Source v and target w of non-dummy edge h cannot be an object $\mathbf{r}.\mu_\mathbf{o}(\vec{\alpha}).\vec{\alpha}'$. But then $\sigma(v) = v$ and $\sigma(w) = w$, so that $\hbar$ and h are not only in $\sigma(\Pi)$ but also in Π, as the invariants demand it.

$\boxed{\{\text{upd}\}}$ Assignment steps are divided, as usually, into several substeps for the purpose of the proof. The substep that decreases of the multiplicity of the old handle at the left-hand side location ℓ in the store preserves the invariants. The argument is the same as in the case of $\{\text{if}_t\}/\{\text{if}_f\}$-steps.

The main aspect of assignment lies in the right-hand handle's conversion from a mode $\tilde{\mu}$ to a mode $\tilde{\mu}'$. The typeability of redex $\hat{e}$ ensures that $\tilde{\mu} \leq_\mathrm{m} \tilde{\mu}'$. We need to look here only at one *elementary* conversion substeps $\mathfrak{g}'' = \mathfrak{g}' \oplus \mathbf{c} \xrightarrow{\mu'} \mathbf{o} \ominus \mathbf{c} \xrightarrow{\mu} \mathbf{o}$, i.e., the substitution of an edge $h'_\mathbf{o} = \mathbf{c} \xrightarrow{\mu'} \mathbf{o}$ for an edge $h_\mathbf{o} = \mathbf{c} \xrightarrow{\mu} \mathbf{o}$ with $\mu \leq^1_\mathrm{m} \mu'$.

First, all ownership paths $\pi \in PAP_{\mathfrak{g}'^{\circledast}}(o, \mu, q.\vec{\gamma})$ have a precursor $\pi' = \sigma(\pi) \in PAP_{\mathfrak{g}'^{\circledast}}(o, \mu', q.\vec{\gamma})$ of the same mode $\mu' = \mu$ or directly compatible mode $\mu' \leq^1_\mathrm{m} \mu$ (Lemma 7), where $\sigma = [h_\mathbf{o}/h'_\mathbf{o}]$ substitutes $h_\mathbf{o}$ handles in in φ's path-base to $h'_\mathbf{o}$ in its precursor's path-base. By the definition of $\leq^1_\mathrm{m}$, the mode $\mu \geq^1_\mathrm{m} \mu'$ cannot be `free` and can be `rep` only if μ' was `free`. Hence neither $h'_\mathbf{o}$ nor—by the nesting constraint on valid modes—any of its extensions can be `free`. All `free` paths π in $\mathfrak{g}''^{\circledast}$ are unchanged ($\pi' = \pi$), and all `rep` paths π are unchanged or have a `rep` precursor $\sigma(\pi)$ of the same mode or have a `free` precursor $\sigma(\pi)$ with the same correlations. Also all $\rightleftharpoons$-paths $\varphi = q \dashrightarrow^{\vec{\gamma}}_{\not\rightleftharpoons} \omega$ **via** Π have a precursor $\varphi' = q \dashrightarrow^{\vec{\gamma}}_{\not\rightleftharpoons} \omega$ **via** $\sigma(\Pi)$ (Lemma 16). Hence for all reservations $\langle \pi, \varphi \rangle$ in $\mathfrak{g}''^{\circledast}$, there was already a reservation $\langle \pi', \varphi' \rangle$ of the same or directly compatible mode in $\mathfrak{g}'^{\circledast}$.

This has the implications summarized in figure 6.14: Without really new reservations, immutability is preserved. Old reservations are preserved, they change from `free` to `rep`, or they are lost. In particular, old cases of $u \in RFree(h)$ remain so, change to $u \in RSub(\mathbf{c})$ (if $h = h_\mathbf{o}$ was `free` and $h'_\mathbf{o}$ is `rep`), or u becomes immutable. Old cases of $u \in RSub_H(o)$ may be preserved, or the ownership reservation connection from o to u was severed, or a `free` reservation with initial $h_\mathbf{o} \in H$ in the connection was converted to a `rep` reservation. In the latter case, $u \in RSub_{H \setminus \{h_\mathbf{o}\}}(o)$ now. In the second case, an object v is not reserved any more that was on the way from o to u, i.e., with $v \in RSub_{H''}(o)$ and $u = v$ or $u \in RSub_{H'}(v)$ in $\mathfrak{g}'^{\circledast}$ for some for $H', H'' \subseteq H$.

The loss of reservation makes v immutable, perhaps also u.

Second. For all objects u in each precursor $\langle\sigma(\pi),\varphi\rangle$ with φ **via** $\sigma(\Pi)$ of a $\mathfrak{g}''^{\circledast}$-reservation $\langle\pi,\varphi\rangle$ with φ **via** Π, the induction hypothesis allows three possibilities. Note that these are the same objects as in $\langle\pi,\varphi\rangle$ since the substitution of $h_{\mathbf{o}}$ for $h'_{\mathbf{o}}$ does not change which objects are in the paths.

- If u was immutable, then it remains so in $\mathfrak{g}''^{\circledast}$.
- If u was in $\mathit{RFree}(h_1)$ then it remains so, or becomes immutable, or switches to $u \in \mathit{RSub}(o)$ if $h_1 = h_{\mathbf{o}}$, so that π's source o is $\mathbf{c}$.
- There was a $\hat{q}$ that was o or immutable and $u \in \mathit{RSub}_H(\hat{q})$ in $\mathfrak{g}'^{\circledast}$ with a h in $\hat{\pi}$ or $\hat{\Pi}$ for every $\hbar \in H$ such that $h = \hbar \vee \mathit{obsbd}(h,\hbar)$. In $\mathfrak{g}''^{\circledast}$, u is immutable or $u \in \mathit{RSub}_{H\setminus\{h_{\mathbf{o}}\}}(\hat{q})$ or there is an immutable $\hat{q}'$ with $u \in \mathit{RSub}_{H'}(\hat{q}')$ and $H' \subseteq H\setminus\{h_{\mathbf{o}}\}$. In other words, u is immutable or there is an object $\hat{q}'$ that is o or immutable and $u \in \mathit{RSub}_{H'}(\hat{q}')$ with $H' \subseteq H \setminus \{h_{\mathbf{o}}\}$. Since H' contains only edges of H and does not contain the changed edge $h_{\mathbf{o}}$, for every $\hbar \in H'$ there is a $\sigma(h)$ in $\hat{\pi}$ or $\hat{\Pi}$ such that $\sigma(h) = h = \hbar \vee \mathit{obsbd}(\sigma(h),\hbar)$.

Third. At the end of all elementary conversion substeps, the fully converted handle $h''_{\mathbf{o}}$ is stored at location ℓ. Storage is relevant for the invariants' preservation in two cases: (1) ℓ is a field location and $h''_{\mathbf{o}}$ is the uncaptured handle in a reservation $\langle\pi,\varphi\rangle$ in a case with $\mathit{obsbd}(h''_{\mathbf{o}},\hbar)$. But a handle in this case is guaranteed to exist only local to observers, so that shallow state encapsulation (Lemma 24) excludes assignment to a field. (2) ℓ is a field location and $h''_{\mathbf{o}}$ is the uncaptured `free` handle with $u \in \mathit{hRSanc}_H(h''_{\mathbf{o}})$ and $\mathit{obsbd}(h,h''_{\mathbf{o}})$. Assignment to a field requires the top-level execution of a mutator by $h''_{\mathbf{o}}$'s source $\mathbf{r}$ (shallow state encapsulation). The observer invocations to which h is local must be below that. Mutator execution in $\mathbf{r}$ presupposes a sequence of mutator calls to $\mathbf{r}$ along ownership paths starting with a `free` path π_0 and followed by a sequence of `rep` paths to $\mathbf{r}$ (Theorem 8). The observer invocations with h must be below all these mutator invocations. In particular, the initial `free` edge $\hbar_0$ of π_0 that initiated the mutator calls is local to an invocation not below h. That is, $\mathit{obsbd}(h,\hbar_0)$. The corresponding membership $u \in \mathit{hRSanc}_{H'}(\hbar_0)$ with $H' = \{h''_{\mathbf{o}}\} \cup H \subseteq \mathit{fgr}_{om''}(\mathfrak{s}'')^{\circledast}$ holds since the capturing of $h''_{\mathbf{o}}$ in $\mathbf{r}$ connects $u \in \mathit{hRSanc}_H(h''_{\mathbf{o}})$ with the ownership paths leading to $\mathbf{r}$ This shows that also the final storage substep preserves the invariants.

$\boxed{\{\text{ret}\}}$ One aspect of `return` steps is to decreases the multiplicity of all handles at the locations of the finished invocation's environment. As shown for the case of $\{\text{if}_t\}/\{\text{if}_f\}$-steps, the intermediate removal step $\mathfrak{g}' = \mathfrak{g}_N \ominus \mathfrak{s}(\text{im}(\eta^\star))$ preserve the invariants. The interesting aspect of `return` steps is the transfer $\mathfrak{g} = \mathfrak{g}' \ominus \mathbf{r} \xrightarrow{\mu_{\mathbf{o}}} \mathbf{o} \ominus \mathbf{s} \xrightarrow{\mu_{\mathbf{r}}} \mathbf{r} \oplus \mathbf{s} \xrightarrow{\mu_{\mathbf{r}}\circ\mu_{\mathbf{o}}} \mathbf{o}$, of the result value, i.e., the substitution of the imported edge $h'_{\mathbf{o}} = \mathbf{s} \xrightarrow{\mu_{\mathbf{r}}\circ\mu_{\mathbf{o}}} \mathbf{o}$. for the exported edge $h_{\mathbf{o}} = \mathbf{r} \xrightarrow{\mu_{\mathbf{o}}} \mathbf{o}$ and the call-link $h_{\mathbf{r}} = \mathbf{s} \xrightarrow{\mu_{\mathbf{r}}} \mathbf{r}$.

First. There are no really new $\rightleftharpoons$-paths (Lemma 17). And the only really new ownership paths π are $\mathbf{s}$'s initially new paths π of a `free` mode $\mu_{\mathbf{r}}\circ\hat{\mu}$ with a counterpart $\mathfrak{expo}(\pi)$ of `free` mode $\hat{\mu}$ (Lemma 8). The new ownership paths π that are not really new are initially new and have a precursor $h_{\mathbf{r}} \bullet \mathfrak{expo}(\pi)$.

in $\mathfrak{g}'^{\circledast}$		in $\mathfrak{g}^{\circledast}$
$\neg rmut(u)$	$\Rightarrow$	$\neg rmut(u)$
$u \in RFree(h)$	$\Rightarrow$	$u \in RFree(\hat{\sigma}(h)) \;\vee\; \neg rmut(u)$
$u \in RSub_H(o)$	$\Rightarrow$	$u \in RSub_{\hat{\sigma}(H)}(o) \;\vee\; \neg rmut(u)$ $\vee\; \exists v, H' \subseteq \hat{\sigma}(H)\bullet\; \neg rmut(v) \wedge u \in RSub_{H'}(v)$

Figure 6.15: Change of relationships in return steps

This has the following implications summarized in figure 6.15: Old cases of immutability are preserved. Old cases of $u \in RFree(h)$ with $h \notin \{h_{\mathbf{o}}, h_{\mathbf{r}}\}$ are preserved unless the ownership reservation connection through h to u is severed by the removal of $h_{\mathbf{r}}$ and $h_{\mathbf{o}}$. And old cases of $u \in RFree(h)$ with $h = h_{\mathbf{o}}$ or $h_{\mathbf{r}}$ are succeeded by $u \in RFree(h'_{\mathbf{o}})$ unless the `free` ownership reservation on u through the handle is lost. Since there are no new ownership reservations except for $h'_{\mathbf{o}}$-based ones, this loss would mean that u is immutable now. In short, $u \in RFree(h)$ in $\mathfrak{g}'^{\circledast}$ entails $\neg rmut(u)$ or $u \in RFree(\hat{\sigma}(h))$ in $\mathfrak{g}^{\circledast}$. Old cases of $u \in RSub_H(o)$ may be preserved (if it contains no $h_{\mathbf{o}} \in H$), or the ownership reservation connection from o to u was severed, or a `free` reservation in the connection with `free` path $\pi = h_{\mathbf{o}} \bullet \tilde{\pi} = \mathfrak{expo}(\pi')$ or $h_{\mathbf{r}} \bullet h_{\mathbf{o}} \bullet \tilde{\pi} = h_{\mathbf{r}} \bullet \mathfrak{expo}(\pi')$ was shortened to a `free` reservation with initially new $\pi' = h'_{\mathbf{o}} \bullet \tilde{\pi}$. In the latter case, $u \in RSub_{\hat{\sigma}(H)}(o)$ now. In the second case, an object v is not reserved any more that was on the way from o to u, i.e., with $v \in RSub_{H''}(o)$ and $u = v$ or $u \in RSub_{H'}(v)$ in $\mathfrak{g}'^{\circledast}$ for some for $H', H'' \subseteq H$. The loss of reservation makes v immutable, perhaps also u.

Second. Let σ be the substitution $[h_{\mathbf{r}} \bullet h_{\mathbf{o}}/h'_{\mathbf{o}}, h_{\mathbf{o}}^{-1} \bullet h_{\mathbf{r}}^{-1}/h'^{-1}_{\mathbf{o}}]$ in case of $\mu_{\mathbf{r}} \circ \mu_{\mathbf{o}} =$ `co<>`, and $[h_{\mathbf{r}} \bullet h_{\mathbf{o}}/h'_{\mathbf{o}}]$ otherwise. By Lemmas 17 and 8, any reservation $\langle \pi, \varphi \rangle$ in $\mathfrak{g}^{\circledast}$ with $\pi = h_1 \bullet \tilde{\pi} \in PAP(o, \mu, q.\vec{\gamma})$ and $\varphi = q \overset{\vec{\gamma}}{\dashrightarrow}_{/\rightleftharpoons} \omega$ **via** Π implies a reservation $\langle \hat{\pi}, \hat{\varphi} \rangle$ in $\mathfrak{g}'^{\circledast}$ with $\hat{\pi} = h'_1 \bullet \tilde{\pi}' \in PAP(o', \mu', q.\vec{\gamma})$ and $\hat{\varphi} = q \overset{\vec{\gamma}}{\dashrightarrow}_{/\rightleftharpoons} \omega$ **via** $\sigma(\Pi)$ where normally $\hat{\pi} = \sigma(\pi)$ with source $o' = o$ and initial edge $h'_1 = h_1[h_{\mathbf{r}}/h'_{\mathbf{o}}]$, except in case of `free` initially new π without precursor, where $\hat{\pi} = \mathfrak{expo}(\pi)$ with source $o' = \mathbf{r}$ and initial edge $h'_1 = h_{\mathbf{o}}$. To the objects u in this reservation, the induction hypothesis applies. It covers all objects in $\langle \pi, \varphi \rangle$ in $\mathfrak{g}^{\circledast}$ since substitution σ never removes any objects from a path (but at most adds object $\mathbf{r}$): path-base Π contains no more objects than $\sigma(\Pi)$, π with $\hat{\pi} = \sigma(\pi)$ contains no more objects than $\hat{\pi}$, and initially new paths $\pi = h'_{\mathbf{o}} \bullet \tilde{\pi}$ contain non-initially no more objects than their precursor $\hat{\pi} = \mathfrak{expo}(\pi) = h_{\mathbf{o}} \bullet \sigma(\tilde{\pi})$ contained non-initially. The induction hypothesis allows three possibilities for u:

- u was immutable. As shown above, this is preserved.
- $\hat{\pi}$ is `free` and $u \in RFree(h'_1)$ in $\mathfrak{g}'^{\circledast}$. Then in $\mathfrak{g}^{\circledast}$, as shown above, u is immutable or $u \in RFree(h''_1)$ with $h''_1 = \hat{\sigma}(h'_1))$. If π is not initially new, then $h'_1 = h_1 \notin \{h_{\mathbf{r}}, h_{\mathbf{o}}\}$, so that in $\mathfrak{g}^{\circledast}$ we have $u \in RFree(h''_1) = RFree(h'_1) = RFree(h_1)$, as desired. And if $\pi = h'_{\mathbf{o}} \bullet \tilde{\pi}$ is initially new—with equivalent precursor $\hat{\pi} = h_{\mathbf{r}} \bullet \mathfrak{expo}(\pi)$ or with `free` counterpart $\hat{\pi} = \mathfrak{expo}(\pi) = h_{\mathbf{o}} \bullet \tilde{\pi}'$—in $\mathfrak{g}^{\circledast}$ we have $u \in RFree(h''_1) = RFree(h'_{\mathbf{o}}) = RFree(h_1)$, as desired.

- $\hat{\pi}$ is `rep` and there was a $\hat{q}$ that was o' or immutable and $u \in RSub_H(\hat{q})$ in $\mathfrak{g}'^{\circledast}$ with a h in $\hat{\pi}$ or $\sigma(\Pi)$ for every $\hbar \in H$ such that $h = \hbar \vee obsbd(h, \hbar)$. In $\mathfrak{g}^{\circledast}$, u is immutable or $u \in RSub_{\hat{\sigma}(H)}(\hat{q})$ or there is an immutable $\hat{q}'$ with $u \in RSub_{H'}(\hat{q}')$ and $H' \subseteq \hat{\sigma}(H)$. Since there are no really new coreceived ***rep*** paths π, $\hat{\pi}$ is equivalent to π, i.e., $o' = o$. That is, either u is immutable or there is an object $\hat{q}'$ that is o or immutable and $u \in RSub_{H'}(\hat{q}')$ with $H' \subseteq \hat{\sigma}(H)$. We check all the $\hbar' \in H'$ by looking at all the $\hbar \in H$. All of these are either in H' and thus exist in $\mathfrak{g}^{\circledast}$, or they are $h_{\mathbf{o}}$ or $h_{\mathbf{r}}$. They all have a corresponding h in $\hat{\pi}$ or $\sigma(\Pi)$, i.e., in some path $\hat{\pi}' \in \{\hat{\pi}\} \cup \sigma(\Pi)$. (In $\{\pi\} \cup \Pi$, there must be a corresponding path $\hat{\pi}''$ with $\hat{\pi}' = \sigma(\hat{\pi}'')$ or $\mathfrak{expo}(\hat{\pi}'')$.)
(i) If h occurs in $\hat{\pi}'$ not as part of a σ-replaced subsequence, this means it exists also in the $\mathfrak{g}^{\circledast}$-path $\hat{\pi}$. In the case of $h = \hbar$, this means that `free` $\hbar$ cannot be $h_{\mathbf{o}}$ or $h_{\mathbf{r}}$, since these would be reduced to zero multiplicity (Theorem 7). Hence $\hat{\sigma}(\hbar) = \hbar = h$ is in $\hat{\pi}$ as necessary for $\hat{\sigma}(\hbar) = \hbar' \in H'$. In the case of $obsbd(h, \hbar)$, $obsbd(h, \hat{\sigma}(\hbar))$ trivially follows if $\hbar \notin \{h_{\mathbf{r}}, h_{\mathbf{o}}\}$. This is what we need for $\hat{\sigma}(\hbar) = \hbar' \in H'$. Otherwise $\hat{\sigma}(\hbar) = \hbar' = h'_{\mathbf{o}} \in H'$ is at the new top-most call-level in $\mathfrak{g}^{\circledast}$ and unchanged h cannot be above it, so that necessarily $obsbd(h, h'_{\mathbf{o}})$, i.e., $obsbd(h, \hbar')$.
(ii) h can neither be $h_{\mathbf{r}}^{-1}$ nor $h_{\mathbf{o}}^{-1}$, since h is the `free` $\hbar$ or a non-co-handle with $obsbd(h, \hbar)$. If $h = h_{\mathbf{r}}$ or $h_{\mathbf{o}}$ in a σ-replaced subsequence $h_{\mathbf{r}} \bullet h_{\mathbf{o}}$ in $\hat{\pi}'$ then $\hat{\pi}''$ contains $h'_{\mathbf{o}}$ instead. In the case with $h = \hbar$, this means it contains $\hbar' = \hat{\sigma}(\hbar) = h'_{\mathbf{o}}$, as desired.
(iii) In the $obsbd(h, \hbar)$ case, if $h = h_{\mathbf{r}}$, then the invocation containing $h = h_{\mathbf{r}}$, i.e., the invocation to which the computation returns, is an observer. Since $h'_{\mathbf{o}}$ is contained in the same invocation as h, and since $\hat{\sigma}(\hbar)$ is $\hbar$ or is $h'_{\mathbf{o}}$, we have the necessary $obsbd(h'_{\mathbf{o}}, \hat{\sigma}(\hbar))$.
(iv) In the $obsbd(h, \hbar)$ case, if $h = h_{\mathbf{o}}$, then $obsbd(h, \hbar)$ for $h = h_{\mathbf{o}}$ at the terminated call-level $n + 1$ means two things: If the execution step returns to an observer at call-level n, it means the necessary $obsbd(h'_{\mathbf{o}}, \hat{\sigma}(\hbar))$ since either unchanged $\hat{\sigma}(\hbar) = \hbar$ is still at the same call-level or $\hat{\sigma}(\hbar) = h'_{\mathbf{o}}$ is at $h'_{\mathbf{o}}$'s call-level n. And if a mutator is executing at call-level n, it means that $\hbar$ cannot be at a level below $h = h_{\mathbf{o}}$'s level $n + 1$. But if it is local to the terminated call-level $n + 1$, the intermediate step $\mathfrak{g}'' = \mathfrak{g} \ominus \mathfrak{s}(\mathrm{im}(\eta^{\star}))$ destroys `free` $\hbar$ since its multiplicity was one by the reserved ownership assumption. This violates the assumption that $\hbar \in H$ exists still in $\mathfrak{g}^{\circledast}$ with the exception only of $\hbar = h_{\mathbf{o}}$ and $h_{\mathbf{r}}$.

$\boxed{\{\text{call}\}}$ As always, we decompose a {call}-step into substeps that convert the handle arguments' mode to the import $\mu_{\mathbf{r}} \circ \mu_{\mathbf{o}}$ of the receiver's parameter modes $\mu_{\mathbf{o}}$—which is safe as was shown for the case of {upd} above,—that insert the `this` handle—which obviously preserves the invariants—and that supply one parameter after another to the receiver. W.l.o.g. we focus on the substep $\mathfrak{g}'' = \mathfrak{g}' \ominus \mathbf{s} \xrightarrow{\mu_{\mathbf{r}} \circ \mu_{\mathbf{o}}} \mathbf{o} \oplus \mathbf{r} \xrightarrow{\mu_{\mathbf{o}}} \mathbf{o}$ of supplying one mode-adapted argument. This is trivial if the received handle $h_{\mathbf{o}} = \mathbf{r} \xrightarrow{\mu_{\mathbf{o}}} \mathbf{o}$ is a nil-handle or if it is not new but existed already in $\mathfrak{g}'^{\star}$. We are concerned

in $\mathfrak{g}'^{\circledast}$		in $\mathfrak{g}''^{\circledast}$
$\neg rmut(u)$	$\Rightarrow$	$\neg rmut(u)$
$u \in RFree(h) \wedge h \neq h'_{\mathbf{o}}$	$\Rightarrow$	$u \in RFree(h)$
$u \in RFree(h'_{\mathbf{o}})$	$\Rightarrow$	$u \in RFree(h_{\mathbf{r}}) \vee u \in RFree(h_{\mathbf{o}})$
$u \in RSub_H(o)$	$\Rightarrow$	$u \in RSub_{H'}(o)$
		where $H' \in \{H, H[h_{\mathbf{r}}/h'_{\mathbf{o}}], H[h_{\mathbf{o}}/h'_{\mathbf{o}}], H[h_{\mathbf{r}}, h_{\mathbf{o}}/h'_{\mathbf{o}}]\}$

Figure 6.16: Change of relationships in supply steps

only with new handles $h_{\mathbf{o}}$ and their inverses $h_{\mathbf{o}}^{-1}$ (if they are co), if they are edges in $\mathfrak{g}''^{\star}$ but not in $\mathfrak{g}'^{\star}$.

First, all μ-reservations $\langle\pi, \varphi\rangle$ of o on ω in $\mathfrak{g}''^{\circledast}$ have a precursor $\langle\pi', \varphi'\rangle$ of the same mode in $\mathfrak{g}'^{\circledast}$—with the exception that o's reservations with free initially new path π have a counterpart $\langle\mathfrak{sent}(\pi), \varphi'\rangle$ that reserved the free ownership of $\mathbf{r}$ on ω in $\mathfrak{g}'^{\circledast}$: If φ is new with a qbridge as precursor that contains quadruples, then π' is its last ownership path and φ' its last $\rightleftharpoons$-path. If there are no quadruples but π has a counterpart with a bridge with a triple series then π' is its last ownership path and φ' is the closure of its last $\rightleftharpoons$-path under φ's precursor $\rightleftharpoons$-path. Otherwise, $\pi' = \pi$ and $\varphi' = \varphi$.

Consequently, all cases of $\neg rmut(u)$ are preserved.

On the other hand, no old $\rightleftharpoons$-path φ gets really lost—only its path-base might change—and no ownership path π gets really lost —but always has an equivalent successor— with the sole exception that the free paths starting with $h'_{\mathbf{o}}$ are the witness $\mathfrak{sent}(\pi)$ of a free initially new path π that succeeds them: If the removed handle $h'_{\mathbf{o}}$ or its inverse occurred in an ownership path $\hat{\pi} = \pi'$ or in an ownership path or association path $\hat{\pi} \in \Pi'$ of old reservation $\langle\pi', \varphi'\rangle$ then it can be expanded with substitution $\sigma = [h_{\mathbf{r}} \bullet h_{\mathbf{o}}/h'_{\mathbf{o}}, h_{\mathbf{o}}^{-1} \bullet h_{\mathbf{r}}^{-1}/h'^{-1}_{\mathbf{o}}]$. The corresponding path $\sigma(\hat{\pi})$ is equivalent to $\hat{\pi}$ except if it is a free path starting with $h'_{\mathbf{o}}$. First, $h'_{\mathbf{o}}$ or $h'^{-1}_{\mathbf{o}}$ may be in ownership or association path $\hat{\pi}$ since they are co-handles, i.e., $\mu_{\mathbf{r}} \circ \mu_{\mathbf{o}} = \texttt{co<>}$. Then the typeability of redex $\hat{e}$ ensures that $h_{\mathbf{r}}$ and $h_{\mathbf{o}}$ are co as well. In this case, the expansion always works. Second, $h'_{\mathbf{o}}$ can be in ownership or association path $\hat{\pi}$ since it is an association handle of mode β<> or since it is extended to a β-path $\hat{\pi}' = h'_{\mathbf{o}} \bullet \hat{\pi}''$ that is a subsequence of $\hat{\pi}$: $\mu_{\mathbf{r}} \circ \mu_{\mathbf{o}}(\vec{\alpha}) = \beta$. Then typeability ensures a decomposition $\vec{\alpha} = \vec{\alpha}_1.\vec{\alpha}_2$ such that $\mu_{\mathbf{o}}(\vec{\alpha}_1) = \gamma$ and $\mu_{\mathbf{r}}$ is a rep or free mode with γ=$\tilde{\mu}$ such that $\tilde{\mu}(\vec{\alpha}_2) = \mu_{\mathbf{r}}(\gamma.\vec{\alpha}_2) = \beta$. In this case, $h_{\mathbf{r}} \bullet h_{\mathbf{o}} \bullet \sigma(\hat{\pi}'') = \sigma(\hat{\pi}')$ is a β-path that can substitute $\hat{\pi}'$ in $\hat{\pi}$. Third, $h'_{\mathbf{o}}$ can be the initial edge of *ownership* path $\hat{\pi} = h'_{\mathbf{o}} \bullet \hat{\pi}'$ of a shape $\mathbf{s} \xrightarrow{\mu_{\mathbf{r}} \circ \mu_{\mathbf{o}}} \mathbf{o} \xrightarrow{\texttt{co}}{}^{*} \bullet \overset{\vec{\alpha}}{\dashrightarrow} \bullet$, i.e., $\mu_{\mathbf{r}} \circ \mu_{\mathbf{o}}(\vec{\alpha}) = m \in \{\texttt{free}, \texttt{rep}\}$. Either $\mu_{\mathbf{o}}(\vec{\alpha}) = \texttt{free} = m$ (there is no rep in parameter mode $\mu_{\mathbf{o}}$), i.e., $\mu_{\mathbf{o}} = \epsilon$ and $\mu_{\mathbf{o}} = \texttt{free<...>}$. Then $\hat{\pi}$ is the free witness $\mathfrak{sent}(\pi)$ of a free initially new path π (and $\sigma(\hat{\pi})$ is no potential access path). Or $\mu_{\mathbf{o}} = \texttt{co<>}$. Then $h_{\mathbf{r}} \bullet h_{\mathbf{o}} = \mathbf{s} \xrightarrow{\mu_{\mathbf{r}}} \mathbf{r} \xrightarrow{\mu_{\mathbf{o}}} \mathbf{o}$ is equivalent to $h'_{\mathbf{o}} = \mathbf{s} \xrightarrow{\mu_{\mathbf{r}} \circ \mu_{\mathbf{o}}} \mathbf{o}$. $\hat{\pi}$'s substitute $\sigma(\hat{\pi})$ is $h_{\mathbf{r}} \bullet h_{\mathbf{o}} \bullet \sigma(\hat{\pi}')$. Or there is a decomposition $\vec{\alpha} = \vec{\alpha}_1.\vec{\alpha}_2$ such that $\mu_{\mathbf{o}}(\vec{\alpha}_1) = \gamma$ and $\mu_{\mathbf{r}}$ is a rep or free mode with γ=$\tilde{\mu}$ such that $\tilde{\mu}(\vec{\alpha}_2) = \mu_{\mathbf{r}}(\gamma.\vec{\alpha}_2) = m$. Then $h_{\mathbf{r}} \bullet h_{\mathbf{o}} \bullet \sigma(\hat{\pi}')$ is a substitute $\sigma(\hat{\pi})$ for $\hat{\pi}$.

Consequently, all old rep reservations $\langle\pi, \varphi\rangle$ with φ **via** Π have an equivalent successor $\langle\sigma(\pi), \varphi\rangle$ with φ **via** $\sigma(\Pi)$, so that old cases of $u \in \mathit{RSub}_{\emptyset}(o)$ are preserved (cf. fig. 6.16). And all free reservations $\langle\pi, \varphi\rangle$ with φ **via** Π either have an equivalent successor $\langle\sigma(\pi), \varphi\rangle$ with φ **via** $\sigma(\Pi)$ if $\mu_{\mathbf{o}} =$ co<>, or $\pi = \mathfrak{sent}(\pi')$ and the reservation of $\mathbf{s}$ switches to a free reservation $\langle\pi', \varphi\rangle$ with φ **via** $\sigma(\Pi)$ of $\mathbf{r}$ on the same object if $\mu_{\mathbf{o}} =$ free<...>. Hence all cases of $u \in \mathit{RFree}(h)$ with $h \neq h'_{\mathbf{o}}$ and of $u \in \mathit{RSub}_H(o)$ with $h'_{\mathbf{o}} \notin H$ are preserved. If $\mu_{\mathbf{o}} =$ co<>, all case of $u \in \mathit{RFree}(h'_{\mathbf{o}})$ and of $u \in \mathit{RSub}_H(o)$ with $h'_{\mathbf{o}} \in H$ become cases of $u \in \mathit{RFree}(h_{\mathbf{r}})$ and of $u \in \mathit{RSub}_{H'}(o)$ with $h_{\mathbf{r}} \in H' = H \setminus \{h'_{\mathbf{o}}\} \cup \{h_{\mathbf{r}}\}$ because they were based on a free path $\hat{\pi} = h'_{\mathbf{o}} \bullet \hat{\pi}'$ that has the equivalent successor $\sigma(\hat{\pi}) = h_{\mathbf{r}} \bullet h_{\mathbf{o}} \bullet \sigma(\hat{\pi}')$. If $\mu_{\mathbf{o}} \neq$ co<>, all case of $u \in \mathit{RFree}(h'_{\mathbf{o}})$ become cases of $u \in \mathit{RFree}(h_{\mathbf{o}})$ because they are based on a free path $\hat{\pi} = \mathfrak{sent}(\pi')$ with free successor π'. In order to say what becomes of $u \in \mathit{RSub}_H(o)$ with $h'_{\mathbf{o}} \in H$ if $\mu_{\mathbf{o}} \neq$ co<>, we have to consider what happens with the correlations to rep and association roles on free paths $\pi = \mathfrak{sent}(\pi')$ where π' is free and of mode $\mu_{\mathbf{o}}$: If π's mode $\mu_{\mathbf{r}} \circ \mu_{\mathbf{o}}$ contained such correlations, i.e., if $\mu_{\mathbf{r}} \circ \mu_{\mathbf{o}}(\vec{\gamma}_1.\vec{\gamma}_2) \in \{\mathtt{rep}\} \cup \mathbb{A}$ for some $\vec{\gamma}_1.\vec{\gamma}_2$, then $\mu_{\mathbf{o}}$ contained a correlation to some association role γ, i.e., $\mu_{\mathbf{o}}(\beta.\vec{\alpha}_1) = \gamma$, and $\mu_{\mathbf{r}}$ contains a correlation $\gamma{=}\hat{\mu}$ from γ to a mode $\hat{\mu}$ that is α or from which α can be extracted, i.e., with $\hat{\mu}(\vec{\gamma}_2) = \mu_{\mathbf{r}}(\gamma.\vec{\gamma}_2) = \alpha$. That is, also $h_{\mathbf{r}}$ and π's successor π' have correlations to a rep or association role. Since $\mu_{\mathbf{o}}$ contains the association mode γ<>, i.e., $\mu_{\mathbf{o}}(\beta.\vec{\alpha}_1) = \gamma$, and the corresponding $\mu_{\mathbf{r}} \circ \mu_{\mathbf{o}}(\beta.\vec{\alpha}_1)$ is not read, the typeability ensures that $\mu_{\mathbf{r}}$ is rep or free. Hence any free reservation starting with $h'_{\mathbf{o}} \in H$ can be expanded to a rep $h_{\mathbf{r}}$ or a free $h_{\mathbf{r}}$ and a free reservation starting with $h_{\mathbf{o}}$. Consequently, $u \in \mathit{RSub}_H(o)$ with $h'_{\mathbf{o}} \in H$ is succeeded by $u \in \mathit{RSub}_{H'}(o)$ with $H' = H \setminus \{h'_{\mathbf{o}}\} \cup \{h_{\mathbf{o}}\}$ if $h_{\mathbf{r}}$ is rep and $H' = H \setminus \{h'_{\mathbf{o}}\} \cup \{h_{\mathbf{r}}, h_{\mathbf{o}}\}$ if $h_{\mathbf{r}}$ is free.

Second. Consider reservations $\langle\pi, \varphi\rangle$ in $\mathfrak{g}''^{\circledast}$ where π is unchanged or internally-only new and φ **via** Π is unchanged. In the former case, π is its own precursor π'. In the latter case, $\mu_{\mathbf{o}} = \mu_{\mathbf{r}} =$ co<> and there is a precursor $\pi' = \pi[h_{\mathbf{r}}^{-1} \bullet h'_{\mathbf{o}}/h_{\mathbf{o}}, h'^{-1}_{\mathbf{o}} \bullet h_{\mathbf{r}}/h_{\mathbf{o}}^{-1}]$ with which π shares the initial edge h_1. Observe that the initial edge of π and π' are the same. The only difference in the set of intermediate objects is that π' contains $\mathbf{s}$ which π might not contain. Hence the induction hypothesis covers all objects u in $\langle\pi, \varphi\rangle$ as objects in the reservation $\langle\pi', \varphi\rangle$ in $\mathfrak{g}'^{\circledast}$:

The cases of $\neg\mathit{rmut}(u)$ and of $\mathit{RFree}(h_1)$ are preserved, as shown above. Notice that free h_1 and $\hbar$ cannot be $h'_{\mathbf{o}}$: If π is internally-only new, $h'_{\mathbf{o}}$ is co. And an unchanged path $\pi' = \pi$ or $\hat{\pi} \in \Pi$ cannot contain a $h'_{\mathbf{o}}$ that is free since the multiplicity of a free $h'_{\mathbf{o}}$ is by Unique Head (Theorem 7) reduced to zero in $\mathfrak{g}''^{\circledast}$. Finally, there could be a $\hat{q}$ that is o or is immutable and $u \in \mathit{RSub}_H(\hat{q})$ in $\mathfrak{g}'^{\circledast}$ with a h in $\hat{\pi}$ or $\hat{\Pi}$ for every $\hbar \in H$ such that $h = \hbar$ or $\mathit{obsbd}(h, \hbar)$. As shown above, we have $u \in \mathit{RSub}_{H'}(\hat{q})$ in $\mathfrak{g}''^{\circledast}$ for some $\hat{q}$ that is still o or still immutable. H' is H with any $h'_{\mathbf{o}}$ in it expanded to $h_{\mathbf{r}}$ and/or $h_{\mathbf{o}}$ (depending on which of them is free). It remains to check all the $\hbar \in H'$: If $h'_{\mathbf{o}}$ was in H then $\hbar = h_{\mathbf{o}}$ and $h_{\mathbf{r}}$ in H' are covered: The case with $h = h'_{\mathbf{o}}$ in π or Π cannot apply since a free $h'_{\mathbf{o}}$ is reduced to zero multiplicity in $\mathfrak{g}''^{\circledast}$ by Unique Head (Theorem 7). And the case of $\mathit{obsbd}(h, h'_{\mathbf{o}})$ means that also $\mathit{obsbd}(h, h_{\mathbf{r}})$

(for $\hbar = h_{\mathbf{r}} \in H'$) since $h'_{\mathbf{o}}$ and $h_{\mathbf{r}}$ are at the same call-level, and $obsbd(h, h_{\mathbf{o}})$ (for $\hbar = h'_{\mathbf{o}} \in H'$) since $h_{\mathbf{o}}$ is one call-level above $h'_{\mathbf{o}}$. Other cases of $\hbar \in H'$ if $h'_{\mathbf{o}} \in H$, and all cases of $\hbar \in H'$ if $h'_{\mathbf{o}} \in H$ presuppose $\hbar \in H$. For every $\hbar \in H$ the necessary h in π or Π is guaranteed with $h = \hbar$ or $obsbd(h, \hbar)$.

Third. Consider reservations $\langle \pi, \varphi \rangle$ in $\mathfrak{g}''^{\circledast}$ where π is *initially new* and φ **via** Π is unchanged. Then $\pi = h_{\mathbf{o}} \bullet \pi_1 \bullet \pi_2$, and in $\mathfrak{g}'^{\circledast}$ there was a witness $\mathfrak{wit}(\pi) = h'_{\mathbf{o}} \bullet \pi_1 \bullet \pi'_2 \in PAP_{\mathfrak{g}'^{\circledast}}(\mathbf{s}, \mu_{\mathbf{r}} \circ \mu, q_0.\vec{\alpha}_0)$ and a $\vec{\alpha}_0$-bridge **via** Π from q_0 to π's target $q.\vec{\gamma}$. The typeability of redex $\hat{e}$ ensures that π is not `rep` but `free`. Because of the nesting constraints on valid modes, this means that $\mu_{\mathbf{o}}$ is a `free` mode, i.e., $h_{\mathbf{o}}$ is `free`, and that $h_{\mathbf{o}}$ can only be extended by `co`-edges. Hence π_1 must a sequence of `co`-edges, and π_2 must be empty and the bridge be trivial since otherwise π_2 would start with (`free`) $h_{\mathbf{o}}$. Consequently, $\mathfrak{wit}(\pi)$ has the same target as π and the same edges except for the initial edge $h'_{\mathbf{o}}$ in place of $h_{\mathbf{o}}$. Since $\mathfrak{wit}(\pi)$ must be `free` like π is, all non-initial objects u in $\langle \pi, \varphi \rangle$ are covered as non-initial objects in $\langle \mathfrak{wit}(\pi), \varphi \rangle$. For this `free` reservation, the induction hypothesis allows the following possibilities:

- u was immutable in $\mathfrak{g}'^{\circledast}$. Hence it still is so in $\mathfrak{g}''^{\circledast}$.
- $u \in RFree(h'_{\mathbf{o}})$ in $\mathfrak{g}'^{\circledast}$ since $h'_{\mathbf{o}}$ is $\mathfrak{wit}(\pi)$'s initial edge. In $\mathfrak{g}''^{\circledast}$, as shown above, we then have $u \in RFree(h_{\mathbf{o}})$ since $h_{\mathbf{o}}$ is `free`, not `co`. This is just right since $h_{\mathbf{o}}$ is π's initial edge.

Fourth. Consider reservations $\langle \pi, \varphi \rangle$ in $\mathfrak{g}''^{\circledast}$ where π is *internally really new* and φ **via** Π is unchanged. Then $\pi = h_1 \bullet \tilde{\pi} = \pi_1 \bullet \pi_2$, and in $\mathfrak{g}'^{\circledast}$ there was a witness $\mathfrak{wit}(\pi) = h_1 \bullet \tilde{\pi}' = \pi_1 \bullet \pi'_2 \in PAP_{\mathfrak{g}'^{\circledast}}(o, \texttt{free<...>}, q_0.\vec{\alpha}_0)$ with $\pi'_2 = q_0 \xrightarrow{\vec{\alpha}_0} q_0.\vec{\alpha}_0$ and a $\vec{\alpha}_0$-bridge from q_0 to ω **via** Π' such that $\{\pi_2\} \cup \Pi_\Lambda \cup \Pi_{\circledast} \equiv \Pi \cup \Pi_{\mathbf{o}}$. From these, the following reservations $\langle \hat{\pi}, \hat{\varphi} \rangle$ can be constructed in $\mathfrak{g}'^{\circledast}$: If there are triples in the bridge, then witness $\mathfrak{wit}(\pi)$ and the bridge's initial $\rightleftharpoons$-path $\varphi_0 = q_0 \overset{\vec{\alpha}_0}{\dashrightarrow}_{/\rightleftharpoons} \omega_1$ **via** Π_0 constituted a reservation $\langle \mathfrak{wit}(\pi), \varphi_0 \rangle$, and each triple in the bridge means two reservations $\langle \pi'_i, \omega_i \overset{\epsilon}{\dashrightarrow}_{/\rightleftharpoons} \omega_i \rangle$ and $\langle \pi_i, \varphi_i \rangle$. The last $\rightleftharpoons$-path $\varphi_n = q_n \overset{\vec{\alpha}_n}{\dashrightarrow}_{/\rightleftharpoons} \omega_n = q.\vec{\gamma}$ **via** Π_n combined with φ to $\tilde{\varphi}_n = q_n \overset{\vec{\alpha}_n}{\dashrightarrow}_{/\rightleftharpoons} \omega$ **via** $\Pi_n \cup \Pi$ (Lemma 12). It combined to a reservation $\langle \tilde{\pi}_n, \tilde{\varphi}_n \rangle$ with the bridge's last ownership path $\tilde{\pi}_n = \pi_n = \mathbf{s} \xrightarrow{\hat{\mu}_n} q_n.\vec{\alpha}_n$. If there are no triples in the bridge, then $\Pi' = \Pi_0$. φ_0 and φ combined to $\tilde{\varphi}_n = q_0 \overset{\vec{\alpha}_0}{\dashrightarrow}_{/\rightleftharpoons} \omega$ **via** $\Pi_0 \cup \Pi = \Pi_n \cup \Pi = \Pi' \cup \Pi$. Hence in $\mathfrak{g}'^{\circledast}$ there was the μ-reservation $\langle \mathfrak{wit}(\pi), \tilde{\varphi}_n \rangle$ of o on ω.

If there are any triples in the bridge, the reserved ownership assumption in $\mathfrak{g}'^{\circledast}$ ensures that $\mathfrak{wit}(\pi)$'s source o is their source $\mathbf{s}$ and that all ownership paths have the mode of $\mathfrak{wit}(\pi)$. Hence if $\langle \pi, \varphi \rangle$ in $\mathfrak{g}''^{\circledast}$ is a `free` reservation then all corresponding reservations $\langle \hat{\pi}, \hat{\varphi} \rangle$ were also `free` reservations. In each reservation $\langle \hat{\pi}, \hat{\varphi} \rangle$ with $\hat{\pi} = h'_1 \bullet \hat{\pi}'$, the sources and targets u of all edges h in $\hat{\pi}'$ or $\hat{\Pi}$ are covered by the induction hypothesis. In case of $\langle \hat{\pi}, \hat{\varphi} \rangle = \langle \mathfrak{wit}(\pi), \varphi_0 \rangle$ or $\langle \mathfrak{wit}(\pi), \tilde{\varphi}_n \rangle$, the missing edge h'_1 is the initial edge h_1 of π, and thus needs no coverage. Otherwise it is the initial edge $h_{\mathbf{r}}$ or $h'_{\mathbf{o}}$ of an ownership path in the bridge, which is skipped in the construction of the new path π and thus irrelevant for $\langle \pi, \varphi \rangle$. (Of course $h_{\mathbf{r}}$ and $h'_{\mathbf{o}}$ will be contained in

π if they are contained elsewhere in the bridge: in one of the $\rightleftharpoons$-paths or non-initially in one of the ownership paths—but then it is covered by the induction hypothesis.) For the sources and targets u of all non-initial edges in each of the above reservations $\langle\hat{\pi},\hat{\varphi}\rangle$, i.e., for all objects in π which need to be covered, the induction hypothesis allows the following possibilities:

- u was immutable in $\mathfrak{g}'^{\circledast}$. Hence it still is so in $\mathfrak{g}''^{\circledast}$.
- $\langle\hat{\pi},\hat{\varphi}\rangle$ is a `free` reservation with initial edge h_1' and $u \in \mathit{RFree}(h_1')$. In the case of $\langle\hat{\pi},\hat{\varphi}\rangle = \langle\mathfrak{wit}(\pi),\varphi_0\rangle$ or $\langle\mathfrak{wit}(\pi),\tilde{\varphi}_n\rangle$, h_1' is π's unchanged initial edge h_1. If `free` h_1' were $h_\mathbf{o}'$, then its multiplicity of one (the reserved ownership assumption) would be reduced to zero in $\mathfrak{g}''^{\circledast}$, so that it could not be unchanged—a contradiction. But if $h_1' \neq h_\mathbf{o}'$, then $u \in \mathit{RFree}(h_1')$ is preserved, i.e., $u \in \mathit{RFree}(h_1)$.
  Other cases of $\langle\hat{\pi},\hat{\varphi}\rangle$ presuppose triples in the bridge contains. The nesting constraint on modes ensures that the `free` internally really new π is made of an initial `free` h_1 followed by `co`-edges. Hence the $h_\mathbf{o}$ in it must be `co`: $\mu_\mathbf{o} =$ `co<>`. But then there is one $\vec{\beta}$ such that every ownership path $\hat{\pi} = \pi_i'$ in $\langle\hat{\pi},\hat{\varphi}\rangle = \langle\pi_i', \omega_i \overset{\epsilon}{\dashrightarrow}\!\!/_{\rightleftharpoons}\, \omega_i\rangle$, or $\hat{\pi} = \pi_i$ in $\langle\hat{\pi},\hat{\varphi}\rangle = \langle\pi_i,\varphi_i\rangle$ has shape $\mathbf{s} \xrightarrow{\mu_\mathbf{r}\circ\mu_\mathbf{o}} \mathbf{o} \xrightarrow{\mathtt{co}}{}^{*} \bullet \overset{\vec{\beta}}{\dashrightarrow} \bullet$ or $\mathbf{s} \xrightarrow{\mu_\mathbf{r}} \mathbf{r} \xrightarrow{\mathtt{co}}{}^{*} \bullet \overset{\vec{\beta}}{\dashrightarrow} \bullet$. For the quadruple $\langle\mathfrak{wit}(\pi),\varphi_0,\pi_i',\omega_i \overset{\epsilon}{\dashrightarrow}\!\!/_{\rightleftharpoons}\, \omega_i\rangle$ of the first and second reservation, this means by the reserved ownership assumption that $\mathfrak{wit}(\pi)$'s unchanged initial edge h_1 must be $h_\mathbf{r}$ (since it cannot be $h_\mathbf{o}'$, as just shown before). Hence if $\langle\hat{\pi},\hat{\varphi}\rangle$ is a reservation with $\hat{\pi}$ of the case with shape $\mathbf{s} \xrightarrow{\mu_\mathbf{r}} \mathbf{r} \xrightarrow{\mathtt{co}}{}^{*} \bullet \overset{\vec{\beta}}{\dashrightarrow} \bullet$, then $u \in \mathit{RFree}(h_1')$ means $u \in \mathit{RFree}(h_\mathbf{r}) = \mathit{RFree}(h_1)$, and this is preserved since $h_1 \neq h_\mathbf{o}'$. And if $\langle\hat{\pi},\hat{\varphi}\rangle$ is a reservation with $\hat{\pi}$ of the case with shape $\mathbf{s} \xrightarrow{\mu_\mathbf{r}\circ\mu_\mathbf{o}} \mathbf{o} \xrightarrow{\mathtt{co}}{}^{*} \bullet \overset{\vec{\beta}}{\dashrightarrow} \bullet$, then $u \in \mathit{RFree}(h_1')$ means $u \in \mathit{RFree}(h_\mathbf{o}')$. As shown above, this is succeeded in $\mathfrak{g}''^{\circledast}$ by $u \in \mathit{RFree}(h_\mathbf{r}) = \mathit{RFree}(h_1)$, since the alternative $u \in \mathit{RFree}(h_\mathbf{o}')$ is impossible with an $h_\mathbf{o}$ of mode $\mu_\mathbf{o} =$ `co<>`.

The other case applies to `rep` reservations. If, in this case, $h_\mathbf{r}$ is a `free` handle then there is an h' in π such that $h' = h_\mathbf{r}$ or $\mathit{obsbd}(h', h_\mathbf{r})$: $\mathfrak{wit}(\pi) = \pi_1 \bullet \pi_2'$ passes through $\mathbf{r}$ since the prefix π_1 common with $\pi = \pi_1 \bullet \pi_2$ ends in $\mathbf{r}$ (from where it continues with $h_\mathbf{o}$ initially in π_2). Hence for object $\mathbf{r}$ in `rep` reservation $\langle\mathfrak{wit}(\pi), \mathbf{r}.\vec{\alpha}_0 \overset{\epsilon}{\dashrightarrow}\!\!/_{\rightleftharpoons}\, \mathbf{r}.\vec{\alpha}_0\rangle$, the induction hypothesis allows the following possibilities: The case of immutable $\mathbf{r}$ cannot apply since it is target of uncaptured `free` $h_\mathbf{r}$. There must be an object $\hat{q}'$ that is o or is immutable such that $\mathbf{r} \in \mathit{RSub}_H(\hat{q}')$. All ownership reservation sequences to $\mathbf{r}$ targeted by `free` $h_\mathbf{r}$ have to include a `free` reservation with initial $h_\mathbf{r}$ (the reserved ownership assumption). Hence $h_\mathbf{r}$ must be included in H. But then we are guaranteed an h' in the `rep` reservation, i.e., in $\mathfrak{wit}(\pi)$, such that $h' = h_\mathbf{r}$ or $\mathit{obsbd}(h', h_\mathbf{r})$. However, whether h' is `free` $h_\mathbf{r}$ or uncaptured ($\mathit{obsbd}(h, \hbar)$), it cannot be in the dummy edge-path π_2'. But then h' is in π_1 and thus in $\pi = \pi_1 \bullet \pi_2$.

- There was a $\hat{q}$ that was o or immutable and $u \in \mathit{RSub}_H(\hat{q})$ in $\mathfrak{g}'^{\circledast}$ with a h in $\hat{\pi}$ or $\hat{\Pi}$ for every $\hbar \in H$ such that $h = \hbar \vee \mathit{obsbd}(h, \hbar)$. As shown above, we have $u \in \mathit{RSub}_{H'}(\hat{q})$ in $\mathfrak{g}''^{\circledast}$ for some $\hat{q}$ that is still o or still immutable. H' is H with any $h'_{\mathbf{o}}$ in it expanded to $h_{\mathbf{r}}$ and/or $h_{\mathbf{o}}$ (depending on which of them is `free`). It remains to check all the $\hbar \in H'$: (1) $\hbar = h_{\mathbf{o}} \in H'$ is always in $\langle \pi, \varphi \rangle$ since any internally really new $\pi = \pi_1 \bullet \pi_2$ contains $h_{\mathbf{o}}$ as the first element of its π_2-segment. (2) For $\hbar = h_{\mathbf{r}} \in H'$, it was shown just before, there is an h' in π with $h' = h_{\mathbf{r}}$ or $obsbd(h', h_{\mathbf{r}})$. (3) The remaining $\hbar \in H'$ were also in H: For every $\hbar \in H$, there was the necessary h in $\hat{\pi}$ or $\hat{\Pi}$. This means that h was in witness $\mathfrak{wit}(\pi) = \pi_1 \bullet \pi'_2$, or in an ownership path π'_i or π_i, or in a $\rightleftharpoons$-path φ_i **via** Π_i or $\tilde{\varphi}_n$ **via** $\Pi_n \cup \Pi$. In short, h was in $\{\pi_1 \bullet \pi'_2\} \cup \Pi' \cup \Pi$, or formally, $\{h\} \Subset \{\pi_1, \pi'_2\} \cup \Pi$. Since we know about π's bridge that $\{\pi_2\} \cup \Pi_\Lambda \cup \Pi_\circledast \equiv \Pi' \cup \Pi_{\mathbf{o}}$, this means $\{h\} \Subset \Pi \cup \{\pi_1, \pi_2, \pi'_2\} \cup \Pi_\Lambda \cup \Pi_\circledast$. Whether h is `free` $\hbar$ or non-`co` and uncaptured ($obsbd(h, \hbar)$), it cannot be in the dummy edge-path π'_2 nor in dummy edge set $\Pi_\circledast$, nor can it be an inverse `co`-edge $h_{\mathbf{r}}^{-1}, h_{\mathbf{o}}'^{-1} \in \Pi_\Lambda$.

(i) If h was in π_1 or π_2 or Π, then it is in $\langle \pi, \varphi \rangle$. Hence we have for $\hbar \in H'$ the desired handle h with still $h = \hbar$ or $obsbd(h, \hbar)$, respectively.

(ii) If, in the $h = \hbar$ case, $h \in \Pi_\Lambda$ then it cannot be $h = \hbar = h'_{\mathbf{o}} \in \Pi_\Lambda$ since $h'_{\mathbf{o}}$ is not in H'. And if $h = \hbar = h_{\mathbf{r}} \in \Pi_\Lambda$ then, it was shown above, there is an h' in π with $h' = h_{\mathbf{r}}$ or $obsbd(h', h_{\mathbf{r}})$.

(iii) If, in the $obsbd(h, \hbar)$ case, $h = h'_{\mathbf{o}}$ or $h_{\mathbf{r}} \in \Pi_\Lambda$ and the invoked method is an *observer*, we switch from h to $h_{\mathbf{o}}$, which is always in $\pi = \pi_1 \bullet \pi_2$ as first edge of π_2. Notice that $h_{\mathbf{o}}$ was assumed to be new at the beginning of the {call}-case. Hence in $\mathfrak{g}''^{\circledast}$, $h_{\mathbf{o}}$ is local to the called method and neither local at other call-levels nor captured in a field. That is, $h_{\mathbf{o}}$ is just one observer higher than h. Hence $obsbd(h, \hbar)$ implies the necessary $obsbd(h_{\mathbf{o}}, \hbar)$ for $\hbar \in H'$.

(iv) If, in the $obsbd(h, \hbar)$ case, $h = h'_{\mathbf{o}}$ or $h_{\mathbf{r}} \in \Pi_\Lambda$ and the invoked method is a *mutator*, then the typeability of redex $\hat{e}$ requires that call-link $h_{\mathbf{r}}$ is `free` since the calling invocation containing h is an observer. It was shown above that then there is an h' in π with $h' = h_{\mathbf{r}}$ or $obsbd(h', h_{\mathbf{r}})$. In the former case, $h' = h_{\mathbf{r}}$ and $h \in \Pi_\Lambda$ are identical or at least at the same call-level. Hence $obsbd(h, \hbar)$ implies the $obsbd(h', \hbar)$ necessary for $\hbar \in H'$ with h' in π. In the latter case, $obsbd(h', h_{\mathbf{r}})$ means that $h_{\mathbf{r}}$ was above h' or only some observer calls below it. On the other hand, we had $obsbd(h, \hbar)$ for $h \in \Pi_\Lambda$, i.e., $obsbd(h_{\mathbf{r}}, \hbar)$ or $obsbd(h'_{\mathbf{o}}, \hbar)$. That is, $\hbar$ was above $h_{\mathbf{r}}$ or $h'_{\mathbf{o}}$ (which is the same) or only some observer calls below it. In combination, $\hbar$ was above h' or only some observer calls below it. Hence we have $obsbd(h', \hbar)$ necessary for $\hbar \in H'$ with h' in π.

Fifth. Consider reservations $\langle \pi, \varphi \rangle$ with a *new* $\rightleftharpoons$-path $\varphi = q \overset{\vec{\gamma}}{\dashrightarrow}_{/\rightleftharpoons} \omega$ **via** Π. The existence of reservation $\langle \pi, \varphi' \rangle$ with the *unchanged* trivial $\rightleftharpoons$-path $\varphi' = (q.\vec{\gamma}).\epsilon \overset{\epsilon}{\dashrightarrow}_{/\rightleftharpoons} q.\vec{\gamma}$ **via** $\emptyset$ ensures, as shown above, that all objects in π satisfy the invariants for ownership reservations. We still need to check the objects u in the path-base Π. For new φ, there was in $\mathfrak{g}'^{\circledast}$ a $\vec{\gamma}$-qbridge from q to ω **via** Π' such that $\Pi' \cup \Pi_{\mathbf{o}} \equiv \Pi \cup \Pi_\Lambda \cup \Pi_\circledast$ (Lemma 18). The objects in $\Pi \Subset \Pi' \cup \Pi_{\mathbf{o}}$ are the source or target of edges occurring also in Π', or of $h_{\mathbf{o}}$ and $h_{\mathbf{o}}^{-1} \in \Pi_{\mathbf{o}}$. Source and target of $h_{\mathbf{o}}$ and $h_{\mathbf{o}}^{-1}$ are covered as the targets of $h_{\mathbf{r}}$ and of $h'_{\mathbf{o}}$ or sources of $h_{\mathbf{r}}^{-1}$ and of $h_{\mathbf{o}}'^{-1}$, which are guaranteed to be contained in Π'. All handles in Π', and thus all objects in Π are contained in one of the following reservations:

Each quadruple in the qbridge means two reservations $\langle \pi_i', \varphi_i' \rangle$ and $\langle \pi_i, \varphi_i \rangle$. The qbridge's initial $\rightleftharpoons$-path $\varphi_0 = q \xrightarrow{\vec{\gamma}}\!\!/_{\rightleftharpoons} \omega_1$ **via** Π_0 is covered by the reservation $\langle \pi_0', \varphi_0 \rangle$ if π is unchanged or internally-only new and π_0' is its precursor π'. And if π is internally really new or initially new, then φ_0 combines with the last $\rightleftharpoons$-path φ_n **via** Π_n'' of π's bridge to a $\rightleftharpoons$-path $\varphi_0' = q \xrightarrow{\vec{\alpha}_n}\!\!/_{\rightleftharpoons} \omega$ **via** $\Pi_n'' \cup \Pi_0$. It combined to the reservation $\langle \pi_0', \varphi_0' \rangle$ with witness $\pi_0' = \mathfrak{wit}(\pi)$ or with the last ownership path $\pi_0' = \mathbf{s} \xrightarrow{\hat{\mu}_n''} q_n''.\vec{\alpha}_n''$ in π's bridge. Either directly or through π's bridge, the reserved ownership assumption guarantees that π_0' has the same mode μ' and source o' as, respectively, π' or $\mathfrak{wit}(\pi)$ (cf. the internally really new case above). Observe that the initial edges of each ownership path in the qbridge and in the bridge is $h_\mathbf{r}$ or $h_\mathbf{o}'$.

If μ' is `free` and the qbridge contains a quadruple or the bridge contains a triple, then this means for the unchanged initial edge h_1 of other π's by the reserved ownership assumption that it is $h_\mathbf{r}$ or $h_\mathbf{o}'$. Actually, it must be $h_1 = h_\mathbf{r}$ since unchanged h_1 cannot be the `free` $h_\mathbf{o}'$ that is decreased to zero multiplicity (cf. the internally really new π case further above). If μ' is `free` and $\mu_\mathbf{o} \neq$ `co<>` then there are no triples and no quadruples: In the non-`co` case, one ownership path $\hat{\pi}$ of the ownership path-pair in each triple or quadruple has shape $\mathbf{s} \xrightarrow{\mu_\mathbf{r} \circ \mu_\mathbf{o}} \mathbf{o} \xrightarrow{\mathtt{co}}{}^{*} \bullet \dashrightarrow^{\vec{\alpha}.\vec{\gamma}} \bullet$ with $\mu_\mathbf{o}(\vec{\alpha}) = \beta$. The nesting constraint on modes, on one hand, allows a `free` $\hat{\pi}$ only if it has shape $\mathbf{s} \xrightarrow{\mu_\mathbf{r} \circ \mu_\mathbf{o}} \mathbf{o} \xrightarrow{\mathtt{co}}{}^{*} \bullet \dashrightarrow^{\epsilon} \bullet$: $\vec{\alpha}$ is ϵ and $h_\mathbf{o}'$ has mode $\mu_\mathbf{o} = \beta$`<>`. On the other hand, it disallows correlations β=`free<...>`, $\mu_\mathbf{r} \circ \mu_\mathbf{o} = \mu_\mathbf{r} \circ \beta$`<>` cannot be `free`.

In each reservation $\langle \hat{\pi}, \hat{\varphi} \rangle$, the sources and targets of all edges except for $\hat{\pi}$'s initial h_1' are covered by the induction hypothesis. h_1' is either the initial edge of π_0', which has nothing to do with φ and was already covered above. Or it is the initial edge $h_\mathbf{r}$ or $h_\mathbf{o}'$ of an ownership path in the qbridge, which is skipped in the construction of the new $\rightleftharpoons$-path φ and thus irrelevant for $\langle \pi, \varphi \rangle$. For the sources and targets u of all non-initial edges in each of the above reservations $\langle \hat{\pi}, \hat{\varphi} \rangle$, i.e., for all objects in φ we need to cover, the induction hypothesis allows the following possibilities:

- u was immutable in $\mathfrak{g}'^{\circledast}$. Hence it still is so in $\mathfrak{g}''^{\circledast}$.
- $u \in \mathit{RFree}(h_1')$ in case of a `free` reservation. If there are no quadruples in φ's qbridge and no triples in π's bridge then $\hat{\pi} = \pi_0'$ is π' or $\mathfrak{wit}(\pi)$. In the initially new π case then $\mu_\mathbf{o}$ is `free` and $\mathfrak{wit}(\pi)$'s initial edge h_1' is $h_\mathbf{o}'$. Hence $u \in \mathit{RFree}(h_1')$ is succeeded by $u \in \mathit{RFree}(h_\mathbf{o})$, which is just right since $h_\mathbf{o}$ is π's initial edge h_1 (cf. the initially new π case further above). If π is not initially new then π''s initial edge h_1' coincides with π's initial edge h_1. Since $h_1' = h_1$ is unchanged, it cannot be `free` $h_\mathbf{o}'$ (cf. the internally really new π case further above). Hence $u \in \mathit{RFree}(h_1') = \mathit{RFree}(h_1)$ is preserved. If there are quadruples or triples then, as shown above, $\mu_\mathbf{o} =$ `co<>`, so that there are no initially new paths, and $\hat{\pi}$'s initial edge is $h_1' \in \{h_\mathbf{r}, h_\mathbf{o}'\}$, and $h_1 = h_\mathbf{r}$ for non-initially new π. If $h_1' = h_\mathbf{r}$, $u \in \mathit{RFree}(h_1') = \mathit{RFree}(h_\mathbf{r}) = \mathit{RFree}(h_1)$ is preserved, and if $h_1' = h_\mathbf{o}'$, we have new $u \in \mathit{RFree}(h_\mathbf{r}) = \mathit{RFree}(h_1)$ in $\mathfrak{g}''^{\circledast}$.

- $u \in RSub_H(\hat{q})$ in case of a `rep` reservation for some $\hat{q}$ that was o' or immutable and for every $\hbar \in H$ there was a h in $\hat{\pi}$ or $\hat{\Pi}$ such that $h = \hbar \vee obsbd(h, \hbar)$. In the `rep` case, π cannot be initially new since there are only `free` initially new ownership paths, and these have a `free` precursor $\mathfrak{sent}(\pi)$, so that also the mode of $\hat{\pi}$ is `free`. For the other π, $o' = o$. As shown above, we have $u \in RSub_{H'}(\hat{q})$ in $\mathfrak{g}''^{\circledast}$ for some $\hat{q}$ that is still $o' = o$ or still immutable. H' is H with any $h'_{\mathbf{o}}$ in it expanded to $h_{\mathbf{r}}$ and/or $h_{\mathbf{o}}$ (depending on which of them is `free`). It remains to check all the $\hbar \in H'$: If `free` $\hbar$ is $h_{\mathbf{r}}$ or $h_{\mathbf{o}}$, then π cannot be internally-only new since this would presuppose $\mu_{\mathbf{r}} = \mu_{\mathbf{o}} =$ `co<>`. And a internally really new $\pi = \pi_1 \bullet \pi_2$ always contains $h_{\mathbf{o}}$ as the first element of its π_2-segment for $\hbar = h_{\mathbf{o}}$, and always contains an h' with $h' = h_{\mathbf{r}}$ or $obsbd(h', h_{\mathbf{r}})$ for `free` $\hbar = h_{\mathbf{r}}$. For details see the internally really new reservation case further above. The remaining $\hbar \in H'$ were also in H: For every $\hbar \in H$, there was the necessary h in $\hat{\pi}$ or $\hat{\Pi}$. This means, h is in some path in Π'. Since $\Pi' \Subset \Pi \cup \Pi_\Lambda \cup \Pi_\circledast$, this means that h must be in Π or must be $h_{\mathbf{r}}$ or $h'_{\mathbf{o}} \in \Pi_\Lambda$. Whether h is `free` $\hbar$ or non-`co` and uncaptured $(obsbd(h, \hbar))$, it cannot be in the dummy edge-path π'_2 nor in dummy edge set $\Pi_\circledast$.
  If h was in Π, then it is in $\langle \pi, \varphi \rangle$. Hence we have for $\hbar \in H'$ the desired handle h with still $h = \hbar$ or $obsbd(h, \hbar)$, respectively. If $h = h_{\mathbf{r}}$ or $h'_{\mathbf{o}} \in \Pi_\Lambda$, then π cannot be internally-only new since this would presuppose $\mu_{\mathbf{r}} = \mu_{\mathbf{o}} = \mu_{\mathbf{r}} \circ \mu_{\mathbf{o}} =$ `co<>`. Otherwise the reasoning is like in the internally really new reservation case: In case of $h = \hbar$, h cannot be $h'_{\mathbf{o}}$, and if it is `free` $h_{\mathbf{r}}$ then there is an h' in π with $h' = h_{\mathbf{r}}$ or $obsbd(h', h_{\mathbf{r}})$. In case of $obsbd(h, \hbar)$, if a mutator is called, $h_{\mathbf{r}}$ must be `free`, which ensures an h' in π with $obsbd(h', \hbar)$. And if an observer is called then $h_{\mathbf{o}}$ is just one observer higher than h, so that we have $obsbd(h_{\mathbf{o}}, \hbar)$ for $\hbar \in H'$. ■

Chapter 7

Discussion

If you have built castles in the air,
your work need not be lost;
that is where they should be.
Now put the foundations under them.
Henry David Thoreau (1817-1862)

7.1 JaM: Some Observations

7.1.1 Programming with Modes

1. A PURELY STATIC TYPE SYSTEM EXTENSION. JaM is essentially a *purely static type system extension* of a Java-subset that imposes a new dimension of classification on old entities, namely the classification of Java's object reference values by modes and the classification of Java's methods into observers and mutators:

No new code composition constructs are introduced (like inheritance, class nesting, packages). No new values are introduced. The annotation of modes to handles in the formal runtime model of the previous chapters was only a formal trick for proving the desired properties; they have no impact on the computation and are invisible from outside of the program (handles cannot be exchanged with the execution environment). For the implementations of the JaM language, no representation of modes at runtime is needed. No new computational mechanism is introduced, like the delegation semantics associated with special references in other proposals (e.g., `parent` references for object inheritance [US87], `inner` references for dynamic mixins [Nic99], and `context` references for context dependent behavior [SPL98]). There is only the new *destructive read* operation. But this is no substantial addition, since a destructive read can clearly always be rewritten to a combination of a normal, non-destructive read and an assignment of `null` with the same net effect.

Actually, even the destructive read operation is not crucial for JaM. It could be replaced by a purely static deadness-analysis as part of type checking: Destructive read only serves as an easily enforceable way of ensuring that `free` handles are used as "linear values" [Wad90, Bak95]. This means that after assignment of a value to a `free` variable, this value—a `free` handle—is used at most once *as a* `free` *handle.* In other words, `free` variables are "dead after use" as the source of a `free` handle [Boy01]: Once a `free` variable's value was used as a *`free`* expression value (read access to the variable), the next use of that variable can only be as the left hand side of an assignment (write access). For example, the same `free` variable cannot be used both as `free` receiver and `free` argument, or as first and as second `free` argument, in the same operation call expression. A compile-time deadness check for read `free` variables would be as effective for reasoning about object ownership as enforcing that `free` variables are read only destructively. (Notice that the variable's deadness is exactly the property which allows the compiler to optimize away destructive read's resetting of the variable.) Indeed, intergrated deadness checking would be preferable since it avoids null pointer errors by calls through an object reference variable ν after it was read destructively (to transfer the target object as a parameter or to call a mutator).

2. INTERPRETATION $\nrightarrow$ COMPUTATION. As a whole, the mode annotations in a program superimpose a *higher-level view*, a *structural interpretation* in terms of composite objects, on the state and the computation. And the purpose of mode checking is to enforce the *consistency of this view* in the dynamic, evolving system. In particular, it prevents (the expression of) an interpretation where an object might have two owners, i.e., is component of two different composites. And it excludes the interpretation of a (dynamic) set $\mathcal{O}$ of elementary objects as one composite object O with representative $o \in \mathcal{O}$ if O's state might change without going through a method of o (state encapsulation at the composite object level).

That is, from the perspective of a given Java program p, the mode system rejects mode-annotated programs p' expressing the wrong higher-level interpretation. But it is equally right to say from the perspective of a given higher-level interpretation, that the mode system rejects (Java or JaM) programs expressing certain computations. This should be the normal perspective of a software developer programming computations in the context of his higher-level view of the system. The former perspective corresponds more to a reverse engineering situation, where one tries to reconstruct a (possible) higher-level view by analyzing a Java program.

3. JAM $\cong$ JAVA. JaM and the Java subset on which it is based allow one to program the same set of computations, they are *computationally equivalent*: On one hand, modulo modes, all JaM computations are Java computations since JaM is a pure type system extension. Without new mechanisms nor values, it is simply not possible to express any new computations in JaM. All legal JaM programs p' can be translated to equivalent legal Java programs p that perform, modulo modes, the same computation: Remove all modes and correlations, all '`mut`', '`obs`', and '`val`', and expand all destructive reads to a read and an assignment of `null`.

On the other hand, JaM's type system extension does not make any old computations impossible to express since one structural interpretation always works: The system is viewed as a "sea of objects" [Bos96], in which all objects are co-objects at the same object composition level; in which there is no composition hierarchy. All legal programs p of JaM's Java subset can be translated to legal JaM programs p' (with "sea of objects" structure) by annotating the declarations of all fields, variables, parameters, and results of class types, with `co<>`, by annotating `new` and `null` with the empty correlation-set `<>`, by annotating all methods and operations with `mut`, and by annotating any r-value occurrence of a variable name with `val`. To be precise, we need a JaM in which the `free<>` references originating from `new` can be converted to `co<>` references (cf. base-JaM, where this was possible).

4. MORE JAVA FEATURES. The language JaM (Java with modes) defined formally in the previous chapter was a very reduced language comprising only the most essential features necessary to demonstrate composite object encapsulation by a static mode system. In order to write programs in the normal Java way, one would surely want to have a more complete Java-with-modes including also *implicit* destructive/non-destructive read access to variables, *primitive values* (arithmetics, boolean logic and characters), *strings* and *arrays*, `for` and `switch`, `interfaces` and *subclassing* and *dynamic casts*, *access specifiers* (`public`, `protected`, `private`), user-defined *constructors*, etc.

Java-subsets with all of the named features have, on one hand, repeatedly been proved to be type safe [Sym97, DE97, Ohe01]. Hence one can be optimistic that JaM's type consistency theorem (Theorem 6) can be generalized to an extended JaM with all these features included. On the other hand, all the features that were listed above allow *no new object graph changes*: References of the same modes as before are required for any reduction step that adds a reference of a particular mode. Therefore all results on object ownership and state encapsulation can easily be generalized—based on the extended type consistency theorem—to the more complete JaM.

There are also other interesting Java features whose use would allow new kinds of changes to the object graph. For example, access to *other objects' fields* (particularly of object reference types); *static variables* of object reference types and the access to them; *static methods* operating on object references; and *inner classes* and their instances' implicit access to the outer class instance's fields and methods. For this kind of features, special mode-based checks would be necessary, and extending the results on object ownership and state encapsulation would require real extra work.

The addition of subclass polymorphism and inheritance, and of static members will be considered in more detail in §7.2.

5. MODES $\perp$ JAVA TYPES. Java has two categories of types: *primitive types* and *reference types*, and correspondingly two categories of first-class values: *primitive values* and *reference values* [GJS00]. "The values of a reference type are references to objects," where an object is "a dynamically created instance of a class type or a dynamically created array." The quotation makes clear that in Java, writing the name c of an object class as the range type t of a variable is just syntactic sugar for a type of *references* to objects of class c or a subclass. Other programming languages, which do not have Java's implicit reference semantics, explicitly construct the type of the references from the type of the references' targets: `pointer of` c in Pascal, `access` c in Ada, `ref` c in ML, and $*c$ in C++.

In JaM, Java's class names are annotated with modes where and only where they are used as types of object references. They are not annotated to class names occurring in object creation expressions, nor in `implements` and `extends` clauses, nor in qualified names. Modes $\mu \in \mathcal{M}$ may appear like (a family of) *type constructors* $\mu : \mathbb{C} \to \mathcal{T}$ to construct reference types from class names (object types). But the different modes in different reference types mean neither that their values, the moded object references, nor the objects they target, are constructed in any way different.

Modes are better understood as *type qualifiers* for Java's reference types (that are still implicitly constructed from class names). They classify reference values w.r.t. their role in object composition (cf. paragraph *2*). As in other systems with type qualifiers, the JaM type system "guarantees that in every program execution the semantic properties captured by the qualifier annotations are maintained" [FFA99]. A mode classification is not simply right or wrong w.r.t. a property of the classified value. It can only be consistent, or not, with the classification of the other values in the system. Modes let programmers thinking in terms composite objects indirectly express powerful global invariants about the object graph and call stack, which the JaM type system enforces statically (uniqueness of ownership, control of mutator executions, etc.). Hence modes fit into Foster, Fähndrich, and Aiken's vision of programmers expressing the interesting, strong invariants they know about their programs through easy to understand type qualification.

6. COMPARISON: DIMENSION QUALIFICATION. Modes are similar to the qualification of numeric types and numerals with *physical dimensions* (length, time, mass, voltage, ...) in physical formulæ and in some proposed extensions to programming and specification languages [Ken94, HM95, Ken97]. Through qualifications 4m, 4s, 4g, 4A, etc., different physical meanings can be superimposed on the number four. (We ignore here the differentiation between different units of measurement for the same dimension.) Since this meaning transcends the values' computational meaning, the set of possible computations is unchanged. The distinction into different physical dimensions places "a useful typing structure on the otherwise homogeneous field of real numbers" [HM95]. And *dimensional analysis* ensures that values of one dimension are never used as, compared with, or added to, values of another dimension, and that multiplications and divisions are assigned the correspondingly multiplied and divided dimensions. Since all dimensions are treated uniformly in dimension analysis and since they have no impact on the computation, there is no real need to introduce a dimension by a declaration before it can be used. A declaration could not define what, e.g., "`volt`" is, except in the form of a comment or relative to other dimensions. It would only help to catch mistyped dimension names in the program.

Like dimensions, modes also superimpose a meaning (on reference values) that transcends the computation (with reference values): The combination $12\Omega \cdot 4\text{A}$ of two dimensioned values by an operation called *multiplication* produces a value $48\Omega\text{A} = 48\text{V}$ with multiplied dimension. Similarly, the combination $\mathbf{s} \xrightarrow{\mu_\mathbf{r}} \mathbf{r} \circ \mathbf{r} \xrightarrow{\mu_\mathbf{o}} \mathbf{o}$ of two references to one by an operation called *return* (of result $\mathbf{r} \xrightarrow{\mu_\mathbf{o}} \mathbf{o}$ through call-link $\mathbf{s} \xrightarrow{\mu_\mathbf{r}} \mathbf{r}$) yields a value $\mathbf{s} \rightarrow \mathbf{o}$ whose mode is the "multiplication" $\mu_\mathbf{r} \circ \mu_\mathbf{o} = \mu'$ of the combined references' modes.

Mode checking imposes more complex constraints on the use and combination of moded reference values than dimension analysis. `free`, `rep`, `co`, and `read` modes are treated specially in accordance with a predefined meaning w.r.t. object composition. Only the different association roles are treated uniformly since their meaning is undeclared and application-specific. A declaration of association roles `elem`, `dest`, `key`, etc. could not define what it means to be an element in a set, a destination of an iterative traversal, or a key in a map.

7.1.2 Submode Polymorphism?

7. No Submode-Polymorphic Reference Values. The classification of object reference values by modes is a *monomorphic* classification, it is not polymorphic like the classification by the target's class (subclass polymorphism). The same reference value cannot normally be assigned two modes. In particular, a compatibility $\mu\ c \leq_\mathrm{m} \mu'\ c$ does not mean that a $\mu\ c$ value *is* also a $\mu'\ c$ value. It merely says that it is safe to *change* a value of mode μ into another value of mode μ'. The mode compatibility relation $\leq_\mathrm{m}$ is no *is-a* or *generalization* relation like the subclass relation $\leq_\mathrm{c}$ in Java (cf. §7.2.2).

For instance, a reference value $h = \langle \mathbf{s}, \texttt{free<>}, \omega \rangle$ is not a special case of a `rep<>` reference since `free<>` references do not mean inclusion in the source's sanctuary. (On the other hand, it seems right to say that any m`<>` reference is at the same time also a `read<>` reference.) If h is assigned to a variable x of mode `rep<>` then x should not mode-polymorphically contain the `free<>` value h. It is the *intention* of the assignment that h be converted to mode `rep<>` in order to add ω to the source's sanctuary $\mathit{Sanc}(\mathbf{s})$. In type inference, it would be unsafe to "approximate" an object reference's type $\mu\ c$ by a type $\mu'\ c \geq_\mathrm{m} \mu\ c$ to which it is mode-compatible (at least in the case that $\Sigma(c)$ contains operations with `co`-parameters): If an expression e evaluating to reference $\mathbf{s} \xrightarrow{\texttt{free<>}} \mathbf{r}$ was typed with mode `rep<>` instead of `free<>`, then the typing rule for operation call expressions with e as receiver expression would allow $\mathbf{s}$ to supply a reference $\mathbf{s} \xrightarrow{\texttt{rep<>}} \mathbf{o}$ to $\mathbf{r}$ as a `co` parameter value $\mathbf{r} \xrightarrow{\texttt{co<>}} \mathbf{o}$. But then an old alias of the `rep` reference and the new `free` path $\mathbf{s} \xrightarrow{\texttt{free<>}} \mathbf{r} \xrightarrow{\texttt{co<>}} \mathbf{o}$ would violate the Unique Head property.

8. Covariant Parameter Modes. If an object reference is mode-converted from compatible $\mu \leq_\mathrm{m} \mu'$ to μ' then the import of the the target's `co` results *and* parameters in the handle's signature changes covariantly from μ to μ'. In case of a depth-conversion, also the import of an association role in result *and* parameter modes in the handle's signature changes covariantly from μ to μ'. For example, the method SetNext of Node objects has the type (co<> Node) $\xrightarrow{\mathtt{mut}}$ co<> Node in signature Σ(Node). Its type in the signature $\Sigma(\mu$ Node) of μ-references to Node objects is (μ Node) $\xrightarrow{\mathtt{mut}}$ μ Node. Hence if μ changes to rep<elem=rep<>> from compatible mode free<elem=rep<>>, then result and parameter mode both change covariantly:

mode μ	type of SetNext in $\Sigma(\mu$ Node)
free<elem=rep<>>	(free<elem=rep<>> Node) $\xrightarrow{\mathtt{mut}}$ free<elem=rep<>> Node
$\leq_{\mathrm{m}}$ rep<elem=rep<>>	(rep<elem=rep<>> Node) $\xrightarrow{\mathtt{mut}}$ rep<elem=rep<>> Node

Covariance in parameter type position is known to be incorrect in polymorphic typing systems that assign a function expression a more or less specific function type [Gun92]. But modes in handle signatures $\Sigma(\mu\ c)$ are not approximations of modes in the target object's interface. If a (potential) sender mode-converts its call-link, nothing changes for the receiver object, its method will still receive actual parameters of the formal parameter type. It changes how the call-link mode-translates the receiver's reference values and formal parameters for the sender. The problem is not correct approximation of modes in the receiver but the consistency of mode-translation with other references to the same object.

7.1.3 Limits of Inter-Object Data and Control Flow

In object systems, data and control is passed between the system components (the objects) through the object references connecting them by method invocation and result messages. The flow of messages invoking *mutator* methods is restricted by the JaM type system to ensure composite state encapsulation. The flow of *object reference values* as data values in the messages is restricted to preserve the mode-specified higher-level interpretation's integrity (§6.2.3). The latter restrictions are encoded in the rule for the signature $\Sigma(\mu\ c)$ of μ-handles to c-objects, repeated here in figure 7.1.

9. Unconstrained Transitive Free/Read Values. The reference value flow is not constrained generally: Some reference values may be passed from any object to any other object, namely, references whose modes μ contain only the base-modes free and read ($\forall \vec{\alpha} \in \mathbb{A}^* \centerdot \mu(\vec{\alpha})$ is defined $\Rightarrow \mu(\vec{\alpha}) \in \{\mathtt{free}, \mathtt{read}\}$). Such modes never change by mode import '$\circ$' ($\mu_{\mathbf{r}} \circ \mu = \mu$), irrespective of the call-link's mode $\mu_{\mathbf{r}}$. And only parameters and results whose mode contains rep, co or an association roles are subject to some flow constraint or the other.

"Transitive read references" are reference values whose mode contains only read, so that following references through them always produces another transitive read reference, except only for "following" a returned free reference. For such references it makes perfect sense that they can flow through the entire object system: Between

$$\frac{\begin{array}{l}\vdash \mathit{FldsMths}(c) = \langle \Gamma, F\rangle \quad F(f) = \kappa\ \mu\ d\ f\,\mathtt{(}\overline{\mu_i\ d_i\ y_i}\mathtt{)\{\ldots\}} \\ \forall i, \vec{\alpha}\bullet ((\mu_{\mathbf{r}}\circ\mu_i)(\vec{\alpha}) = \mathtt{read} \Rightarrow \mu_i(\vec{\alpha}) = \mathtt{read}) \wedge (\mu_i(\vec{\alpha}) \in \{\mathtt{co}\} \cup \mathbb{A} \Rightarrow \mu_{\mathbf{r}}(\epsilon) \notin \{\mathtt{read}\} \cup \mathbb{A}) \\ \forall \vec{\alpha}, \alpha\bullet\ \mu(\vec{\alpha}) = \mathtt{free} \wedge \mu(\vec{\alpha}.\alpha) \in \mathbb{A} \wedge \mu_{\mathbf{r}}\circ\mu(\vec{\alpha}.\alpha) \neq \mathtt{read} \Rightarrow \mu_{\mathbf{r}}(\epsilon) \neq \mathtt{read}\end{array}}{\vdash (\ f : \overline{\mu_{\mathbf{r}}\circ\mu_i\ d_i} \xrightarrow{\kappa} \mu_{\mathbf{r}}\circ\mu\ d\ ) \ \in\ \Sigma(\mu_{\mathbf{r}}\ c)}$$

Figure 7.1: The signature of handles (repeated)

any two objects there may be `read` references, and extensions of `read` references to `read` paths without having any impact on object ownership or state encapsulation.

The unconstrained flow of `free` references means that objects created in one part of the object system can migrate to any other part of the system, provided their flow is not constrained by non-`read` correlations. There is no problem here too, except for one suspicious situation: If the `free` reference that is, or can be extended to, a `free` path $o \dashrightarrow \omega$ is passed to the `free` object ω itself or a sub-object of it, then this creates a cycle of ownership paths through ω. While this is safe w.r.t. object ownership and state encapsulation, it leads to a breakdown of the ownership path-based structural interpretation of the object system in terms of composite objects: There is no such thing like "*cyclic object composition.*" No notion of parthood permits an object to be its own proper part [OMG00, Sim87, Var96]. However, the corruption of the composition hierarchy would only be temporary: The mutator control path property (Theorem 8) guarantees that if ω possesses its own `free` reference, it cannot be on the stack, so that ω cannot be executing a mutator. And sub-objects of ω could obtain it as parameters of a mutator only from ω or other sub-objects of ω.

Along `read` and association references, transitive read/free references are the only reference values that can flow. This makes sense since a `read` or association reference does not tell the source where its target is in the object composition hierarchy (absolutely or relative to its own position). It could be anywhere. For every other reference value there is a good reason, explained in §6.2.3, why forward flow along `read` and association references is not generally safe.

10. HIERARCHICAL ASSOCIATION FLOW. `Free` and `rep` references as call-links give their sources not only privileges w.r.t. calling mutators, but also transport more references than any other call-link. In particular, parameters with an association role α in their mode may be supplied only through references $o \xrightarrow{\mathtt{free}\langle\alpha=\mu\rangle} \omega$ and $o \xrightarrow{\mathtt{rep}\langle\alpha=\mu\rangle} \omega$ which correlate this association role.

This means, on one hand, that the representatives controls the flow of *association references* from outside into the composite's interior: Association references $\omega \xrightarrow{\alpha} q$ can arrive in ω *by parameter* only if representative o supplies a reference $o \xrightarrow{\mu} q$. And only the representative o can give ω a reference $\omega \xrightarrow{\mathtt{read}\langle\beta=\alpha\langle\rangle\rangle} q'$ or $\omega \xrightarrow{\mathtt{free}\langle\beta=\alpha\langle\rangle\rangle} q'$ through which ω can obtain association references $\omega \xrightarrow{\alpha} q$ *as results* (and through which further references $\omega \xrightarrow{m\langle\beta=\alpha\langle\rangle\rangle} q''$ can be obtained by following q''s co-references). This way, the representative limits the (groups of) external objects with which its sub-objects can enter into association.

On the other hand, representatives should be able to store through a `free` or `rep` reference any of their reference values as association references in a generic container object like a `Set` and `Map`. The mode system defined in the previous chapter supports only the storage of `rep` and association references in generic containers. Storage of `free` and `co`-references fails since it would make a correlation on the component reference necessary that makes its mode invalid. And despite valid correlations α=`read`(...), the representative was not allowed to pass a `read` reference as an association moded parameter of its component. (While this prevents to store `read` references in generic containers, `read` references can of course always be stored in other objects *as* `read` *references*.) Extending the support for container objects to containers of `read`, `co` and `free` references, and relaxing the restrictions on correlations are left as subjects of future work.

11. THE STRUCTURE OF MESSAGE FLOW IN JAM. The integrity of the higher-level view demands only one limitation of the flow of mutator messages through the object system, namely the one captured in the mutator control invariant. Like all static type systems, the JaM type system excludes many message flows that would be safe. In particular, JaM allows invocations, even of observers, only if all reference parameters and results may be exchanged, and allows mutator messages to flow only along base-JaM-like ownership paths. These restrictions could safely be relaxed at the cost of making the mode system more complex.

In JaM, mutators messages may flow only along `free`, `rep` and `co` references, and along the latter two only if they are sent from a mutator method. Hence, all mutator flow starts with a mutator message sent along a `free` references, followed by mutator messages sent along `rep` and `co` references. In particular, control can reach the mutators of a state-representing component ω from its owner's mutators only by flowing *downward* along a `rep` reference either directly to ω or first to a co-object of ω and then *sideways* along `co` references. Once this hierarchical mutator flow is interrupted by the call of an observer (downward or across the hierarchy), one can come back to mutator messages only by a call through a `free` reference. Hence, since program execution starts with the call of observer `main`, the path of calls from program start to the current method always has a form captured in the following regular expression:

$$\Big(\underbrace{(\xrightarrow{\mu} \texttt{obs})^*}_{\text{any direction}} \quad \underbrace{\xrightarrow{\texttt{free}} \texttt{mut}(\xrightarrow{\texttt{co}} \texttt{mut})^* \left(\xrightarrow{\texttt{rep}} \texttt{mut}(\xrightarrow{\texttt{co}} \texttt{mut})^*\right)^*}_{\text{downward}} \Big)^*$$

where $\xrightarrow{\mu} \kappa$ stands for the call of a κ-method through a μ-reference.

Permitting more cases of safe mutator calls would require refining the rules for signature import, or refining the classification of references or of methods. A refinement of the import rule could decouple the question of permitted parameter and result exchange from the question of permitted invocation. This will be considered in §7.2.3. A finer classification of *references* by additional modes can single out more references through which mutators may always safely be sent. Some such new modes will be considered in §7.2.4. A finer classification of observer and mutator *methods* according to the objects they might mutate can identify those methods which may only be executed while these objects' owner is executing a mutator. Such an extension would make the mode system look more like a (simple) *effects system.* It will be explored in §7.3.2.

7.1.4 Consistency of Reference Value Flow

When a reference value is passed from one object to another, its mode may change (from $\mu_i = \mu$ to $\mu_{i+1} = \mu_{\mathbf{r}}\circ\mu$, or vice versa). A consistency property of reference value flow is that an object, by enlisting the service of other objects, should not be able to change the modes of its reference values other than it could do itself by mode conversion. Let us look at just two scenarios:

12. NEARLY MONOTONE. An object o might supply a reference $h = \mathbf{s} \xrightarrow{\tilde{\mu}} \mathbf{o}$ to an object $\mathbf{r}$ through a reference $h_{\mathbf{r}} = \mathbf{s} \xrightarrow{\mu_{\mathbf{r}}} \mathbf{r}$ as a parameter reference $h' = \mathbf{r} \xrightarrow{\mu} \mathbf{o}$ if $\tilde{\mu} = \mu_{\mathbf{r}}\circ\mu$. $\mathbf{s}$ could convert $h_{\mathbf{r}}$ to $h'_{\mathbf{r}}$ of mode $\mu'_{\mathbf{r}} \geq_{\mathrm{m}} \mu_{\mathbf{r}}$, and $\mathbf{r}$ could convert h' to h'' of mode $\mu' \geq_{\mathrm{m}} \mu$. If then h'' is returned back to $\mathbf{s}$, we get a reference $h''' = \mathbf{s} \xrightarrow{\tilde{\mu}'} \mathbf{o}$ of mode $\tilde{\mu}' = \mu'_{\mathbf{r}}\circ\mu'$. Consistency in this case demands that the original h was compatible to h'''. In other words, we should have the algebraic property of monotonicity: $\tilde{\mu} = \mu_{\mathbf{r}}\circ\mu \leq_{\mathrm{m}} \mu'_{\mathbf{r}}\circ\mu' = \tilde{\mu}'$. (The monotonicity is a partial one since $\circ$ is a partial mapping.)

To see that mode import $\circ$ is monotone in the right argument, consider elementary conversions $\mu \leq^1_{\mathrm{m}} \mu'$: If μ was $\texttt{co<>}$ or $\alpha\texttt{<>}$ then $\mu' = \texttt{read<>}$, so that $\mu_{\mathbf{r}}\circ\mu = m\texttt{<}\delta\texttt{>} \leq_{\mathrm{m}} \texttt{read<}\delta\texttt{>} \leq_{\mathrm{m}} \texttt{read<>} = \mu_{\mathbf{r}}\circ\texttt{read<>} = \mu_{\mathbf{r}}\circ\mu'$. If μ was $m\texttt{<}\ldots,\alpha_i\texttt{=}\mu_i,\ldots\texttt{>}$ with $m = \texttt{free}$ or $\texttt{rep}$ then $\mu_{\mathbf{r}}\circ\mu$ is $\tilde{m}\texttt{<}\ldots,\alpha_i\texttt{=}\mu_{\mathbf{r}}\circ\mu_i,\ldots\texttt{>}$ with $m = \texttt{free}$ or $\texttt{read}$, and μ' is $m'\texttt{<}\ldots,\alpha_i\texttt{=}\mu_i,\ldots\texttt{>}$ with $m = \texttt{rep}$ or $\texttt{read}$, so that we have $\mu_{\mathbf{r}}\circ\mu' = \texttt{read<}\ldots,\alpha_i\texttt{=}\mu_{\mathbf{r}}\circ\mu_i,\ldots\texttt{>}$. Hence $\mu_{\mathbf{r}}\circ\mu \leq_{\mathrm{m}} \mu_{\mathbf{r}}\circ\mu'$. If μ was $\texttt{read<}\ldots,\alpha_i\texttt{=}\mu_i,\ldots\texttt{>}$ and μ' has less or weaker correlations, then $\mu_{\mathbf{r}}\circ\mu = \texttt{read<}\ldots,\alpha_i\texttt{=}\mu_{\mathbf{r}}\circ\mu_i,\ldots\texttt{>}$, and $\mu_{\mathbf{r}}\circ\mu'$ has fewer correlations or a correlation $\alpha_i\texttt{=}\mu_{\mathbf{r}}\circ\mu'_i$ with $\mu_i \leq^1_{\mathrm{m}} \mu'_i$. In the first case, $\mu_{\mathbf{r}}\circ\mu \leq_{\mathrm{m}} \mu_{\mathbf{r}}\circ\mu'$ by width compatibility. In the second case, $\mu_{\mathbf{r}}\circ\mu_i \leq^1_{\mathrm{m}} \mu_{\mathbf{r}}\circ\mu'_i$ can be shown by induction, so that $\mu_{\mathbf{r}}\circ\mu \leq_{\mathrm{m}} \mu_{\mathbf{r}}\circ\mu'$ by depth compatibility.

Mode import $\circ$ is not monotone in the left argument, but $\mu_{\mathbf{r}}\circ\mu \leq_{\mathrm{m}} \mu'_{\mathbf{r}}\circ\mu$ holds where it matters: If $\mu = \texttt{co<>}$ then $\mu_{\mathbf{r}}\circ\mu = \mu_{\mathbf{r}} \leq_{\mathrm{m}} \mu'_{\mathbf{r}} = \mu'_{\mathbf{r}}\circ\mu$. If $\mu = \alpha\texttt{<>}$ then $\mu_{\mathbf{r}} = m\texttt{<}\ldots,\alpha\texttt{=}\mu_\alpha,\ldots\texttt{>}$ and $\mu'_{\mathbf{r}} = m'\texttt{<}\ldots,\alpha\texttt{=}\mu'_\alpha,\ldots\texttt{>}$ with $\mu_\alpha \leq_{\mathrm{m}} \mu'_\alpha$ (depth compatibility), so that $\mu_{\mathbf{r}}\circ\mu = \mu_\alpha \leq_{\mathrm{m}} \mu'_\alpha = \mu'_{\mathbf{r}}\circ\mu$. If $\mu = m\texttt{<>}$ with $m = \texttt{free}$, $\texttt{read}$ or $\texttt{rep}$ then $\mu_{\mathbf{r}}\circ\mu = \mu'_{\mathbf{r}}\circ\mu$. The same obviously holds for if we add correlations $\alpha_i\texttt{=}\mu_i$ with $\mu_{\mathbf{r}}\circ\mu_i = \mu'_{\mathbf{r}}\circ\mu_i$. If μ contains a correlation $\alpha_i\texttt{=}\mu_i$ with $\mu_{\mathbf{r}}\circ\mu_i \neq \mu'_{\mathbf{r}}\circ\mu_i$, this presupposes that μ_i is an association mode, or contains one. The latter case can be followed by structural induction. If μ_i is $\beta()$ with $\mu_{\mathbf{r}}\circ\beta\texttt{<>} \neq \mu'_{\mathbf{r}}\circ\beta\texttt{<>}$, $\mu_{\mathbf{r}}$ must have been depth-converted (see above), so that $\mu_{\mathbf{r}}\circ\beta\texttt{<>} \leq_{\mathrm{m}} \mu'_{\mathbf{r}}\circ\beta\texttt{<>}$. If $\mu = m\texttt{<}\ldots,\alpha_i\texttt{=}\mu_i,\ldots\texttt{>}$ is $\texttt{read}$ or `rep` then $\mu_{\mathbf{r}}\circ\mu = \mathtt{read<}\ldots,\alpha_i\mathtt{=}\mu_i,\ldots\mathtt{>} \leq_{\mathrm{m}} \mathtt{read<}\ldots,\alpha_i\mathtt{=}\mu'_i,\ldots\mathtt{>} = \mu'_{\mathbf{r}}\circ\mu$. If μ is `free` then $\mu_{\mathbf{r}}\circ\mu = \mathtt{free<}\ldots,\alpha_i\mathtt{=}\mu_i,\ldots\mathtt{>} \not\leq_{\mathrm{m}} \mathtt{free<}\ldots,\alpha_i\mathtt{=}\mu'_i,\ldots\mathtt{>} = \mu'_{\mathbf{r}}\circ\mu$ since only `read` modes are depth-compatible in $\leq_{\mathrm{m}}$. This particular case of violated monotonicity, however, is irrelevant for the consistency of reference value flow: To convert the call-link $h_{\mathbf{r}} = \mathbf{s} \xrightarrow{\mu_{\mathbf{r}}} \mathbf{r}$ to $h'_{\mathbf{r}} = \mathbf{s} \xrightarrow{\mu'_{\mathbf{r}}} \mathbf{r}$, the call through $h_{\mathbf{r}}$ must store h' in $\mathbf{r}$ and return, and a second call $h'_{\mathbf{r}}$ must retrieve it from $\mathbf{r}$. But retrieving a `free` μ-handle from a field requires destructive read and thus a mutator call, while depth-conversion of $h_{\mathbf{r}}$ implied that $\mu'_{\mathbf{r}}$ is `read`, so that a mutator cannot be called.

13. Partial Associativity. Given a path $\pi = o \xrightarrow{A} p \xrightarrow{B} q$ of two references, there are two ways of passing a reference value h from o to q or vice versa. Let $h_q = q \xrightarrow{\mu} \mathbf{o}$ be the reference value when q has it and $h_o = o \xrightarrow{\mu'} \mathbf{o}$ when o has it. That is, μ-reference h_q returned from q is imported by o as the μ'-reference h_o, and μ-reference h_o supplied by o is received by q as μ-reference h_q. If h is passed via p, then the intermediate reference is $p \xrightarrow{B\circ\mu} \mathbf{o}$, and $\mu' = A\circ(B\circ\mu)$. Alternatively, π can first be shortened to $\pi' = o \xrightarrow{A\circ B} q$ by returning π's second through its first reference. If h is passed through this shortcut, $\mu' = (A\circ B)\circ\mu)$. For consistency, the transport of h through the original path and through the shortcut should be equivalent. If o passes its reference value h_o to q on one way, and retrieves it back the other way, the mode should be unchanged. In other words, we need the algebraic property of associativity: $A\circ(B\circ\mu) = \mu' = (A\circ B)\circ\mu$. (The associativity is a partial one since $\circ$ is a partial mapping.)

If $\mu = \mathtt{free<>}$ then $A\circ(B\circ\mu) = \mathtt{free<>} = (A\circ B)\circ\mu$. If $\mu = \mathtt{read<>}$ or $\mathtt{rep<>}$ then $A\circ(B\circ\mu) = \mathtt{read<>} = (A\circ B)\circ\mu$. If $\mu = \mathtt{co<>}$ then $A\circ(B\circ\mu) = A\circ B = (A\circ B)\circ\mu$. If $\mu = \alpha\mathtt{<>}$ then B must have a correlation $\alpha\mathtt{=}\mu_\alpha$. Consequently, $A\circ B$ has the correlation $\alpha\mathtt{=}A\circ\mu_\alpha$. Hence $A\circ(B\circ\mu) = A\circ\mu_\alpha = (A\circ B)\circ\mu$. If $\mu = m\mathtt{<}\ldots,\alpha_i\mathtt{=}\mu_i,\ldots\mathtt{>}$ then $A\circ(B\circ\mu) = m'\mathtt{<}\ldots,\alpha_i\mathtt{=}A\circ(B\circ\mu_i),\ldots\mathtt{>}$ and $(A\circ B)\circ\mu = m'\mathtt{<}\ldots,\alpha_i\mathtt{=}(A\circ B)\circ\mu_i,\ldots\mathtt{>}$, so that $A\circ(B\circ\mu) = (A\circ B)\circ\mu$ follows by induction.

A stronger form of consistency of mode import than JaM's associativity can be formulated in ownership-based systems that give moded type terms t an absolute interpretation $[\![t]\!]_o$ relative to each object o in terms of ownership: In Ownership Types [CPN98] and Universes [MP01, MP99a], consistency is expressed as the equivalence $[\![t_A * t_B]\!]_o = [\![t_B]\!]_p$ of how p interpets type term t_B and how objects o with a t_A-reference to p interpret the combined type term $t_A * t_B$.[1]

14. Pitfalls of Depth/Width-Compatibility. When defining the mode compatibility relation $\leq_{\mathrm{m}}$ in §6.2.3, the compatibility between modes with changed correlation sets, width-compatibility (more or fewer correlations) and depth-compatibility (correlations with compatible modes), was limited to `read` modes. Without this restriction, it would be possible for an object to set up object references through which

[1]This equivalence is stated as the *combination lemma* in [MP01, MP99a], and as part of the *visibility lemma* in [CPN98]. $t_A * t_B$ would in [CPN98] be written $\psi(t_A)(t_B)$ or "$\sigma(t_B)$ with $\sigma = \psi(t_A)$" and $[\![t_B]\!]_p$ would in [MP01, MP99a] be written $\tau(t_B, p)$.

```
mut void  gotcha1(read<> T  dont_mut)
{  rep<data=rep<>> Node<T>  repnode  = new Node<T>();
   rep<data=read<>> Node<T> readnode = repnode; // depth-compatible? - no!
   rep<> T                  can_mut;

   readnode.SetData(dont_mut); // store one way
   can_mut = repnode.data();   // retrieve other way
   can_mut.Mutate();           // oops!
}
mut void  gotcha2(read<> T  dont_mut)
{  rep<data=rep<>> Node<T>  repnode  = new Node<T>();
   rep<data=read<>> Node<T> readnode = new Node<T>();
   rep<> T                  can_mut;
   rep<> Node<T>  hiddenrep  = repnode;  // width-compatible? - no!
   rep<> Node<T>  hiddenread = readnode; // width-compatible? - no!

   hiddenrep.SetNext(hiddenread); // links repnode --next--> readnode
   repnode = repnode.next();      // now repnode==readnode
   readnode.SetData(dont_mut);    // store one way
   can_mut = repnode.data();      // retrieve other way
   can_mut.Mutate();              // oops!
}
```

Figure 7.2: Dangerous width- and depth-conversion

it can exchange references in such a way that can effectively convert `read` reference to `rep`:

Depth-compatibility, for instance, would allow an object to weaken the `rep<data=rep<>>` reference to a `Node` object to a `rep<data=read<>>` reference. Through this reference the source could store a `read` reference in the `Node` as a `data` reference and read it back through (a copy of) the original reference as a `rep` reference. The code in figure 7.2 shows how an object can use this trick to obtain surreptitious write access to an object to which it was given a `read` reference by converting it to `rep`.

The same scenario can be set up using width-compatibility: A `rep<data=rep<>>` reference and a `rep<data=read<>>` reference to two `Node` objects could be converted to the same mode `rep<>`, and then linked by a co-reference. By reading it back through the original references, the source can obtain, as with depth-compatibility, a `rep<data=rep<>>` reference *and* a `rep<data=read<>>` reference to the same node as demonstrated in figure 7.2.

This property of type systems regarding qualified reference types is not new. It is known from the type system of C++ [ISO98], where it concerns the `const` qualification of pointer types. Pointers are compatible to `const` pointers, i.e., read-only pointers, of the same type: `T*` $\leq$ `const T*`. But pointers to variables of these pointers are not compatible, `T**` $\not\leq$ `const T**`: Converting a `T**` pointer to `const T**` would allow one to store, through the new pointer, a `const` pointer into a variable and retrieve it as a non-`const` pointer through a copy of the old pointer. Pointers to pointers are, however, compatible to ***const*** pointers to `const` pointers: `T**` $\leq$ `const T*const*`. Converting a `T**` pointer to `const T*const*` is safe since the new poitner can never be used to *store* a `const` pointer into the variable.

In the same fashion, depth- and width-compatibility in JaM's type system exists only between ***read*** modes: `rep<data=rep<>>` $\not\leq_{\mathrm{m}}$ `rep<data=read<>>` and `rep<data=rep<>>` $\not\leq_{\mathrm{m}}$ `rep<>`, but `read<data=rep<>>` $\leq_{\mathrm{m}}$ `read<data=read<>>` and `read<data=rep<>>` $\leq_{\mathrm{m}}$ `read<>`. The `read` modes are compatible because through correspondingly convert `read` references nothing can be stored in the target (since only observers can be called on the target).

7.1.5 Precursors of JaM's Base-Modes

The base-modes of JaM's mode system have precursors in other work.

1. '`rep`' for the class of object references from an object to its state-representing components appeared first in Flexible Alias Protection as an *alias mode* [NVP98], and then in its successor Ownership Types as an *ownership context* [CPN98] and in Universes [MP01]. The user-specified distinction of `rep` references by some kind of annotation is fundamental to nearly all typing disciplines for composite object encapsulation. The same function as '`rep`' serves the '`part`' section in LOOPS classes [SB85], the '`internals`' section in Sina classes [AW+92], the *reference type attribute* '`private`' in [KM95], the '`pivot`' predicate of [DLN98], the '`unshared`' annotation in [GB99], and the *alias annotation* '`owned`' in [ACN02].

2. '`free`' appeared first as an *access mode* in Islands [Hog91], from where it was assumed by Flexible Alias Protection [NVP98]. Hogg [Hog91] defined `free` as indicating references to whose target no other reference exist anywhere in the system. The initial reference to a new object was always `free`, could be exchanged between objects, and converted to other modes. Hogg's `free` *variables* could be accessed only by destructive read. A similar function serves the '`virgin`' references of [LS97, DLN98], which point to objects which have never been targets of field-captured references, and the '`unique`' references in [GB99] and [ACN02]. New in JaM is the interpretation of the targets of `free` references as *behavioral*, not state-representing components of the source. In JaM, `free` references can have aliases, namely, `co` and `read` aliases, and the `free` reference as well as its aliases can be captured.

3. '`co`' is JaM's name for the class of references guaranteed to connect two objects with the same owner, i.e., two "co-objects." It appeared first under the name '`protected`' in [KM95] and '`owner`' in Ownership Types [CPN98]. In Universes, it is the unmarked default [MP01]. `Co` references are important for the proper typing of the `this` reference. This was first observed by Clarke, et al. [CPN98].

4. '`read`', and its long form '`readonly`', are common type qualifiers to classify references through which no mutator operations may be called. In typing disciplines for composite object encapsulation, they occurred already in Islands [Hog91], in [KT99], and in Universes [MP01]. `Read` references in Islands cannot be assigned to variables but they can be bound to parameters (and returned, used as call-link, ...). `Readonly` in [KT99] means *transitive* read-only: All references returned through them are imported as `readonly`. JaM's `read` references are simple read-only reference, like Universes' `readonly` references.

5. Association roles $\alpha \in \mathbb{A}$, for a user-defined classification of object references according to different semantic roles (*associations* [OMG00], *relationship types*) appeared first in Flexible Alias Protection [NVP98] as "argument roles" in *alias modes* `arg` α and `var` α (with/without limitation to the target's "clean" interface). Ownership Types [CPN98] had only plain identifiers α with the same function as `var` α, but now understood as (ownership) *context parameters* to the class.

7.2 Shortcomings and Extensions

7.2.1 Syntactic Sugar

Ubiquitous explicit read access to variables and bulky mode annotations make JaM programs difficult to read and tedious write. But programming language research is not a beauty contest. To sweeten some bitter aspects of the JaM syntax, "syntactic sugar" can be offered that give JaM programs a look and feel much closer to traditional Java programs. Appendix B.3 demonstrates this for the code of the map example.

1. In sugared JaM, read access to variables is implicit as in Java. Every occurrence of a field or local variable name ν at an r-value position can be desugared by the following rules:

- ν stands for `val(`ν`)` if ν is a non-`free` variable or if ν is a `free` variable and a typing with a `read` mode suffices to type the statement in which ν occurs.
- ν stands for `destval(`ν`)` if ν is a `free` variable otherwise.

2. Empty correlations `<>` on base-modes m (in reference types and in correlations), and on `new` and `null` can be omitted. This reduces the bulkiness of all modes, making declarations and `new` expressions much easier to read. In particular, association modes α`<>`, mode `co<>`, and `null<>` can always be abbreviated to just α, `co`, and `null`.

3. Also Java's `void` methods can be supported as syntactic sugar: The result type `void` in the header of a method in a class c stands for the type `co<>` c. The body of `void` methods is extended by the return statement `return val(this);`. Calls $e.f$`(...);` of `void` methods on c objects (operation call statements) are expanded to assignment statements of desugared JaM as follows:

- Normally, they are expanded to $\texttt{dummy}_c$ = $e.f$(...), where $\texttt{dummy}_c$ is an implicit local variable of type `read<>` c.
- The case of a destructive read receiver expression $e \equiv \texttt{destval}(\nu)$ is treated specially to support the incremental modification of `free` objects by the call of `void` mutators through `free` references: Such calls are expanded to $\nu = e.f(\ldots)$ (equivalent to desugared $\nu = \nu.f(\ldots)$). That is, variable ν is reset for use as a call-link in a mutator call, but when the call is finished, ν gets back its old value back since the mutator returns `this`. This temporary reset can only be observed during the made mutator call if ν is a field. Hence if ν is a *local* variable, the reset can safely be optimized away.[2]

4. JaM's typing rules required an exact match between a method's declared result mode and its return expression's mode (whereas a compatible mode suffices in assignment and for parameters). The motivation was merely to simplify the formal treatment by not avoiding to record an executing method's declared result mode in order to automatically convert the return expression's value at the point of return. Nothing is lost by this computationally: If the return expression e has the wrong mode, simply treat the return statement "`return` e;" as syntactic sugar for "*result*=e; `return` *result*;" where *result* is an implicit variable whose range type is the method's (exact) result type.

5. JaM's formal runtime model made it necessary for the `main` method initially called on an instance of the program's last class c_n to be an observer: The start-up expression `new<>` c_n`().main()` is evaluated in an environment $\emptyset^{\mathsf{obs}}_{\langle\mathsf{nil},\mathsf{read<>},\mathsf{nil}\rangle}$ with receiver nil. Hence the initial `free` handle to the created c_n-object **o** has source nil, so that it does not show up in the object graph. But then there is no ownership path that would allow **o** to be executing a mutator (without violating the mutator control path property). This is no real restriction. A program p with last class c_n can always be extended to the program $p\ D_{n+1}$ with a new, implicit last class definition D_{n+1} = `class Main2{obs void main(){new` c_n`().main();}}`. Then an object of class `Main2` is created first. It may of course safely call a `main` mutator through its `free` initial reference to a new c_n-instance.

7.2.2 Subclass Polymorphism and Class Inheritance

Object-oriented programming would not be complete without *subclass polymorphism* and *class inheritance.* Java programming additionally relies heavily on *interfaces* (and partially abstract classes), and on *dynamic casts* of objects references (which are checked at runtime against the class of the target object). And the Java-extension *generic Java* allows generic classes that are parameterized by classes (reducing the

[2]This optimization is similar to that possible through Boyland's borrowed aliases of unique references [Boy01] (cf. §7.2.4). However, the `this` reference in the receiver is not borrowed: Aliases of it can be captured in fields and survive the return from the call; they only cannot be `free` or `rep`.

need for dynamic casts). All these features concern only the classes of the objects in the system. Since modes are orthogonal to classes, no problems are to be expected by extending JaM with any of these features: For the invariants about object ownership and the property of state encapsulation, which is based on them, the classes of the objects in the object graph is irrelevant. And in the proofs, the class of an object is only of interest in so far as it determines its methods' kind, result mode, and parameter modes.

Appendix B.3 shows the reformulation of the map example using interfaces, generic classes, and syntactic sugar. Types like `rep<data=elem> Node<T>` for the `anchor` reference in class `SetImp<T>` demonstrate that the orthogonality of reference mode and object class extends to the independent parameterization of (full) modes by correlations and of (generic) classes by classes.

Let us look more closely here at the issue of polymorphism and inheritance with concrete implementation classes (no `abstract class`, no `interfaces`):

1. Subtype Polymorphism. In a subtype-polymorphic programming language, a subtype relation $\leq_c$ is defined over types, and *dynamic types* can be subtypes of *static types*: The types of values in variables can be subtypes of the variables' nominal types; and the types of the values to which expression evaluate can be subtypes of the type inferred for that expression. For JaM's runtime model, this means that, on one hand, the notion of a type-consistent store is relaxed to allow for subtype values ("subtype-consistency"):

$$\models \mathfrak{s} \Leftrightarrow_{\mathrm{df}} \forall \tau \in \mathcal{M} \times \mathbb{C}, \ell \in \mathrm{dom}(\mathfrak{s}) \bullet\ \ell \in \mathcal{L}oc_\tau \Rightarrow \underline{\exists \tau' \leq_c \tau \bullet}\ \ \mathfrak{s}(\ell) \in [\![\tau']\!]$$

On the other hand, instead of reduction steps preserving the term's type exactly (Lemma 5), they may change it to a subtype (the *subject reduction* property):

$$\begin{array}{l}(e, \vec{\eta}, \mathfrak{s}, om, \mathfrak{g} \Longrightarrow e', \vec{\eta}', \mathfrak{s}', om', \mathfrak{g}') \wedge \Gamma, \kappa \vdash_X e : \tau \quad \wedge\ \vec{\eta} \models \tilde{\mu}, \Gamma, \kappa, X \quad \wedge \models \mathfrak{s}, om \\ \qquad \Rightarrow \ \exists\, X',\ \underline{\tau' \leq_c \tau\ \bullet}\quad \Gamma, \kappa \vdash_{X'} e' : \tau' \wedge\ \vec{\eta}' \models \tilde{\mu}, \Gamma, \kappa, X' \wedge \models \mathfrak{s}', om'\end{array}$$

Since dynamic types are allowed to be subtypes of static types, it is always safe in type inference to widen an expression's inferred type τ' to a supertype τ. That is, we can add a so-called *subsumption rule* to the typing rules:

$$\text{[sub]}\ \frac{\Gamma, \kappa \vdash e : \tau' \qquad \vdash \tau' \leq_c \tau}{\Gamma, \kappa \vdash e : \tau}$$

(Through subsumption for runtime terms, the type preservation property can be recovered from the subject reduction property since then $\exists X', \tau' \leq_c \tau \bullet \Gamma, \kappa \vdash_{X'} e' : \tau'$ entails $\exists X' \bullet \Gamma, \kappa \vdash_{X'} e' : \tau$.)

2. Subclass-based Subtyping. In Java, the polymorphic types, i.e., the types with non-trivial subtypes, are the so-called *reference types* [GJS00], i.e., the types of object reference values. (The types $\mathsf{ref}\,\tau$ of τ-variables, on the other hand, have no subtypes, even if τ is a reference type.) The subtypes $\tau' \leq_c \tau$ of a type τ of reference values with target class c are the types of reference values whose target class c' is a subclass of c. If we ignore `interfaces`, the subclass relation $\leq_c$ between classes is derived from the `extends` clauses of the program's `class` definitions, with `Object` as the implicit superclass above all others:

$$\frac{p \equiv D_1 \ \dots \ \texttt{class}\ c_i\ \texttt{extends}\ d\ \texttt{\{}\dots\texttt{\}}\ \dots\ D_n}{\vdash c_i \leq_c d}$$

$$\frac{\vdash c\ \mathbf{ok}}{\vdash c \leq_c c} \qquad \frac{\vdash c\ \mathbf{ok}}{\vdash c \leq_c \texttt{Object}} \qquad \frac{\vdash c \leq_c c' \quad \vdash c' \leq_c c''}{\vdash c \leq_c c''}$$

This is easily lifted to JaM's mode-qualified reference types: If c' is a subclass of c then the type $\tau' = \mu\ c'$ of μ-references to c'-instances is a subtype of the type $\tau = \mu\ c$ of μ-references to c-instances:

$$[\text{subty}]\ \frac{\vdash c \leq_c c'}{\vdash \mu\ c \leq_c \mu\ c'}$$

3. INHERITANCE. A subclass inherits member definitions from its base-class (direct superclass). This is achieved by the following special rule for the instance record type and method suite $\mathit{FldsMths}(c_i)$ of subclasses c_i:

$$\frac{\begin{array}{c} p \equiv D_1 \dots\ \texttt{class}\ c_i\ \texttt{extends}\ d\ \texttt{\{}\overline{t_i\ x_i}\texttt{;}\ \overline{\kappa_i\ t_i\ f_i\texttt{(}\pi_i\texttt{)\{}b_i\texttt{\}}}\texttt{\}}\ \dots\ D_n \\ \vdash \mathit{FldsMths}(d) = \langle \Gamma, F\rangle \end{array}}{\vdash \mathit{FldsMths}(c_i) = \langle \{\overline{x_i : \mathsf{ref}\ t_i}\} \cup \Gamma,\ \{\overline{f_i \mapsto \kappa_i\ t_i\ f_i\texttt{(}\pi_i\texttt{)\{}b_i\texttt{\}}}\} \cup F\,|_{Id\setminus\{\overline{f_i}\}}\rangle}$$

In Java, a field x cannot be declared again in a subclass, and the (re)implementation of an operation f in a subclass cannot change its parameters' and result's types, or changes only the result's type to a subtype. The following well-formedness rules for subclass definitions ensures that, additionally to well-formed and unique member definitions, the named base-class exists, if inherited fields are not overridden and it inherited methods are overridden only without changing method kind, result type, and parameter types:

$$[\text{subclass}]\ \frac{\begin{array}{l} \vdash M_1\ \mathbf{defs}\ x_1 \quad \cdots \quad \vdash M_n\ \mathbf{defs}\ x_n \\ \forall i,j = 1,\dots,n\bullet\ x_i = x_j\ \Rightarrow\ i = j \\ \mathit{FldsMths}(d) = \langle\Gamma, F\rangle \quad \mathrm{dom}(\Gamma) \cap \{x_1,\dots,x_n\} = \emptyset \\ x_i \in \mathrm{dom}(F) \Rightarrow M_i \quad = \kappa\ t\ x_i\texttt{(}\overline{t_i\ y_i}\texttt{)\{}\dots\texttt{\}} \\ \qquad\qquad\qquad \wedge\ \ F(x_i) = \kappa\ t\ x_i\texttt{(}\overline{t_i\ z_i}\texttt{)\{}\dots\texttt{\}} \end{array}}{\vdash \texttt{class}\ c\ \texttt{extends}\ d\texttt{\{}M_1\ \dots\ M_n\texttt{\}}\ \mathbf{defs}\ c}$$

4. TYPE PRESERVATION. For Java, it has been shown repeatedly that static type checking with the subsumption rule entails the subject reduction property [Sym97, DE97, Ohe01]. To convince oneself that the same holds for JaM, one can reread the proof for type preservation given on page 103 for the base-JaM version of the type preservation theorem: First, the relaxation to *sub*type-consistency of stores weakens old implications of the form $\ell \in \mathcal{L}oc_\tau \overset{\models_\mathfrak{s}}{\Longrightarrow} \mathfrak{s}(\ell) \in [\![\tau]\!]$ to implications $\ell \in \mathcal{L}oc_\tau \overset{\models_\mathfrak{s}}{\Longrightarrow} \exists\tau' \leq_\mathrm{c} \tau\centerdot\mathfrak{s}(\ell) \in [\![\tau']\!]$. For read access redices $\hat{e}$ reduced to $\mathfrak{s}(\ell)$, this entails $\Gamma, \kappa \vdash_X \mathfrak{s}(\ell) : \tau'$, which by subsumption can be widened to the original $\Gamma, \kappa \vdash_X \mathfrak{s}(\ell) : \tau$. For field access `this`.$x$, it means that the instance record ϱ' of the `this` object is consistent with the instance record type $\Gamma_{c'}$ of a subclass $c' \leq_\mathrm{c} c$ of the class c in `this`'s static type: $\varrho' \models \Gamma_{c'}$. The conclusion remains the same since the type of field x in $\Gamma_{c'}$ must be the same as in the superclass's instance record type $\Gamma_{c'}$: $\Gamma_c(x) = \tau = \Gamma_{c'}(x)$. Second, the possibility of subsumption weakens old implications of the form $\Gamma, \kappa \vdash_X \langle o, \mu, \omega\rangle : \mu\ c \Rightarrow \langle o, \mu, \omega\rangle \in [\![\mu\ c']\!]$ to implications $\Gamma, \kappa \vdash_X \langle o, \mu, \omega\rangle : \mu\ c \Rightarrow \exists c' \leq_\mathrm{c} c\centerdot\langle o, \mu, \omega\rangle \in [\![\mu\ c']\!]$. For return steps reduced to h, the correspondingly adapted conclusion is $\Gamma, \kappa \vdash_{X'} h : \mu^\star\circ\mu''\ c'$, which by subsumption can be widened to the original $\Gamma, \kappa \vdash_{X'} h : \mu^\star\circ\mu''\ c$. For operation call expressions $\langle \mathbf{s}, \hat{\mu}, \mathbf{r}\rangle$`.`$f$`(...)`, it means a weakening of Lemma 2 so that the target's method suite $F_\mathbf{r}$ and the method suite F_c of the receiver expression's target class c do not coincide any more. But since receiver $\mathbf{r}$ must be instance of a subclass of c, $F_\mathbf{r}$ and F_c can differ at f only in the irrelevant declarations π and λ of parameters and local variables: $F_\mathbf{r}(f) = \kappa^\star\ \mu\ d\, f\,(\pi)\,\{\lambda\ s\}$ while $F_\mathbf{r}(f) = \kappa^\star\ \mu\ d\, f\,(\pi')\,\{\lambda'\ s'\}$. This suffices for drawing the same old conclusion.

5. Integrity of the Higher-Level View. It is easy to verify that the extension by polymorphism and inheritance preserves properties Unique Owner and Unique Head: The typing of terms is relevant for the proof of Unique Ownerand Unique Head (see Lemma 23) only in two cases. (Note that all lemmas used in the proof are independent from the runtime term.) Assignment steps need just a compatibility of *modes*—which does not change through subclass polymorphism. And reasoning about the reduction of operation call expressions requires only that the sent handles' *mode* $\mu_{\mathbf{o}_i}'$ is compatible to the import $\mu_\mathbf{r}\circ\mu_{\mathbf{o}i}$ of the modes $\mu_{\mathbf{o}i}$ of the parameters in the receiver's method relative to the call-link's mode $\mu_\mathbf{r}$. This still holds even if the receiver expression $\langle \mathbf{s}, \mu_\mathbf{r}, \mathbf{r}\rangle$ was typed via the subsumption rule with a supertype $\mu_\mathbf{r}\ c$, so that receiver object $\mathbf{r}$ is an instance of a subclass $c' \leq_\mathrm{c} c$: The receiver's method $F_\mathbf{r}(f)$ has the same parameter modes $\mu_{\mathbf{o}i}$ as the superclass's method $F_c(f)$ used to calculate the parameter modes $\mu_\mathbf{r}\circ\mu_{\mathbf{o}i}$ in the receiver expression's signature $\Sigma(\mu_\mathbf{r}\ c)$ against which the sent handle's modes are checked.

Similarly it can be shown that properties Mutator Control Path and Mutator Control still hold since subclasses cannot change inherited methods' kinds. Finally, reasoning about coherence and shallow state encapsulation is completely independent of objects' classes. Hence composite state encapsulation follows as before.

To sum up, *the extension of JaM by subclass polymorphism and class inheritance preserves JaM's ownership and state encapsulation properties.*

7.2.3 Unlimited Calling?

The JaM type system, since it is purely static, excludes more message flows than necessary for mutator control (§7.1.3). This section considers some rather straight forward relaxations that will enable more invocations and reference exchanges.

1. Invocation Right $\perp$ Exchange Right. JaM's type system permits the invocation of any operation, including observers, only if it is able to determine the import of the exported operation's signature. Since there are constraints on the exchange of references, operations with certain parameters or result may not be invocable through certain references. This coupling can be eliminated since the question of legal invocation actually has nothing to do with the question of legal parameter and result exchange:

- Even if a parameter has a mode that cannot be imported, the invocation can be allowed under the condition that no actual object reference is supplied, but only a `null` reference. This is safe because `null` references do not show up in the object graph. This relaxation could be integrated into the mode system by importing parameter types with unimportable mode as Null, a special type assigned only to expressions '`null`<δ>'.
- If the result has a mode that cannot be imported, the invocation can be allowed if the result reference is discarded. This is safe because then no reference value needs not be exchanged in the first place. This relaxation could be integrated into the mode system by importing result types with unimportable mode as `void`, so that the call expression can be used only as a statement. (More precisely, it should be imported as the formal type Cmd, and `void` should be imported the same instead of treating it as syntactic sugar.)

2. Receiver-Side Conversion. Instead of enabeling the invocation by not exchanging problematic reference values, it would be more intelligent to convert the reference values from and to modes that are known to be safely exchangeable: One could assume implicit conversions of received parameter values to formal parameter modes, and of calculated result values to a mode that the call-link can return. This makes reference exchange possible in situations where it had to be prohibited before:

a) If through a μ_r-reference, μ-results may not be accessed, they could be automatically converted before return to a compatible mode $\mu' \geq_m \mu$ that allows the return. For example, an α<> result not returnable through a reference without α-correlation could be converted to a `read<>` reference and then returned to the sender as a `read<>` reference. And a result of mode `free`<β=m<>> not returnable through a `read` reference could be converted to `read`<β=m<>> to arrive at the sender as `read`<β=$\mu_r \circ m$<>>.
b) If through a μ_r-reference, μ'-parameters may not be accessed, one might make a supply to a parameter of mode $\mu \geq_m \mu'$ compatible to it, and then convert it

$$
\frac{\begin{array}{l}\vdash \mathit{FldsMths}(c) = \langle \Gamma, F\rangle \quad F(f) = \kappa\ \mu'\ d\, f\,\texttt{(}\overline{\mu'_i\ d_i\ y_i}\texttt{)\{...\}} \\ \vdash \mu' \leq \mu \quad \overline{\vdash \mu \leq \mu'_i} \quad \mu = \mathit{rescnv}(\mu', \mu_{\mathbf{r}}) \\ \forall i, \vec{\alpha}\bullet ((\mu_{\mathbf{r}}\circ\mu_i)(\vec{\alpha}) = \texttt{read} \Rightarrow \mu_i(\vec{\alpha}) = \texttt{read}) \wedge (\mu_i(\vec{\alpha}) \in \{\texttt{co}\} \cup \mathbb{A} \Rightarrow \mu_{\mathbf{r}}(\epsilon) \notin \{\texttt{read}\} \cup \mathbb{A}) \\ \forall \vec{\alpha}, \alpha\bullet\ \mu(\vec{\alpha}) = \texttt{free} \wedge \mu(\vec{\alpha}.\alpha) \in \mathbb{A} \wedge \mu_{\mathbf{r}}\circ\mu(\vec{\alpha}.\alpha) \neq \texttt{read} \Rightarrow \mu_{\mathbf{r}}(\epsilon) \neq \texttt{read}\end{array}}{\vdash (\, f : \overline{\mu_{\mathbf{r}}\circ\mu_i\ d_i} \xrightarrow{\kappa} \mu_{\mathbf{r}}\circ\mu\ d\,) \ \in\ \Sigma(\mu_{\mathbf{r}}\ c)}
$$

$$
\{\mathrm{ret}\}\ \frac{\begin{array}{l}\mu = \mathit{rescnv}(\mu', \mu_{\mathbf{r}}) \\ \mathfrak{s}' = \mathfrak{s}[\ell \mapsto \bot \mid \ell \in \mathrm{im}(\eta^\star)] \quad \mathfrak{g}' = \mathfrak{g} \oplus \mathbf{s} \xrightarrow{\mu_{\mathbf{r}}\circ\mu} \mathbf{o} \ominus \mathbf{s} \xrightarrow{\mu_{\mathbf{r}}} \mathbf{r} \ominus \mathbf{r} \xrightarrow{\mu'} \mathbf{o} \ominus \mathfrak{s}(\mathrm{im}(\eta^\star))\end{array}}{\ll\!\texttt{return}\ \langle \mathbf{r}, \mu', \mathbf{o}\rangle\texttt{;}\!\gg,\ \vec{\eta}\bullet\ {\eta^\star}^{\kappa^\star}_{\langle \mathbf{s}, \mu_{\mathbf{r}}, \mathbf{r}\rangle}, \mathfrak{s}, \mathit{om}, \mathfrak{g} \longrightarrow \langle \mathbf{s}, \mu_{\mathbf{r}}\circ\mu, \mathbf{o}\rangle, \vec{\eta}, \mathfrak{s}', \mathit{om}, \mathfrak{g}'}
$$

Figure 7.3: Adapted rules for receiver-side conversion

to μ'. For example, if a method has an inaccessible parameter of mode `rep<>`, the caller could supply a `free<>` reference as in a call with a `free<>` parameter, which is then converted in the receiver from `free<>` to `rep<>`.

With receiver-side conversion, the caller can always make a mutator control-conforming call by supplying `free` parameter values and importing the result as `read`.

In order to extend JaM by the receiver-side conversion of *parameters*, only the rule for handle signatures needs to be adapted. No change is needed in the semantic rule for operation calls, since actual parameter values are still converted to the method's declared parameter mode. The extension by receiver-side conversion of *results* would require a parallel adaption of the semantic rule for return steps and of the handle signature rule. In the return step, the return expression's value of mode μ' could be converted to any mode $\mu \geq_{\mathrm{m}} \mu'$ for which import $\mu_{\mathbf{r}}\circ\mu$ is defined. But in order to avoid non-determinism in the runtime conversion and ensure coincidence with the conversion assumed in type checking (a prerequisite for type preservation), we better use a mapping *rescnv* that determines for the method's real result mode μ' a unique mode $\mu = \mathit{rescnv}(\mu', \mu_{\mathbf{r}})$ to which μ' is compatible ($\mu' \leq_{\mathrm{m}} \mu$) and that is always importable as the result of operations in $\mu_{\mathbf{r}}$-handle signatures.

Figure 7.3 shows the adapted rules for handle signatures and for `return` steps.

$\mathit{rescnv}(\mu', \mu_{\mathbf{r}})$ should adapt μ' no more than necessary. In the examples above, this was the adaption α`<>` $\leq^1_{\mathrm{m}}$ `read<>` and `free<`β`=`m`<>>` $\leq^1_{\mathrm{m}}$ `rep<`β`=`m`<>>` $\leq^1_{\mathrm{m}}$ `read<`β`=`m`<>>`. A look at the handle signature rule and at the definition of $\leq_{\mathrm{m}}$ and proper modes $\mathcal{M}$, shows that circumventing the restrictions on importable result mode μ always involves the substitution of `read` or `rep` for other base-modes at certain positions in μ'. It is not difficult to convince oneself that such an adaption can be defined in a way that μ' is always compatible to $\mathit{rescnv}(\mu', \mu_{\mathbf{r}})$ and that $\mathit{rescnv}(\mu', \mu_{\mathbf{r}})$ passes the conditions on the result of operations imported into the signatures of $\mu_{\mathbf{r}}$-references. Limiting receiver-side conversion of results to conversion to $\mathit{rescnv}(\mu', \mu_{\mathbf{r}})$

and no further is no limitation for the sender: It is possible to show that the imported $rescnv(\mu', \mu_r)$ reference can be converted on the server-side to any mode $\mu_r \circ \mu''$ as which the server would have imported a reference that was further converted on the receiver-side to a mode $\mu'' \geq_m rescnv(\mu', \mu_r)$: $\mu_r \circ \mu'' \geq_m \mu_r \circ rescnv(\mu', \mu_r)$. This is consequence of the algebraic property of *monotonicity* of mode-import $\circ$ in the right hand side argument w.r.t. mode-compatibility $\leq_m$.

This is not a substantial change to the mode system since the references which are actually transported between sender and receiver are the same as could be transported before. Hence *the extension of JaM by implicit mode receiver-side conversion preserves JaM's ownership and state encapsulation properties.*

3. UNLIMITED SELF-CALLS. The proposed typing rules for simplicity treat self-calls like operation calls on a normal co-object. Consequently, self-calls are subjected to unnecessary restrictions: An object is not permitted to "exchange" `rep` references and association references with itself (since the `this` reference, like all `co` references, has no correlations). These restrictions are superfluous for self-calls, where sender and receiver coincide, since here *the object graph does not change at all* if modes are left unadapted. All references can be "exchanged" without limitations in self-calls. (Of course, a mutator invocation by self-call is still subject to the same constraint: not from within an observer.) Allowing calls through `this` without restrictions and without adaption is crucial for factoring out operations (even on references with `rep` and association modes) into separate methods, as one is used to. An example will be given further below.

Figure 7.4 shows additional rules that extend JaM by self-calls with implicit receiver `this` that are limited only through mutator control. The reduction step for these special operation call expressions is like that for a normal one with the receiver expression `val(this)`. The only difference is that it tags the inlined method body with an "*s*" in order to signify for the return step that the result value is not to be imported. (Controlling through a tag that result adaption happens only at the end of self-calls—and not more intelligently for all calls where receiver and sender coincide—is necessary to preserve the parallelism with type checking, and thus the type preservation property.) In correspondence to this special return, inlined method body of self-calls have their own typing rule that works without mode import.

4. REFACTORING EXAMPLE. For example, in the `MapImp` class in appendix B, the iteration over the `entryset` component in search for a given (potential) key object `k` is defined three times (in methods `Add`, `Remove`, and `lookup`). This search can be factored out into one separate (private) method `find_entry` that returns the reference to the desired entry pair with the mode `rep<fst=key, snd=value>` (or, in desugared JaM, `rep<fst=key<>, snd=value<>>`). This leads to a considerable simplification of the class. Special support for self-calls permits method `Add` to import the result as a `rep` reference, and not just a `read` reference, thus enabling it to update old entries with the given key to the new value. The restructured `MapImp` class is shown below

$$[\mathrm{call}_s]\ \frac{(\texttt{this} : \mathsf{ref}\ \texttt{co<>}\ c) \in \Gamma \quad \vdash \mathit{FldsMths}(c) = \langle R, F\rangle \quad F(f) = \kappa\ \tau\ f\,(\overline{\tau_i\ y_i})\{\ldots\} \quad \kappa^\star = \texttt{mut} \Rightarrow \texttt{co<>} \in \mathbf{Wr}(\kappa) \quad \overline{\Gamma, \kappa \vdash e_i : \tau_i'} \quad \overline{\vdash \tau_i' \leq_{\mathrm{m}} \tau_i}}{\Gamma, \kappa \vdash f\,(\overline{e_i}) : \tau}$$

$$\{\mathrm{call}_s\}\ \frac{\texttt{val}(\texttt{this}) \Leftarrow f\,(\overline{\langle \mathbf{s}, \mu_i'', \mathbf{o}_i\rangle}),\ \vec{\eta}, \mathfrak{s}, om, \mathfrak{g} \Longrightarrow^* \ll s \gg,\ \vec{\eta} \bullet \eta^{\star\kappa^\star}_{\langle \mathsf{s},\mathsf{co<>},\mathsf{s}\rangle}, \mathfrak{s}', om, \mathfrak{g}'}{f\,(\overline{\langle \mathbf{s}, \mu_i'', \mathbf{o}_i\rangle}),\ \vec{\eta}, \mathfrak{s}, om, \mathfrak{g} \longrightarrow\ \ll s \gg_s,\ \vec{\eta} \bullet \eta^{\star\kappa^\star}_{\langle \mathsf{s},\mathsf{co<>},\mathsf{s}\rangle}, \mathfrak{s}', om, \mathfrak{g}'}$$

$$\{\mathrm{ret}_s\}\ \frac{\mathfrak{s}' = \mathfrak{s}[\ell \mapsto \bot \mid \ell \in \mathrm{im}(\eta^\star)] \quad \mathfrak{g}' = \mathfrak{g} \oplus \mathbf{s} \xrightarrow{\mu} \mathbf{o} \ominus \mathbf{s} \xrightarrow{\mathsf{co<>}} \mathbf{s} \ominus \mathbf{s} \xrightarrow{\mu} \mathbf{o} \ominus \mathfrak{s}(\mathrm{im}(\eta^\star))}{\ll \texttt{return}\ \langle \mathbf{s}, \mu, \mathbf{o}\rangle ; \gg_s,\ \vec{\eta} \bullet \eta^{\star\kappa^\star}_{\langle \mathsf{s},\mathsf{co<>},\mathsf{s}\rangle}, \mathfrak{s}, om, \mathfrak{g} \longrightarrow \langle \mathbf{s}, \mu, \mathbf{o}\rangle, \vec{\eta}, \mathfrak{s}', om, \mathfrak{g}'}$$

$$\frac{\Gamma, \kappa \vdash_X\ s : \mu\ c}{\Gamma', \kappa' \vdash_{\tilde{\mu}', \Gamma, \kappa, X}\ \ll s \gg_s : \mu\ c}$$

Figure 7.4: Additional rules for unlimited self-calls

with syntactic sugar that hides the different variants of read access to variables. The complete, desugared version can be found in appendix B.

```
class MapImp {
   ...
   obs rep<fst=key, snd=value> Pair  find_entry(read Object k) ...

   mut co<> MapImp  Add(key Object k, value Object v)
   {  rep<fst=key, snd=value> Pair  p;

      // check for old entry with key k
      p = find_entry(k);  // <- self call
      // if there is none, create new entry and insert it
      if( p == null ) {  p = new<fst=key,snd=value> Pair();
                         this.entryset.Add(p);  }
      // set key and value of old/new entry
      p.Set(k,v);          // <- here we need the found reference to be rep
      return this;
   }
   ...
}
```

7.2.4 More Mutable Modes: Shared, Inside-Out, Borrowed

Several more cases of safe mutator calls not permitted by JaM can be identified, and have been identified in the related literature. Each of them can be supported through the addition of a corresponding mode. Let us briefly present them and sketch their integration into the mode system.

1. Globally Shared Top-Level Objects. In some object systems, there are particular "global" objects that are shared system-wide and do not belong to any composite object. For example, the `static` fields and methods of a class module c can be understood as making up a special "static" object o_c that is globally shared, and accessed through implicit references. Objects that are to be shared globally are often stored in `static` fields so that they are easy to access from every object. As top-level objects in the object composition hierarchy, state encapsulation allows them to receive mutator calls from any object and from any method.

It may sound surprising, but as far as state encapsulation is concerned, even methods labeled `obs` may invoke mutators on global objects because no owner's control of state representation changes can be violated. That is, the real meaning of *observer* methods is not "no side-effects" (anywhere in the system) but rather "no effects on the composite receiver" (and its co-objects).

The mode system can safely permit mutator calls to globally shared top-level object if the call-link's mode guarantees its target's top-level status. Such references go under the name "unprotected" [Hog91], `public` [KM95], `var` [NVP98], `norep` [CPN98], `plenary` [DLN98], or `shared` [ACN02]. `Shared` references can be exchanged freely between objects (if their mode contains only base-modes `shared` and `read`). They can also be stored as association references in generic container objects through references of modes `rep<elem=shared>`, `free<elem=shared>`, or `shared<elem=shared>`. The implicit reference to "static" objects o_c, through which `static` methods are called, would be treated as `shared<>` for checking the invocation of mutators and the exchange of references.

2. Inside-Out and Upward References. While an object ω is executing a mutator, any object o to whose state representation it belongs is guaranteed to be executing a mutator (mutator control). Hence while ω is executing a mutator, mutator calls from it to o and any of o's component objects are safe w.r.t. state encapsulation and mutator control: The call crosses sanctuary boundaries only inside-out, never outside-in.

The mode system is able to safely permit inside-out mutator calls if they are made from within a mutator and through a reference of a special mode guaranteeing that it is an *inside-out reference*: Targets of references of this mode are not enclosed in any sanctuary not also enclosing the source. A simple case of inside-out references are *upward references*, which a component object has to its owner; they are the inverses of `rep` paths. `Shared` references are another special case. Inside-out may include the limit case that source and target are in the *same* sanctuaries; then also `co` references are inside-out references. Inside-out references can be safely exchanged with co-objects, can flow forward as parameters along `rep` references and flow back as results along inside-out references. They can also be stored as association references in generic container objects through references of mode `rep<elem=insideout>`.

Inside-out references separated into different roles are supported by modes `var` α and `arg` α in [NVP98], and by context parameters α in [CPN98].

3. BORROWED WRITE ACCESS. In some object systems, there are particular, more or less widely shared objects providing a service of making changes to given objects. For example, `Mechanic` objects may provide a service to "repair" `Engine` objects (see §7.3.3 for the `Car` example with code): To get its `Engine` component repaired, a `Car` object may make use of a `Mechanic`'s `repair` service. This is safe w.r.t. state encapsulation if the `Car` calls `repair` from within a mutator and if the `Mechanic` does not preserve the reference to the `Engine` component when it is finished.

No object, not even an external one, and no method, not even an observer, can violate state encapsulation by calling a mutator through references that *exist* only while target ω's owner o is executing a mutator: Mutator control ensures that any object o to whose state representation ω belongs is currently executing a mutator.

The mode system is able to safely permit such mutator calls if a special mode `temp` guarantees the reference's life-time limitation. A `temp` reference to ω with such a property can safely be created from longer lasting references by the following conversions: *In a mutator*, a `rep` reference to ω is converted to `temp` (by ω's owner o), a `co` reference to ω is converted to `temp` (by ω or one of its co-objects), or an `insideout` reference to ω is converted to `temp`. A `temp` reference can safely be passed forwardly, as parameter, but neither be returned nor captured in fields in order to make sure the reference cannot last longer than the mutator. There is no problem w.r.t. state encapsulation if `temp` references are converted to `read` references, and these are returned and captureed. `Temp` references may also be stored as association references in generic container objects through references of mode `free<elem=temp>`, `rep<elem=temp>`, or `temp<elem=temp>`. It must only be ensured that such references are, like `temp` references, never returned nor captured. An example of containers of `temp` references in combination with refined method classification can be seen in §7.3.3.

`Temp` references are similar to the notion of *borrowing* in alias control: A borrowed reference is a temporary alias of a *unique* reference used in methods called by the method with the unique reference. The mode classifying references that are borrowed was called `unique` [Hog91], `uncaptured` [Ho$^+$92], `borrowed` [Boy01], or `lent` [ACN02]. The difference from `temp` references is that borrowed references can be created from a unique reference in observers and in mutators, and the intention of borrowing does not normally allow one to convert borrowed to non-borrowed references (e.g., `read` references) which are not prevented from being returned or captured in fields (thus causing a non-temporary violation of uniqueness).

NOT OWNERSHIP PATHS. A `shared` reference is obviously no ownership path, and it cannot be that it extends any path to an ownership path to its target, since the target is not owned. Also `insideout` and `temp` references cannot be ownership paths and cannot extend paths to ownership paths to their targets since different `insideout` references in the same object and different `temp` references in the same method execution can point to objects with different owners. (Given appropriate correlations, however, also `shared`, `insideout` and `temp` references might be *extensible* by association paths to ownership path.) This means that, when mutators are sent through `shared`, `insideout` or `temp` reference, they are *not sent through an ownership path.* Hence, in the extended mode system, the mutator control path property used in the formal treatment does not hold any more. But, as in the explanation of each new mode above, the property of mutator control is preserved, and this is the property which counts for the achievement of composite state encapsulation.

7.3 Some Applications and More Examples

7.3.1 Behavioral Type Checking

1. THE SPECIFICATION ORIGIN OF THE THESIS. The starting point for this dissertation were considerations about a programming language with *behavioral object types*, incorporating the notion that object types and the subtype relation on them ought to be defined in terms of the instances' external behavior ("*behavioral subtyping*") [Sny86, Ame87, LW94, DL97]. While detailed behavior specification would require something like Eiffel's runtime checked pre-/postconditions and invariants [Mey88] (augmented by history constraints [LW94]), two things should be possible to check (conservatively) already at compile time: Objects' *representation invariants* should not change between calls, so that their runtime check at the end of method calls makes a check at the beginning superfluous. Methods (operation implementations) should change nothing other than the part of the system state specified in their *frame conditions* coarsely in terms of composite objects (not in terms of single variables, which is the norm in specification techniques [Wil92, Lei95, Lea99, LN00]).

This static part of "behavioral type checking" is not an easy task if one realizes that object refinement is not data record refinement: The object specified by its external behavior, the *abstract object*, may be implemented by a composition of several implementation objects [Bre91, Wil92, Utt92, Lei95, MP99a]. Hence, first, the *abstract* map object's state is characterized by the simple invariant of uniquely associating each key in it with a unique value. One perfectly normal way of implementing maps is to represent their state in the fields of pair objects stored in an entry-set component. Consequently, the invariant about one abstract map object's state translates to an invariant about the states of several objects in its implementation (*representation invariant*). But how could a behavioral type checker exclude that the objects over which the representation invariant is expressed change between invocations of the maps' operations? Second, an *abstract* map object's `Add` operation has the simple frame condition of changing only the map itself. It is perfectly normal to implement `Add` by making changes to the object which is the entry-set component in the map's composite implementation. But how could a behavioral type checker know which objects—besides those from the abstract object's frame condition—the implementation is allowed to change?

Providing an answer for these questions with the help of a static type system was

```
interface Map {
   void          Add(key Object k, value Object v) mutates this;
   void          Remove(read Object k) mutates this;
   value Object  lookup(read Object k) depends this;
   ...
}
class MapImp implements Map {
   rep<elem=rep<fst=key, snd=value>> PSet  entryset;
   // => mutation of entryset and its elements subsumed under 'mutates this'
   ...
}
```

Figure 7.5: Map classes with `mutates` and `depends` clauses

the original motivation for the development of the mode system.

2. CHECKING FRAME CONDITIONS. The frame conditions for the operations of abstract object type `Map` could be specified by `mutates` clauses as shown in figure 7.5. (Note that writing `mutates` clauses makes annotations `mut` and `obs` superfluous.) For any object o implementing the `Map` type, the meaning of "`mutates this`" at operations `Add` and `Remove` is that these operations may change the state of *o as a composite object.* That is, the `mutates` clause expresses the frame condition that the caused changes are limited to the objects in $StRep(o)$ (at the beginning of the call). For the behavioral type checking of `Map`'s implementation classes is means that the implementions of `Add` and `Remove` are allowed to update `this`'s fields, and to send mutators to the targets of `rep` paths (and to `this`). Additionally, a method can always be allowed to mutate the `free` objects in it, since `free` objects are considered an implementation detail of the method similar to local variables.

3. CHECKING INVARIANTS FOR INDEPENDENCE FROM EXTERNAL STATE. The mode system guarantees the global property of *composite state encapsulation*, meaning that a composite object's state representation cannot change between invocations of o's operations. Hence to guarantee that an object's invariants should never become violated between invocations of its methods (independent of what the invariant says), one only has to ensure the obvious: Objects' *representation invariants* can be expressed only over their respective *state representation.* Based on the mode-classification of object references this can be ensured by a simple rule: A representation invariant is safe if it is expressed only by access to the fields of `this` and the states of the abstract objects reachable from it via `rep` paths. This rule is similar to Müller and Poetzsch-Heffter's rule that "[a]bstract values of [runtime] components must not depend on *states* of objects referenced read-only" [MP99a]. (Wills [Wil92] and Leino et al. [Lei95, DLN98, LN00] work the other way: They specify on which other objects an object's (composite) state depends, and derive from this which objects are state-representing components that require protection.)

For example, the `lookup` operation of an abstract `Map` object is a virtual attribute operation: Its result depends only on the map and on no other object. One could specify this, in analogy to a `mutates` clause, by a `depends` clause as shown in figure 7.5. For the behavioral type checking of `Map`'s implementation classes this means that `lookup`'s result should depend only the fields of `this` and on the results of virtual attribute operations of its `rep` path targets (and of `this`). However, enforcing this by permitting no other calls at all would be much too strict:

It must be possible to call external objects' *clean operations* [NVP98], i.e., observers whose result does not depend on any *changeable* state ("`depends nothing`"): For instance, the representation invariants of hash-tables and sorted lists depends on the hash-code or sorting criterion provide by its element objects. This attribute of the element objects should be constant; the virtual attribute operation calculating it should be clean.

Moreover, the implementation of `lookup` must be able to use a `free` iterator to get at the pair objects in its `entryset`, and an iterator's `current` operation depends not only on the iterator's state (the state of the iteration) but also on the state of the set. Declaring this dependency in the specification of the `Iterator` interface would require one to refer to the set over which it iterates, and in the specification of the `Set` interface the `elements` operation must be specified to return an `Iterator` iterating over `this`. It is not clear how the use and the checking of this information should best be integrated into a behavioral type checker using only the standard type system machinery. This could be a subject for further research.

7.3.2 Mode/Effects System and the Observer Pattern

The set of all (composite) objects reachable from an object o via association paths of mode α<> can be called o's α-association region (cf. §6.3). These sets of objects can be used as the regions of a coarse *effects system* that refines JaM's `mut`/`obs`-classification of methods by specifying the association regions whose objects the method may mutate. The syntax could be a variation of the previous section's `mutates` clauses, e.g., `mutates` α. Based on this refined classification, a combined mode/effects system can now additionally permit mutator calls through α-references in methods with `mutates` α. This makes it possible for the programmer to exploit *all* ownership paths, and not just base-JaM-style ones (cf. §7.1.3), for sending a mutator from the owner to the component object. In particular, container objects can now provide methods through which clients (with appropriate rights on the element objects) can make the container mutate its elements.

Let us demonstrate this in the context the Observer pattern [Ga+95], where Observer objects register with a Subject object o to be notified when it changes, so that they can synchronize their own state with the Subject's new state. An `Observer` is an object with a `Notify` operation to be called when the Subject, i.e., the observed object, changed state. It must be a mutator in order to allow the notified Observer to update its own state.

```
interface Observer {
   mut void  Notify();
}
```

Registries are specialized set objects useful for implementation of Subjects: In a Registry component, a Subject can not only keep the references to all its Observers; it can also notify them all at once through the Registry by calling operation notifyAll. This operation calls Notify on all elements through temporary elem references that it retrieved by reading the nodes' data references through a rep<data=elem> reference. The mutator call to an elem object is permitted in the mode/effects system since it is advertised as part of the method's effects-type by the clause "mutates elem."

```
class Registry extends SetImp<Observer> {
   obs void  notifyAll() mutates elem  // <- extension
   {  rep<data=elem> Node<Observer>  n;
      n = this.anchor;
      while( n != null )
      {  n.data().Notify();  // Permit this mutator call
                             // through returned elem reference
                             // because of 'mutates elem'
         n = n.next();
      }
   }
}
```

A Document is an example of a Subject whose state changes some Observers may want to follow. Documents are equipped with a Registry component to which Observers are added through operation Register. Documents call notifyAll on it whenever they changed (in a way relevant for Observers).

```
class Document {
   rep<elem=observer> Registry  reg;
   mut void  Register(observer Observer o)  { this.reg.Add(o); }
   mut void  SomeChange() mutates observer  // <- extension
   {
      ...  // some change
      this.reg.notifyAll(); // Permit call of 'mutates elem' method
                            // through a rep<elem=observer> reference
                            // since this is a 'mutates observer' method
   }
}
```

The called operation notifyAll may be an observer, but it also has "mutates elem" in its effects type. Hence the mode system has to check the call the same as checking a mutator call through a (temporary) reference returned from a Registry operation with result mode elem. Since elem references are imported as observer references through the rep<elem=observer> reference to the Registry, this means a check like a mutator call through an observer reference. As above, this is permitted *if* method SomeChange() advertises this by "mutates observer."

A View is an Observer for a Document. When notified of a change in the observed Document, the View will update its presentation of the Document.

```
class View implements Observer {
   read Document  doc;  // the document shown by the view
   mut void  Show(read Document d)  { this.doc = d; }
   mut void  Notify() { ... }  // update the presentation of doc
}
```

Finally, an Application object can link a Document and a View, so that the View shows the Document and the Document notifies the View of its changes.

```
class Application {
   rep<observer=rep> Document  doc;
   rep View                    view;
   mut void main()
   {  ...  // initialization
      this.view.Show(this.doc);
      this.doc.Register(this.view);
      ...
      this.doc.SomeChange(); // Permit call of 'mutates observer' method
                             // since, through reference 'doc', observer=rep
                             // and 'main' is a mutator
   }
}
```

The Application's call of SomeChange on the Document in a method main without mutates clause is subject to two conditions: First, SomeChange is a mutator. Hence the used reference doc must have mode free, rep, or co, which it has. Second, SomeChange is a "mutates observer" method. Hence the import of a returned observer reference must have mode free, rep, or co, which it has.

7.3.3 Domain Modeling: The Car Example

Object compositions is not only used in object-oriented design to implement higher-level software objects by lower-level ones, is it also used in object-oriented analysis to model the structure of real-world objects from the application domain. The example of a Car object was first introduced into the literature of composite object encapsulation by Clarke, Potter, and Nobel [CPN98], and varied in subsequent publications.

1. Car Objects have an Engine object as component [CPN98]. Figure 7.6 shows the JaM code for classes Car and Engine. By declaring Car objects' engine field as rep, the Engine component is protected in JaM as in Ownership Types [CPN98]

```
class Engine {
   rep Spark  spark;
   mut void  Start() { ... }
   mut void  Stop() { ... }
   mut void  ReplaceSpark(free Spark s) { this.spark = s; }
   obs int   exhaust() { ... }
}
class Car {
   rep Engine  engine;
   mut void       Init()  { this.engine = new Engine(); }
   obs rep Engine getEngine()  { return this.engine; }
   mut void       SetEngine(free Engine e)  { this.engine = e; }
   mut void       Go()  { ... this.engine.Start(); ... }
}
```

Figure 7.6: Classes Car and Engine

from being started or stopped other than through the Car. Unlike Ownership Types, JaM makes it possible for Cars to let outside objects read the Engine's exhaust level. Hence the exhausts of two cars can be compared without the need for an expose construct, "friendly functions", or parametric "ownership polymorphism" of methods proposed by Clarke [Cla01].

```
rep Car car = new Car();
car.go();
car.getEngine().Stop();              // error in Jam and OT

rep Car car2 = new Car();
compare(car.getEngine().exhaust(),   // ok in Jam, error in OT
        car2.getEngine().exhaust());
```

Also it is possible to let outside objects supply a new engine component for the car, provided it is newly created: As in Ownership Types, it is not possible to put the Engine from one Car into another Car or give the same Engine to two Cars.

```
free Engine e = new Engine();
car.setEngine(e);                   // ok in Jam, cannot do in OT
car2.setEngine(e);                  // e is null (violates use once convention)
car.setEngine( car2.getEngine() ); // error in Jam and OT
```

2. Engine Repair. An external object, like a Mechanic, that needs *write* access to the Engine component (e.g., for replacing the spark) cannot be handled by the presented mode system. It requires an extension like the mode temp described in §7.2.4. The resulting code can be seen in figure 7.7.

The Car needs no special access to the Mechanic (assuming the Mechanic does not change by repairing an Engine). Hence the Mechanic object given to the Car for repairing the Engine can be anywhere in the object system, e.g., a rep component

```
class Mechanic { ...
   obs void  repair(temp Engine e) { ... e.ReplaceSpark(s); ... }
}
class Car { ...
   mut void  GetEngineRepairedBy(read Mechanic m) { m.repair(this.engine); }
}
class Garage {
   rep Mechanic  bill, bob;
   obs void  repair(temp Car c) { ... c.GetEngineRepairedBy(this.bob); ... }
}
```

Figure 7.7: Repairing a car's engine

of a `Garage` object. The `Garage` object needs `temp` access to the `Car` first, so that it can tell it to give one of its `Mechanics` access to its `Engine`. The `Car` and the `Garage` can be brought together by any method that has write access to the `Car` and some reference the `Garage`.

```
rep Car     car;
rep Garage  garage;
... // initialization
garage.repair(car);  // 'rep' converted to 'temp'
```

3. THE ACIDBATH. Figure 7.8 shows an extension of the engine repair example by an acid-bath. It varies Clarke's example of storing a temporary *car* reference in a temporary `Acidbath` object during the execution of an ownership polymorphic method [Cla01]: While the `Mechanic` is not permitted to capture its `temp` reference to the `Engine` in its own field, it could store them as an association reference in a temporary `Acidbath` component. If we additionally have the `mutates` extension from §7.3.2, then the `Mechanic` is able to change the `Engine` through the `Acidbath` object.

7.3.4 Transfer Across Abstraction Boundaries: The Lexer/ Reader Example

In [DLN98], Detlef, Leino and Nelson presented the example of a lexer abstraction built on top of a reader abstraction: The reader produces a stream of characters (e.g., from a file). The lexer produces a stream of token from the characters from a reader component. The reader to be used by the lexer should not be fixed. The client should be able to configure the lexer with the reader whose output it wants to be tokenized. For example, a lexer for the password file could be constructed using a `FileReader` object (see figure 7.9).

The reader is a state-representing component of the lexer using it: The validity of the lexer's current state is dependent on the validity of the reader in the lexer's `rdr` field ("*dynamic dependency*") [DLN98]. Detlef, Leino and Nelson ensured the

```
class Acidbath {
   tobathe Engine e;
   mut void  PrepareFor(tobathe Engine e) { this.e = e; ... }
   obs void  bathe() mutates tobathe { ... }
   obs void  clean() mutates tobathe { ... }
}
class Mechanic { ...
   obs void  repair(temp Engine e)
   {  free<tobathe=temp> Acidbath  b;
      b = new<tobathe=temp> Acidbath();
      b.PrepareFor(e);
      b.bathe();
      b.clean();
      ...
   }
}
```

Figure 7.8: Acid-bathing a car's engine

soundness of passing the reader across the lexer's abstraction boundary by declaring the dependency on the reader in the lexer's interface. In JaM, we declare field `rdr` to have mode `rep`. Then the mode system ensures that only `rep` and `free` values can be assigned to it. The references which the client (here, method `main`) passes to the lexer for initialization can only be `free`. By supplying the `free` reader reference, the client gives up its only writable reference to the reader. Hence the client is unable in the following to invalidate the lexer by manipulating the reader, e.g., by closing the reader from under the lexer.

Since the lexer did not initialize the reader, it might not know how to shut it down at the end. Only the client should know. However, once the reader has become part of the lexer's state representation, the lexer cannot detach it again and return it to the client for mutation (the methodology in [DLN98] has the same limitation): There is no way back from a `rep` reference to a `free` reference since the lexer may have stored aliases of it in itself or as `co`- or association references in other objects.

But the lexer's `rdr` field can be declared `free`. Then the lexer cannot create non-`read` aliases of the given reader reference, but can return it via a method

```
mut free Reader  GiveBack() { return destval(this.rdr); }
```

Observe that effectively also a `free` reader represents an aspect of the lexer's state. Although the mode system does not explicitly take this into account, state encapsulation also holds here (cf. §6.5): Still only the lexer can send mutators to the reader, and it can send them only from its own mutators since that requires a destructive read of the field (which may be hidden behind syntactic sugar, §7.2.1).

```
class Lexer {
   rep Reader  rdr;

   mut co Lexer  Init(free Reader r) { this.rdr = r; return this; }
   mut free Token  GetToken() { ... }  // using rdr
}
class Main {
   mut read Object  main()
   {  free Reader  rd;
      rep Lexer    lx;
      ...
      rd = new FileReader().Open("/etc/passwd");
      lx = new Lexer().Init(rd);
      ...
   }
}
```

Figure 7.9: The lexer/reader example

7.3.5 Transfer of Multiple Objects at Once

The transfers across abstraction boundaries considered in the literature transfer only one composite object at a time [DLN98, GB99, ACN02]. JaM supports the natural generalization of this idea: Entire linked lists of `free` composite objects can be transfered by one reference exchange, and—by allowing correlations to `free` modes—`free` objects can be stored in an ordinary (`free`) container object and transfered all together by passing that container. The limitation is only that such aggregated `free` objects will all be added at the same time to the same state representation by a corresponding mode conversion.

Consider the `AddAll` generalization of the `Add` operation on `Sets`. It adds an arbitrary number of new elements, i.e., logically, it has a variable number of parameters. In JaM, this can be implemented by storing the new element objects $e_1, \ldots, e_k$ in a linked list passed to `AddAll`:

```
class Set<T> {
   ...
   mut void  AddAll(read<data=elem> Node<T>  list);
}
```

A simple implementation of this operation could traverse the list to extract each new element e_k and add it by a self-call `Add`(e_k). (Here one needs the real self-calls of §7.2.3 in order to be able to pass the `elem` reference obtained from the node and to pass the `read<data=elem>` to the rest of the list.) Observe that class `SetImp` represents the abstract set's state in an internal list with the same type of nodes as those passed to `AddAll`. But it would not generally be safe to simply concatenate the passed list to the internal list since the passed list might still be "in use", e.g., as the

```
class Set<T> { ...
   mut void  AddAll(free<data=elem> Node<T>  list);
}
class SetImp<T> implements Set<T> { ...
   mut void  AddAll(free<data=elem> Node<T>  list)
   {  if( this.anchor != null )  { this.anchor.ListAppend(list); }
      if( this.anchor == null )  { this.anchor = list; }
   }
}
class Node<T> { ...
   mut void  ListAppend(co Node<T> list)
   {  if( this.next != null )  { this.next.ListAppend(list); }
      if( this.next == null )  { this.next = list; }
   }
}
```

Figure 7.10: AddAll with transfer of linked nodes

internal list of another `SetImp` set. JaM catches this since a passed `read` list (more precisely, a list of `read` nodes) cannot be concatenated with an internal `rep` list.

The mode system allows concatenation only if the passed list is `free`, or `rep`. By changing the parameter's mode to `free`, the set object can oblige its clients to supply lists that are guaranteed not belong to any object's state representation, so that it can safely append them to its own internal list. This way, all new elements in the list are added in one step, without creating new node objects and copying the new elements into them. The sugared JaM code of `AddAll` and the used `ListAppend` method of `Nodes` is shown in figure 7.10. (For simplicity, no check is made here whether the elements in the given list already exist in the set.)

7.3.6 The Builder Pattern: Bottom-Up Creation with Free Fields

Up to this point, only uncaptured `free` references were considered , i.e., `free` references as parameters, in local variables, or as temporary values. `Free` *fields* enable JaM to support the Director, or Builder, design pattern [Ga+95]: Through a Builder object, a client can control the incremental construction of a complex object, the Product, without knowing the Product's implementation. For example, an application window object (with text area, menu bar, scroll bars, etc.) should be constructed through a Builder, so that by switching the Builder object, windows in different GUI-frameworks (AWT, Swing, Windows, Motif, Athena, ...) can be constructed.

The JaM code in figure 7.11 sketches the definition of a `WindowBuilder` interface, and one implementation (for Motif windows, based on a library of classes with names `Xm...`). Scroll bars, tabs, status bars, and tool bars can be added repeatedly to the left, right, top, or bottom of what was constructed so far (e.g., vertical and

```
interface WindowBuilder {     // integer codes for addwhere and colors
   mut void CreateTextArea    (int bgcol, int fgcol);
   mut void CreateDrawingArea(int bgcol, int fgcol);
   ...
   mut void AddMenus (int bgcol, int fgcol, read<data=read> List<String> titles);
   mut void AddTools (int bgcol, read<data=read> List<Bitmap> buttons);
   mut void AddScroll(int addwhere, int bgcol, int fgcol);
   ...
   mut free Window GetWindow();
}
class MotifBuilder implements WindowBuilder {
   free Widget  top;
   mut void CreateTextArea(int bgcol, int bgcol)
   {  this.top = new XmTextArea(bgcol,fgcol);  }
   ...
   mut void AddMenus(int bgcol, int fgcol, read<data=read> List<String> titles)
   {  free XmMenuBar m = new XmMenuBar(bgcol,fgcol);
      free XmForm    f = new XmForm();
      while(titles!=null) { m.AddEntry(new XmString(titles.data()));
                            titles = title.next(); }
      f.arrange(m, XmForm.ABOVE, this.top);  // N.B. destructive reads
      this.top = f;                          // new top
   }
   ...
   mut free Window GetWindow() { return new XmAppWindow(this.top); }
}
```

Figure 7.11: Sketch of a WindowBuilder

horizontal scrollbars). The Motif builder keeps a `free` reference to the largest widget composed so far in field `top`. In each addition step, the current `top` widget and the new widget are transfered into an `XmForm` widget that combines them and fixes their relative geometrical placement (horizontally left/right, or vertically one above the other). This `XmForm` widget is the new `top` widget. When construction is complete, the Builder wraps the latest `top` widget in an `XmAppWindow` and returns that.

This is a good example of an object construction process in which a complex composite object (the window) is created *bottom-up*, i.e., in which sub-objects are created before their owners. Bottom-up construction cannot be handled by the ownership type systems of [CPN98, MP99a, MP01, Cla01]: There an object's owner has to be fixed when it is created. That is, composite objects can only be constructed *top-down*. To handle the lexer/reader example (§7.3.4), Clarke describes how a Factory object *readerClass* with an ownership polymorphic creation method allows one to delay a prospective component's construction until its prospective owner requests it. It may be possible to extend this workaround to a reversal of the entire construction process of the window. The Builder would then not produce a window but a factory constructing the window in top-down fashion, a solution which is far less elegant.

Chapter 8

Conclusion

The encapsulation of composite objects is an important criterion for the quality of object-oriented designs: Composite objects are the nested mid-scale components of the runtime system. The recursive combination of smaller objects to one composite object (object composition), is a central technique in the construction of object-oriented software. Structuring the runtime system into hierarchies of composite objects demonstrably helps managing the structural and dynamic complexity of the object system. A lack of encapsulation makes the composite object's correct functioning depend on its context, so that its implementation cannot be verified in a modular way and cannot safely be reused in new contexts (without rechecking it).

This dissertation provided (a) a formal definition of the desired property of *state encapsulation*, (b) *mode qualifiers* on all object reference types to express object composition and identify composites' state-representing components, and (c) *static mode checks* to ensure that composite objects change state only through operations declared 'mutator' (state encapsulation). The mode checks are a purely static, orthogonal extension of standard typing rules. They imply no changes in the program execution and do not limit the range of possible computations. But they make sure that the new mode and 'mutator' annotations are only added in ways that are consistent with the structure of object composition and with state encapsulation.

The composition of objects was defined based on a mode-classification of paths of object references in the evolving object graph. The classification is inductively derived from the object references' modes, which is imposed on them through the mode annotations qualifying the types of object reference-valued fields, variables, parameters and results. The path-based approach supports better than others the flexible, dynamic creation and incremental construction of complex composite objects: It enables one to decouple object creation and object use (in particular, use as a composite's component) with no significant restriction on composite objects' internal structure. Composite objects can be constructed bottom-up, and can be transfered across abstraction boundaries (one by one, linked to lists, or stored in containers).

As a proof of concept, a subset of the Java language was extended by mode and mutator annotations, and by mode checks to the language JaM. To ensure the consistency of the object graph's mode-labeling, the mode checks restrict the compatibility between different modes when object references are assigned, supplied as parameter, or returned as result. On this basis, the actual enforcement of state encapsulation consists of limiting, depending on the kind of method, whether fields may be assigned to, and whether mutators may be invoked through references of certain modes. Support for the decoupling of object creation and object use is based on a weak uniqueness property for reference path classified as `free`. Its preservation is enforced by allowing only destructive read access to variables holding `free` references or a non-destructive access that creates a `read`-moded alias. (Destructive read could be replaced by the check that the variable is "dead" after the read access.)

It was shown—first for a simplified mode and then for the full mode system—that the extended typing rules guarantee state encapsulation (relative to the mode-specified object composition structure) in a purely static way; no runtime checks are necessary. The addition of association modes and correlations in the full mode system was crucial to enable the path-based handling of recursively composed composite objects. But while the addition was easy to define, the complexity of the formal treatment increased more than expected, despite several simplifying constraints.

The usability of the proposed mode system was demonstrated with the non-trivial map example. It covers recursively composed composite (container) objects, the construction of a composite (iterator) object with an externally created (iterator) component object, and the Iterator and Abstract Factory design patterns. Furthermore, it was shown how JaM handles examples from the composite object encapsulation literature (modeling the car domain, transfer across abstraction boundaries), and how it supports the Builder design pattern.

The global system properties guaranteed by the mode system can serve as a basis for other work, like behavioral type systems, the modular verification of object behavior against specifications, reasoning about aliasing and interference, and object-oriented effects systems. While the presented mode system is more restrictive than desirable, it provides a sound stable basis that can be developed further. In particular, the constraints made for the formal treatment could be relaxed. A nicer syntax for specifying the references' modes could be found (including the possibility of mode inference). Variations and extensions of the mode system could be investigated. Specialized modes could express subclassifications of object references w.r.t. different levels of component encapsulation (public, read-only private, inaccessible private), or w.r.t. more detailed aliasing properties and access rights than required for state encapsulation. Or they could make finer distinctions that enable us to safely permit more cases of mutator calls (modes `shared`, `insideout`, `borrowed`, etc.), or the migration also of *state-representing* components from one composite object to another.

Appendix A

The Definition of JaM

A.1 Syntactic Structures

1. JaM Programs — extension of Java subset (changes underlined)

$$
\begin{array}{rrl}
p \in & \mathrm{P} ::= & \mathrm{D}^* \\
D \in & \mathrm{D} ::= & \texttt{class}\ \mathbb{C}\ \ \{\ (\mathrm{T}\ \mathit{Id};)^*\ \mathrm{Mth}^*\ \} \\
 & \mathrm{Mth} ::= & \underline{\mathcal{K}}\ \mathrm{T}\ \mathit{Id}((\mathrm{T}\ \mathit{Id})^*)\ \{(\mathrm{T}\ \mathit{Id};)^*\ \mathrm{S}\} \\
\kappa \in & \underline{\mathcal{K}} ::= & \underline{\texttt{mut} \mid \texttt{obs}} \\
t \in & \mathrm{T} ::= & \underline{\mathrm{M}}\ \mathbb{C} \\
\mu \in & \underline{\mathrm{M}} ::= & \underline{\mathrm{B}\texttt{<}\Delta\texttt{>}} \\
m \in & \underline{\mathrm{B}} ::= & \underline{\texttt{free} \mid \texttt{rep} \mid \texttt{co} \mid \mathbb{A} \mid \texttt{read}} \\
\delta \in & \underline{\Delta} ::= & \underline{(\mathbb{A}\texttt{=}\mathrm{M})^*} \\
s \in & \mathrm{S} ::= & \mathrm{S\ S} \mid \mathrm{N}\texttt{=}\mathrm{E}; \mid \texttt{return}\ \mathrm{E}; \mid \texttt{if}(\mathrm{E}\,\Psi\,\mathrm{E})\{\mathrm{S}\} \mid \texttt{while}(\mathrm{E}\,\Psi\,\mathrm{E})\{\mathrm{S}\} \\
\psi \in & \Psi ::= & \texttt{==} \mid \texttt{!=} \\
e \in & \mathrm{E} ::= & \underline{\texttt{val}}(\mathrm{N}) \mid \underline{\texttt{destval}(\mathrm{N})} \mid \texttt{null}\underline{\texttt{<}\Delta\texttt{>}} \mid \texttt{new}\underline{\texttt{<}\Delta\texttt{>}}\ \mathbb{C}\texttt{()} \mid \mathrm{E}\underline{\Leftarrow}\mathit{Id}(\mathrm{E}^*) \\
\nu \in & \mathrm{N} ::= & \mathit{Id} \mid \texttt{this}.\mathit{Id}
\end{array}
$$

Given identifier sets:

- classes	$c, d \in \mathbb{C}$	
- association roles	$\alpha, \beta, \gamma \in \mathbb{A}$	(excludes `free`, `rep`, `co`, `read`)
- variables, fields, methods	$x, y, z, f \in \mathit{Id}$	(includes `this`, excludes `null`)

2. Runtime Terms — extension of program's statements and expressions

$$
\begin{array}{rll}
R ::= & R\ \mathrm{S} \mid R\texttt{=}R; \mid \texttt{return}\ R; & \\
\mid & \texttt{if}(R\,\Psi\,R)\{\mathrm{S}\} \mid \texttt{while}(\mathrm{E}\,\Psi\,\mathrm{E})\{\mathrm{S}\} & \\
\mid & \mathrm{N} \mid \texttt{val}(R) \mid \texttt{destval}(R) \mid \texttt{null<}\Delta\texttt{>} \mid \texttt{new<}\Delta\texttt{>}\ \mathbb{C}\texttt{()} \mid R \Leftarrow \mathit{Id}(R^*) & \\
\mid & \mathcal{L}oc & \textit{location of a variable (l-value)} \\
\mid & \mathcal{V} & \textit{expression value (r-value)} \\
\mid & \ll R \gg & \textit{inlined executing method}
\end{array}
$$

3. FORMAL TYPE TERMS — extension of program's type terms for type checking and semantic consistency

$\tau \in$	$\mathcal{T} ::= \mathsf{ref}\,\mathcal{M}\;\mathbb{C}$	*l-values (locations)*
	$\mid\ \mathcal{M}\;\mathbb{C}$	*values (handles)*
	$\mid\ \mathsf{obj}\;\mathbb{C}$	*object values*
	$\mid\ \mathsf{Cmd}$	*continueing statements*

A.2 Type System

All definitions are relative to a given program $p \in P$.

4. VALID MODE, RANGE TYPE, CLASS NAME $\boxed{\vdash \mu\ \mathbf{ok}}$ $\boxed{\vdash t\ \mathbf{ok}}$ $\boxed{\vdash c\ \mathbf{ok}}$

$$\text{[mode]}\ \frac{\begin{array}{l}\forall i,j \in \{1,\dots,n\}\bullet\ \alpha_i = \alpha_j \Rightarrow \mu_i \equiv \mu_j \\ m \in \{\mathtt{co}\} \cup \mathbb{A} \Rightarrow n = 0 \\ \forall i \in \{1,\dots,n\}\bullet\ \mu_i \neq \mathtt{free}\texttt{<...>} \wedge \mu_i \neq \mathtt{co}\texttt{<>}\ \wedge \vdash \mu_i\ \mathbf{ok}\end{array}}{\vdash m\texttt{<}\alpha_1\texttt{=}\mu_1,\dots,\alpha_n\texttt{=}\mu_n\texttt{>}\ \mathbf{ok}}$$

$$\text{[rtype]}\ \frac{\vdash \mu\ \mathbf{ok} \qquad \vdash c\ \mathbf{ok}}{\vdash \mu\ c\ \mathbf{ok}} \qquad\qquad \text{[cname]}\ \frac{p \equiv D_1 \dots\ \mathtt{class}\, c\,\texttt{\{...\}}\ \dots D_n}{\vdash c\ \mathbf{ok}}$$

where $\mu \equiv \mu' \Leftrightarrow_{\mathrm{df}} \forall \vec{\alpha} \in \mathbb{A}^*\bullet\ \mu(\vec{\alpha}) = \mu'(\vec{\alpha})$
where $\mu(\epsilon) \quad =_{\mathrm{df}} m \qquad\qquad \text{if } \mu = m\texttt{<...>}$

$$\mu(\alpha.\vec{\alpha}) =_{\mathrm{df}} \begin{cases}\mu'(\vec{\alpha}) & \text{if } \mu = m\texttt{<}\dots,\alpha\texttt{=}\mu',\dots\texttt{>} \\ \bot & \text{otherwise}\end{cases}$$

5. MODE-COMPATIBILITY JUDGEMENT $\boxed{\vdash \tau \leq_{\mathrm{m}} \tau'}$

$$\frac{\mu \leq_{\mathrm{m}} \mu'}{\vdash \mu\ c \leq_{\mathrm{m}} \mu'\ c} \qquad \begin{array}{lll} m\texttt{<}\delta\texttt{>} & \leq^1_{\mathrm{m}} \mathtt{read}\texttt{<}\delta\texttt{>} & \\ \mathtt{free}\texttt{<}\delta\texttt{>} & \leq^1_{\mathrm{m}} \mathtt{rep}\texttt{<}\delta\texttt{>} & \\ \mathtt{read}\texttt{<}\delta,\alpha\texttt{=}\mu,\delta'\texttt{>} & \leq^1_{\mathrm{m}} \mathtt{read}\texttt{<}\delta,\delta'\texttt{>} & \\ \mathtt{read}\texttt{<}\delta,\alpha\texttt{=}\mu,\delta'\texttt{>} & \leq^1_{\mathrm{m}} \mathtt{read}\texttt{<}\delta,\alpha\texttt{=}\mu',\delta'\texttt{>} & \text{if } \mu \leq^1_{\mathrm{m}} \mu' \end{array}$$

6. WHAT CLASS MODULES DEFINE $\boxed{\vdash \mathit{FldsMths}(c) = \langle \Gamma, F\rangle}$

$$\frac{p \equiv D_1 \dots\ \mathtt{class}\ c_i\ \texttt{\{}\overline{t_i\ x_i}\texttt{;}\ \overline{\kappa_i\ t_i\ f_i\texttt{(}\pi_i\texttt{)\{}b_i\texttt{\}}}\texttt{\}}\ \dots\ D_n}{\vdash \mathit{FldsMths}(c_i) = \langle \{\overline{x_i : \mathsf{ref}\, t_i}\}, \{\overline{f_i \mapsto \kappa_i\ t_i\ f_i\texttt{(}\pi_i\texttt{)\{}b_i\texttt{\}}}\}\rangle}$$

7. HANDLE SIGNATURE JUDGEMENTS $\boxed{\vdash (\, f : \overline{\tau_i} \xrightarrow{\kappa} \tau\,) \in \Sigma(\mu\ c)}$

$$\frac{\begin{array}{l}\vdash \mathit{FldsMths}(c) = \langle \Gamma, F\rangle \quad F(f) = \kappa\ \mu\ d\, f\,(\overline{\mu_i\ d_i\ y_i})\texttt{\{...\}} \\ \forall i, \vec{\alpha}\bullet\ ((\mu_{\mathbf{r}}\circ\mu_i)(\vec{\alpha}) = \texttt{read} \Rightarrow \mu_i(\vec{\alpha}) = \texttt{read}) \wedge (\mu_i(\vec{\alpha}) \in \{\texttt{co}\} \cup \mathbb{A} \Rightarrow \mu_{\mathbf{r}}(\epsilon) \notin \{\texttt{read}\} \cup \mathbb{A}) \\ \forall \vec{\alpha}, \alpha\bullet\ \mu(\vec{\alpha}) = \texttt{free} \wedge \mu(\vec{\alpha}.\alpha) \in \mathbb{A} \wedge \mu_{\mathbf{r}}\circ\mu(\vec{\alpha}.\alpha) \neq \texttt{read} \Rightarrow \mu_{\mathbf{r}}(\epsilon) \neq \texttt{read}\end{array}}{\vdash (\ f : \overline{\mu_{\mathbf{r}}\circ\mu_i\ d_i} \xrightarrow{\kappa} \mu_{\mathbf{r}}\circ\mu\ d\ ) \ \in\ \Sigma(\mu_{\mathbf{r}}\ c)}$$

where
$$\mu(\epsilon) =_{\mathrm{df}} m \quad \text{if } \mu = m\texttt{<...>}$$
$$\mu(\alpha.\vec{\alpha}) =_{\mathrm{df}} \begin{cases}\mu'(\vec{\alpha}) & \text{if } \mu = m\texttt{<}\ldots,\alpha\texttt{=}\mu',\ldots\texttt{>} \\ \perp & \text{otherwise}\end{cases}$$

and
$$\begin{array}{ll} \mu_{\mathbf{r}}\circ\texttt{read<}\overline{\alpha_i\texttt{=}\mu_i}\texttt{>} & =_{\mathrm{df}} \texttt{read<}\overline{\alpha_i\texttt{=}\mu_{\mathbf{r}}\circ\mu_i}\texttt{>} \\ \mu_{\mathbf{r}}\circ\texttt{free<}\overline{\alpha_i\texttt{=}\mu_i}\texttt{>} & =_{\mathrm{df}} \texttt{free<}\overline{\alpha_i\texttt{=}\mu_{\mathbf{r}}\circ\mu_i}\texttt{>} \\ \mu_{\mathbf{r}}\circ\texttt{rep<}\overline{\alpha_i\texttt{=}\mu_i}\texttt{>} & =_{\mathrm{df}} \texttt{read<}\overline{\alpha_i\texttt{=}\mu_{\mathbf{r}}\circ\mu_i}\texttt{>} \\ \mu_{\mathbf{r}}\circ\texttt{co<>} & =_{\mathrm{df}} \mu_{\mathbf{r}} \\ \mu_{\mathbf{r}}\circ\alpha\texttt{<>} & =_{\mathrm{df}} \mu' \quad \text{if } \mu_{\mathbf{r}} = m_{\mathbf{r}}\texttt{<}\ldots,\alpha\texttt{=}\mu',\ldots\texttt{>}\end{array}$$

8. WELLFORMED PROGRAM, DEFINITION, TYPE ASSIGNMENT $\boxed{\vdash p\ \ \textbf{start}\ e_0}$, $\boxed{\vdash D\ \textbf{defs}\ x}$, $\boxed{\vdash \Gamma\ \textbf{ok}}$

$$[\text{prog}]\ \frac{\begin{array}{l}\vdash D_1\ \textbf{defs}\ c_1 \quad \cdots \quad \vdash D_n\ \textbf{defs}\ c_n \qquad \forall i,j = 1,\ldots,n\bullet\ \ c_i = c_j \ \Rightarrow\ i = j \\ \vdash \mathit{FldsMths}(c_n) = \langle R, F\rangle,\ \ F(\texttt{main}) \doteq \texttt{obs}\ \tau\ \texttt{main()\{...\}}\end{array}}{\vdash D_1\ \ldots\ D_n\ \textbf{start}\ \texttt{new<>}\ c_n\texttt{().main()}}$$

$$[\text{class}]\ \frac{\vdash M_1\ \textbf{defs}\ x_1 \quad \cdots \quad \vdash M_n\ \textbf{defs}\ x_n \qquad \forall i,j = 1,\ldots,n\bullet\ x_i = x_j \ \Rightarrow\ i = j}{\vdash \texttt{class}\ c\,\texttt{\{}M_1\ \ldots\ M_n\texttt{\}}\ \textbf{defs}\ c}$$

$$[\text{meth}]\ \frac{\begin{array}{l}\vdash t\ \textbf{ok} \quad \overline{\vdash t_i\ \textbf{ok}} \quad \overline{\vdash t'_j\ \textbf{ok}} \\ \Gamma = \texttt{this}:\mathsf{ref}\ \mathsf{co<>}\ c,\ \overline{x_i:\mathsf{ref}\,t_i},\ \overline{z_j:\mathsf{ref}\,t'_j} \qquad \vdash \Gamma\ \textbf{ok} \qquad \Gamma,\kappa \vdash s : t\end{array}}{\vdash \kappa\ t\ f\,(\overline{t_i\,x_i})\texttt{\{}\overline{t'_j\,z_j}\texttt{;}\ s\texttt{\}}\ \textbf{defs}\ f}$$

$$[\text{field}]\ \frac{\vdash t\ \textbf{ok}}{\vdash t\ x\texttt{;}\ \textbf{defs}\ x} \qquad\qquad [\text{tassg}]\ \frac{\forall i,j = 1,\ldots,n\bullet\ \ x_i = x_j \Rightarrow i = j}{\vdash x_1 : \tau_1,\ldots,x_n : \tau_n\ \ \textbf{ok}}$$

9. TERM TYPING JUDGEMENTS $\boxed{\Gamma,\kappa \vdash e : \tau}$

$$[\text{var}_l]\ \frac{(x:\tau) \in \Gamma}{\Gamma,\kappa \vdash x : \tau} \qquad [\text{var}_f]\ \frac{(\texttt{this}:\mathsf{ref}\,\mu\ c) \in \Gamma \quad \vdash \mathit{FldsMths}(c) = \langle\{\ldots,x{:}\tau,\ldots\},F\rangle}{\Gamma,\kappa \vdash \texttt{this.}x : \tau}$$

$$[\text{rd}_{cp}]\ \frac{\Gamma,\kappa \vdash \nu : \mathsf{ref}\,\tau \quad \tau' = \tau[\texttt{read}/\texttt{free}] \quad \tau = \texttt{free<...>}\ c \Rightarrow \kappa = \texttt{obs} \wedge \nu \in \mathit{Id}}{\Gamma,\kappa \vdash \texttt{val(}\nu\texttt{)} : \tau'}$$

$$[\text{rd}_{dst}]\ \frac{\Gamma,\kappa \vdash \nu : \mathsf{ref}\,\tau \quad \nu \neq \texttt{this} \quad \nu = \texttt{this.}y \Rightarrow \kappa = \texttt{mut}}{\Gamma,\kappa \vdash \texttt{destval(}\nu\texttt{)} : \tau}$$

$$[\text{null}]\ \frac{\vdash c\ \textbf{ok}}{\Gamma,\kappa \vdash \texttt{null<}\delta\texttt{>} : \texttt{free<}\delta\texttt{>}\ c} \qquad [\text{new}]\ \frac{\vdash c\ \textbf{ok}}{\Gamma,\kappa \vdash \texttt{new<}\delta\texttt{>}\ c\texttt{()} : \texttt{free<}\delta\texttt{>}\ c}$$

$$\text{[call]}\ \frac{\Gamma,\kappa \vdash e : \mu\ c \quad \vdash (\,f : \overline{\tau_i} \xrightarrow{\kappa^\star} \tau\,) \in \Sigma(\mu\ c) \quad \kappa^\star = \mathtt{mut} \Rightarrow \mu \in \mathbf{Wr}(\kappa) \quad \overline{\Gamma,\kappa \vdash e_i : \tau_i'} \quad \overline{\vdash \tau_i' \leq_{\mathrm{m}} \tau_i}}{\Gamma,\kappa \vdash e \Leftarrow f\,(\overline{e_i}) : \tau}$$

$$\text{[upd]}\ \frac{\Gamma,\kappa \vdash \nu : \mathsf{ref}\,\tau \quad \Gamma,\kappa \vdash e : \tau' \quad \vdash \tau' \leq_{\mathrm{m}} \tau \quad \nu \neq \mathtt{this} \quad \nu = \mathtt{this}.y \Rightarrow \kappa = \mathtt{mut}}{\Gamma,\kappa \vdash \nu = e;\ : \mathsf{Cmd}}$$

$$\text{[ret]}\ \frac{\Gamma,\kappa \vdash e : \tau}{\Gamma,\kappa \vdash \mathtt{return}\ e;\ : \tau} \qquad \text{[seq]}\ \frac{\Gamma,\kappa \vdash s_1 : \mathsf{Cmd} \quad \Gamma,\kappa \vdash s_2 : \tau}{\Gamma,\kappa \vdash s_1\ s_2 : \tau}$$

$$\text{[if]}\ \frac{\Gamma,\kappa \vdash e_1 : \mu_1\ c_1 \quad \Gamma,\kappa \vdash e_2 : \mu_2\ c_2 \quad \Gamma,\kappa \vdash s : \mathsf{Cmd}}{\Gamma,\kappa \vdash \mathtt{if}\,(e_1\,\psi\,e_2)\ \{s\} : \mathsf{Cmd}}$$

$$\text{[wh]}\ \frac{\Gamma,\kappa \vdash e_1 : \mu_1\ c_1 \quad \Gamma,\kappa \vdash e_2 : \mu_2\ c_2 \quad \Gamma,\kappa \vdash s : \mathsf{Cmd}}{\Gamma,\kappa \vdash \mathtt{while}\,(e_1\,\psi\,e_2)\ \{s\} : \mathsf{Cmd}}$$

where $\mathbf{Wr}(\mathtt{obs}) =_{\mathrm{df}} \{\mathtt{free}\text{<}\ldots\text{>}\}$
$\mathbf{Wr}(\mathtt{mut}) =_{\mathrm{df}} \{\mathtt{free}\text{<}\ldots\text{>}, \mathtt{rep}\text{<}\ldots\text{>}, \mathtt{co}\text{<>}\}$

A.3 Semantic Structures

10. SEMANTIC DOMAINS

environment	$\eta_h^\kappa \in \mathit{Env}$	$=_{\mathrm{df}} (\mathit{Id} \nrightarrow \mathcal{Loc}) \times \mathcal{K} \times \mathcal{V}$
store	$\mathfrak{s} \in \mathfrak{Store}$	$=_{\mathrm{df}} \mathcal{Loc} \nrightarrow \mathcal{V}$
object-map	$\mathit{om} \in \mathit{Omap}$	$=_{\mathrm{df}} \mathbb{O} \nrightarrow ((\mathit{Id} \nrightarrow \mathcal{Loc}) \times (\mathit{Id} \nrightarrow \mathit{Mth}))$
object graph	$\mathfrak{g} \in \mathfrak{Graph}$	$=_{\mathrm{df}} \mathbb{N}^{\mathbb{O}\times\mathcal{M}\times\mathbb{O}}$
location (l-value)	$\ell \in \mathcal{Loc}$	$=_{\mathrm{df}} \biguplus_{\tau\in\mathcal{M}\times\mathbb{C}} \mathcal{Loc}_\tau$
handle (value)	$h \in \mathcal{V}$	$=_{\mathrm{df}} (\mathbb{O} \cup \{\mathsf{nil}\}) \times \mathcal{M} \times (\mathbb{O} \cup \{\mathsf{nil}\})$
object-identifier	$o \in \mathbb{O}$	$=_{\mathrm{df}} \biguplus_{c\in\mathbb{C}} \mathbb{O}_c$
object value	$\langle \varrho, F\rangle \in$	$(\mathit{Id} \nrightarrow \mathcal{Loc}) \times (\mathit{Id} \nrightarrow \mathit{Mth})$
valid modes	$\mu \in \mathcal{M}$	$=_{\mathrm{df}} \{\mu \mid\ \vdash \mu\ \mathbf{ok}\}$

with infinite countable sets $\mathbb{O}_c$ given for all $c \in \mathbb{C}$ and $\mathcal{Loc}_\tau$ for all $\tau \in \mathcal{M} \times \mathbb{C}$

11. INTERPRETATION OF FORMAL TYPE TERMS AND TYPE CONSISTENCY

$$\begin{aligned}
[\![\mathsf{ref}\,\mu\ c]\!] &=_{\mathrm{df}} \mathcal{Loc}_{\mu\ c}\\
[\![\mu\ c]\!] &=_{\mathrm{df}} (\mathbb{O} \cup \{\mathsf{nil}\}) \times \{\mu\} \times (\mathbb{O}_c \cup \{\mathsf{nil}\})\\
[\![\mathsf{obj}\ c]\!] &=_{\mathrm{df}} \{\langle\varrho, F\rangle \mid\ \vdash \mathit{FldsMths}(c) = \langle\Gamma, F\rangle \text{ and } \varrho \models \Gamma\}\\
[\![\mathsf{Cmd}]\!] &=_{\mathrm{df}} \{\epsilon\}
\end{aligned}$$

$$\begin{aligned}
\eta \models \Gamma \quad &\Leftrightarrow_{\mathrm{df}} \mathrm{dom}(\eta) = \mathrm{dom}(\Gamma)\ \wedge\ \forall x \in \mathrm{dom}(\Gamma)\bullet\ \ \eta(x) \in [\![\Gamma(x)]\!]\\
\models \mathfrak{s} \quad &\Leftrightarrow_{\mathrm{df}} \forall \tau \in \mathcal{M}\times\mathbb{C}, \ell \in \mathrm{dom}(\mathfrak{s})\bullet\ \ell \in \mathcal{Loc}_\tau \Rightarrow \mathfrak{s}(\ell) \in [\![\tau]\!]\\
\models \mathit{om} \quad &\Leftrightarrow_{\mathrm{df}} \forall c \in \mathbb{C}, o \in \mathrm{dom}(\mathit{om})\bullet\ o \in \mathbb{O}_c \Rightarrow \mathit{om}(o) \in [\![\mathsf{obj}\ c]\!]
\end{aligned}$$

A.4 Small Step Semantics

All definitions are relative to a program p.

12. INITIAL CONFIGURATION for $p \equiv D_1, \ldots, D_n$ with $\vdash D_n$ **ok** c_n

$$e_0,\ \eta_0, \mathfrak{s}_0, om_0, \mathfrak{g}_0 \text{ where } \begin{array}{ll} e_0 & =_{\mathrm{df}} \texttt{new<> ().main()} \\ \eta_0 & =_{\mathrm{df}} \emptyset^{\mathsf{obs}}_{\langle \mathsf{nil}, \mathsf{read<>}, \mathsf{nil} \rangle} \\ \mathfrak{s}_0 & =_{\mathrm{df}} \emptyset \\ om_0 & =_{\mathrm{df}} \emptyset \\ \mathfrak{g}_0 & =_{\mathrm{df}} \emptyset \end{array}$$

13. REDUCTION STEP CONTROLLED BY REDUCTION CONTEXT

$$\frac{\mathcal{E} \in R_1^{\square} \quad e,\ \vec{\eta}, \mathfrak{s}, om, \mathfrak{g} \longrightarrow e',\ \vec{\eta}', \mathfrak{s}', om', \mathfrak{g}'}{\mathcal{E}[e],\ \vec{\eta}, \mathfrak{s}, om, \mathfrak{g} \Longrightarrow \mathcal{E}[e'],\ \vec{\eta}', \mathfrak{s}', om', \mathfrak{g}'}$$

$$\frac{\mathcal{E} \in R_1^{\square} \quad e,\ \vec{\eta}, \mathfrak{s}, om, \mathfrak{g} \Longrightarrow e',\ \vec{\eta}', \mathfrak{s}', om', \mathfrak{g}'}{\mathcal{E}[\ll e \gg],\ \eta_h^{\kappa} \bullet \vec{\eta}, \mathfrak{s}, om, \mathfrak{g} \Longrightarrow \mathcal{E}[\ll e' \gg],\ \eta_h^{\kappa} \bullet \vec{\eta}', \mathfrak{s}', om', \mathfrak{g}'}$$

$$\begin{array}{rl} R_1^{\square} ::= & \square \\ \mid & \texttt{val}(R_1^{\square}) \mid \texttt{destval}(R_1^{\square}) \mid R_1^{\square} \Leftarrow Id(\mathrm{E}^*) \mid \mathcal{V} \Leftarrow Id((\mathcal{V},)^*\, R_1^{\square}\, (,\mathrm{E})^*) \\ \mid & R_1^{\square}\,\mathrm{S} \mid R_1^{\square} = \mathrm{E}; \mid \mathcal{L}oc = R_1^{\square}; \mid \texttt{return}\ R_1^{\square}; \mid \texttt{if}\,(R_1^{\square} \Psi\, R)\,\{\mathrm{S}\} \mid \texttt{if}\,(\mathcal{V}\, \Psi\, R_1^{\square})\,\{\mathrm{S}\} \end{array}$$

14. REDUCTION AT THE REDEX, where η_h^{κ} is the top element in $\vec{\eta}$

$$\{\mathrm{var}_l\}\ \frac{\eta(x) \doteq \ell}{x,\ \eta_h^{\kappa}, \mathfrak{s}, om, \mathfrak{g} \longrightarrow \ell,\ \eta_h^{\kappa}, \mathfrak{s}, om, \mathfrak{g}}$$

$$\{\mathrm{var}_f\}\ \frac{\eta(\texttt{this}) \doteq \ell \quad \mathfrak{s}(\ell) \doteq \langle o, \mu, o \rangle \quad om(o) \doteq \langle \varrho, F \rangle \quad \varrho(x) \doteq \ell'}{\texttt{this}.x,\ \eta_h^{\kappa}, \mathfrak{s}, om, \mathfrak{g} \longrightarrow \ell',\ \eta_h^{\kappa}, \mathfrak{s}, om, \mathfrak{g}}$$

$$\{\mathrm{rd}_{cp}\}\ \frac{\mathfrak{s}(\ell) \doteq \langle o, \mu, \omega \rangle \quad \mu' = \mu[\texttt{read}/\texttt{free}]}{\texttt{val}(\ell),\ \eta_h^{\kappa}, \mathfrak{s}, om, \mathfrak{g} \longrightarrow \langle o, \mu', \omega \rangle,\ \eta_h^{\kappa}, \mathfrak{s}, om, \mathfrak{g} \oplus o \xrightarrow{\mu'} \omega}$$

$$\{\mathrm{rd}_{dst}\}\ \frac{\mathfrak{s}(\ell) \doteq \langle o, \mu, \omega \rangle}{\texttt{destval}(\ell),\ \eta_h^{\kappa}, \mathfrak{s}, om, \mathfrak{g} \longrightarrow \langle o, \mu, \omega \rangle,\ \eta_h^{\kappa}, \mathfrak{s}[\ell \mapsto \langle o, \mu, \mathsf{nil} \rangle], om, \mathfrak{g}}$$

$$\{\mathrm{null}\}\ \frac{h \doteq \langle \mathbf{s}, \mu_{\mathbf{r}}, \mathbf{r} \rangle}{\texttt{null<}\delta\texttt{>},\ \eta_h^{\kappa}, \mathfrak{s}, om, \mathfrak{g} \longrightarrow \langle \mathbf{r}, \texttt{free<}\delta\texttt{>}, \mathsf{nil} \rangle,\ \eta_h^{\kappa}, \mathfrak{s}, om, \mathfrak{g}}$$

$$\{\mathrm{new}\}\ \frac{\begin{array}{lll} h \doteq \langle \mathbf{s}, \mu_{\mathbf{r}}, \mathbf{r} \rangle & \vdash \mathit{FldsMths}(c) \doteq \langle \{\overline{x_i : \mathsf{ref}\ \mu_i\ c_i}\}, F \rangle & h' = \langle \mathbf{r}, \texttt{free<}\delta\texttt{>}, \mathbf{o} \rangle \\ \text{fresh } \mathbf{o} \in \mathbb{O}_c & \overline{\text{fresh } \ell_i \in [\![\mathsf{ref}\ \mu_i\ c_i]\!]} \quad \varrho = \{\overline{x_i \mapsto \ell_i}\} & \overline{h_i = \langle \mathbf{o}, \mu_i, \mathsf{nil} \rangle} \end{array}}{\texttt{new<}\delta\texttt{>}\ c\texttt{()},\ \eta_h^{\kappa}, \mathfrak{s}, om, \mathfrak{g} \longrightarrow h',\ \eta_h^{\kappa}, \mathfrak{s}[\ell_i \mapsto h_i], om[\mathbf{o} \mapsto \langle \varrho, F \rangle], \mathfrak{g} \oplus h'}$$

$$\{\text{call}\}\ \dfrac{\begin{array}{l} \mathbf{r} \in \mathbb{O}_c, \quad om(\mathbf{r}) \doteq \langle \ldots, F \rangle, \quad F(f) \doteq \kappa^\star\, \tau\, f\,(\overline{\mu_i\ c_i\ y_i})\,\texttt{\{}\overline{\mu'_j\ c'_j\ z_j}\texttt{;}\ s\texttt{\}} \\ \text{fresh } \ell \in [\![\mathsf{ref}\ \mathsf{co}\texttt{<>}\ c]\!],\ \overline{\text{fresh } \ell^y_i \in [\![\mathsf{ref}\ \mu_i\ c_i]\!]},\ \overline{\text{fresh } \ell^z_j \in [\![\mathsf{ref}\ \mu'_j\ c'_j]\!]} \\ \eta^\star\ = \{\texttt{this} \mapsto \ell, \overline{y_i \mapsto \ell^y_i}, \overline{z_j \mapsto \ell^z_j}\} \\ \mathfrak{s}'\ = \mathfrak{s}[\ell \mapsto \langle \mathbf{r}, \mathsf{co}\texttt{<>}, \mathbf{r} \rangle, \overline{\ell^y_i \mapsto \langle \mathbf{r}, \mu_i, \mathbf{o}_i \rangle}, \overline{\ell^z_j \mapsto \langle \mathbf{r}, \mu'_j, \mathsf{nil} \rangle}] \\ \mathfrak{g}'\ = \mathfrak{g} \ominus \overline{\mathbf{s} \xrightarrow{\mu''_i} \mathbf{o}_i} \oplus \mathbf{r} \xrightarrow{\mathsf{co}\texttt{<>}} \mathbf{r} \oplus \overline{\mathbf{r} \xrightarrow{\mu_i} \mathbf{o}_i} \end{array}}{\langle \mathbf{s}, \mu_{\mathbf{r}}, \mathbf{r} \rangle \Leftarrow f\,(\overline{\langle \mathbf{s}, \mu''_i, \mathbf{o}_i \rangle}),\ \eta^\kappa_h, \mathfrak{s}, om, \mathfrak{g} \longrightarrow\ \ll s \gg,\ \eta^\kappa_h \bullet \eta^{\star\kappa^\star}_{\langle \mathbf{s}, \mu_{\mathbf{r}}, \mathbf{r} \rangle}, \mathfrak{s}', om, \mathfrak{g}'}$$

$$\{\text{ret}\}\ \dfrac{\mathfrak{s}' = \mathfrak{s}[\ell \mapsto \bot \mid \ell \in \mathrm{im}(\eta^\star)] \quad \mathfrak{g}' = \mathfrak{g} \oplus \mathbf{s} \xrightarrow{\mu_{\mathbf{r}} \circ \mu} \mathbf{o} \ominus \mathbf{s} \xrightarrow{\mu_{\mathbf{r}}} \mathbf{r} \ominus \mathbf{r} \xrightarrow{\mu} \mathbf{o} \ominus \mathfrak{s}(\mathrm{im}(\eta^\star))}{\ll \texttt{return}\ \langle \mathbf{r}, \mu, \mathbf{o} \rangle\,\texttt{;} \gg,\ \eta^\kappa_h \bullet \eta^{\star\kappa^\star}_{\langle \mathbf{s}, \mu_{\mathbf{r}}, \mathbf{r} \rangle}, \mathfrak{s}, om, \mathfrak{g} \longrightarrow \langle \mathbf{s}, \mu_{\mathbf{r}} \circ \mu, \mathbf{o} \rangle, \eta^\kappa_h, \mathfrak{s}', om, \mathfrak{g}'}$$

$$\{\text{upd}\}\ \dfrac{\ell \in \mathcal{L}oc_{\mu\ c}}{\ell = \langle o, \tilde{\mu}, \tilde{\omega} \rangle;,\ \eta^\kappa_h, \mathfrak{s}, om, \mathfrak{g} \longrightarrow\ \epsilon,\ \eta^\kappa_h, \mathfrak{s}[\ell \mapsto \langle o, \mu, \tilde{\omega} \rangle], om, \mathfrak{g} \ominus o \xrightarrow{\tilde{\mu}} \tilde{\omega} \ominus \mathfrak{s}(\ell) \oplus o \xrightarrow{\mu} \tilde{\omega}}$$

$$\{\text{if}_t\}\ \dfrac{[\![\psi]\!](\omega, \omega')}{\texttt{if}\,(\langle o, \mu, \omega \rangle\ \psi\ \langle o, \mu', \omega' \rangle)\ \texttt{\{}s\texttt{\}},\ \eta^\kappa_h, \mathfrak{s}, om, \mathfrak{g} \longrightarrow s,\ \eta^\kappa_h, \mathfrak{s}, om, \mathfrak{g} \ominus o \xrightarrow{\mu} \omega \ominus o \xrightarrow{\mu'} \omega'}$$

$$\{\text{if}_f\}\ \dfrac{\neg[\![\psi]\!](\omega, \omega')}{\texttt{if}\,(\langle o, \mu, \omega \rangle\ \psi\ \langle o, \mu', \omega' \rangle)\ \texttt{\{}s\texttt{\}},\ \eta^\kappa_h, \mathfrak{s}, om, \mathfrak{g} \longrightarrow \epsilon,\ \eta^\kappa_h, \mathfrak{s}, om, \mathfrak{g} \ominus o \xrightarrow{\mu} \omega \ominus o \xrightarrow{\mu'} \omega'}$$

$$\{\text{wh}\}\ \dfrac{}{\texttt{while(}e_1 \psi\, e_2\texttt{)\{}s\texttt{\}},\ \eta^\kappa_h, \mathfrak{s}, om, \mathfrak{g} \longrightarrow \texttt{if(}e_1 \psi\, e_2\texttt{)\{}s\ \texttt{while(}e_1 \psi\, e_2\texttt{)\{}s\texttt{\}\}},\ \eta^\kappa_h, \mathfrak{s}, om, \mathfrak{g}}$$

where $[\![\texttt{==}]\!](\omega, \omega') \Leftrightarrow_{\mathrm{df}} \omega = \omega'$ and $[\![\texttt{!=}]\!](\omega, \omega') \Leftrightarrow_{\mathrm{df}} \omega \neq \omega'$

15. Helper Functions

$$\mathfrak{g} \oplus o \xrightarrow{\mu} \omega =_{\mathrm{df}} \begin{cases} \mathfrak{g} & \text{if } \mathsf{nil} \in \{o, \omega\} \\ \mathfrak{g} \uplus \{o \xrightarrow{\mu} \omega\} & \text{otherwise} \end{cases} \qquad \mathfrak{g} \ominus o \xrightarrow{\mu} \omega =_{\mathrm{df}} \begin{cases} \mathfrak{g} & \text{if } \mathsf{nil} \in \{o, \omega\} \\ \mathfrak{g} \setminus\!\!\setminus \{o \xrightarrow{\mu} \omega\} & \text{otherwise} \end{cases}$$

where $\uplus$ is multiset-union and $\setminus\!\!\setminus$ is multiset-subtraction.

Appendix B

JaM Code of the Map Example

B.1 In Basic Desugared JaM

Below, the map example with iterators is implemented in raw JaM, without syntactic sugar. This means, every read access to a variable ν is explicitly specified by val(ν) or destval(ν).

The limited Java base naturally entails some awkwardnesses in the expression. Methods without real return value, i.e., mutator methods that would be `void` in Java, are written to return `this`. Returning `this` is necessary for calling these mutator methods through a (destructively read) `free` reference without losing it forever. Since there is no `boolean`, `SetImp`'s `contains` operation returns `null` if the given object o was not found in the set, and returns an `elem` reference to o if it turned out to be an element in the set. Without the possibility of returning directly from the middle of a method, `res` variables are sometimes needed to transport the result to the end of the method (`current`, `contains`, `lookup`). Since additionally the loop guards are so restricted, we always continue going through the entire list even when the desired element was already found (and thus is not expected to occur again).

```
/********************* DSComponents package *********************************/
// standard pair class
class Pair {
   fst<> Object      fst;
   snd<> Object      snd;

   mut co<> Pair     Set(fst<> Object a, snd<> Object b)
                                        { this.fst = val(a); this.snd = val(b);
                                          return val(this); }
   obs fst<> Object  first()            { return val(this.fst); }
   obs snd<> Object  second()           { return val(this.snd); }
}

// single linked nodes with Pair data
class PNode {
```

```
   co<> PNode      next;
   data<> Pair     data;

   mut co<> PNode  SetNext(co<> PNode n)  { this.next = val(n); return val(this); }
   mut co<> PNode  SetData(data<> Pair p) { this.data = val(p); return val(this); }
   obs co<> PNode  next()                 { return val(this.next); }
   obs data<> Pair data()                 { return val(this.data); }
}

/********************* DSIterators package **********************************/
// iterator over single-linked list of PNodes
class PNodeIt {
   dest<> PNode      curnode;

   mut co<> PNodeIt  StartAt(dest<> PNode n)
                                { this.curnode = val(n); return val(this); }
   mut co<> PNodeIt  Step()     { this.curnode = val(this.curnode)<=next();
                                  return val(this); }
   obs dest<> PNode  current()  { return val(this.curnode); }
}

// iterator over Pairs in a PNodeIt's PNodes
class PDataIt {
   rep<dest=read<data=dest<>>> PNodeIt  nodes;

   mut co<> PDataIt   Wrap(free<dest=read<data=dest<>>> PNodeIt nn)
                                { this.nodes = destval(nn); return val(this); }
   mut co<> PDataIt   Step()    { val(this.nodes)<=Step(); return val(this); }
   obs dest<> Pair    current() { dest<> Pair res;
                                  if( val(this.nodes)<=current() != null<> )
                                  {  res = val(this.nodes)<=current()<=data();  }
                                  return res;
                                }
}

/********************* DSCollectionImp package ******************************/
// set of Pairs implemented with single-linked list
class PSetImp {
   rep<data=elem<>> PNode  anchor;

   mut co<> PSetImp   Add(elem<> Pair e)
   {
      if( val(this)<=contains(e) == null<> )
      {  this.anchor = new<data=elem<>> PNode();
         this.anchor<=SetData( val(e) );
         this.anchor<=SetNext( val(this.anchor) );
      }
      return val(this);
   }

   mut co<> PSetImp   Remove(read<> Pair e)
```

```
  {  rep<data=elem<>> PNode  prenode;

     if( val(this.anchor) != null<> )
     {  if( val(this.anchor)<=data() != val(e) )
        {  prenode = val(this.anchor);
           while( val(prenode)<=next() != null<> )
           {  if( val(prenode)<=next()<=data() == val(e) )
              {  val(prenode)<=SetNext( val(prenode)<=next()<=next() );  }
              prenode = val(prenode)<=next();
           }
        }
        if( val(prenode) == null<> )  // equivalent to 'else'
        {  this.anchor = val(this.anchor)<=next();  }
     }
     return val(this);
  }

  obs elem<> Pair   contains(read<> Pair e)
  {  elem<> Pair  res;
     rep<data=elem<>> PNode  node;

     node = val(this.anchor);
     while( val(node) != null<> )
     {  if( val(node)<=data() == val(e) ) { res = val(node)<=data(); }
        node = val(node)<=next();
     }
     return val(res);
  }

  obs free<dest=elem<>> PDataIt   elements()
  {  free<dest=rep<data=elem<>>> PNodeIt nn;

     nn = new<dest=rep<data=elem<>>> PNodeIt()<=StartAt( val(this.anchor) );
     return new<dest=elem<>> PDataIt()<=Wrap( destval(nn) );
  }
}

// standard map Object to Object implementation with an entry-set object
class MapImp {
  rep<elem=rep<fst=key<>, snd=value<>>> PSetImp  entryset;

  mut co<> MapImp   Init()
  {  this.entryset = new<elem=rep<fst=key<>, snd=value<>>> PSetImp();
     return val(this);
  }

  mut co<> MapImp   Add(key<> Object k, value<> Object v)
  {  rep<fst=key<>, snd=value<>> Pair  p;
     free<dest=rep<fst=key<>, snd=value<>>> PDataIt  entries;

     // check for old entry with key k
```

```
      entries = val(this.entryset)<=elements();
      while( val(entries)<=current() != null<> )
      {  if( val(entries)<=current()<=first() == val(k) )
         {  p = val(entries)<=current();  }
         entries = destval(entries)<=Step();
      }

      // if there is none, create new entry and insert it
      if( val(p) == null<> )
      {  p = new<fst=key<>,snd=value<>> Pair();
         val(this.entryset)<=Add( val(p) );
      }

      // set key and value of old/new entry
      val(p)<=Set( val(k), val(v) );
      return val(this);
   }

   mut co<> MapImp   Remove(read<> Object k)
   {  free<dest=rep<fst=key<>, snd=value<>>> PDataIt  entries;

      entries = val(this.entryset)<=elements();
      while( val(entries)<=current() != null<> )
      {  if( val(entries)<=current()<=first() == val(k) )
         {  val(this.entryset)<=Remove( val(entries)<=current() );  }
         entries = destval(entries)<=Step();
      }
      return val(this);
   }

   obs value<> Object   lookup(read<> Object k)
   {  value<> Object  res;
      free<dest=rep<fst=key<>, snd=value<>>> PDataIt  entries;

      entries = val(this.entryset)<=elements();
      while( val(entries)<=current() != null<> )
      {  if( val(entries)<=current()<=first() == val(k) )
         {  res = val(entries)<=current()<=second();  }
         entries = destval(entries)<=Step();
      }
      return val(res);
   }

   obs free<dest=rep<fst=key<>, snd=value<>>> PDataIt   entries()
   {
      return val(this.entryset)<=elements();
   }
}
```

B.2 Refactored with Self-Calls

Class `MapImp` defines three time—in methods `Add`, `Remove`, and `lookup`—the same iteration over the `entryset` component in search for a given (potential) key object `k`. With the extension of JaM for unrestricted self-calls in §7.2.3, class `MapImp` can be restructured by factoring this search into a separate method `find_entry`:

```
class MapImp {
  rep<elem=rep<fst=key<>, snd=value<>>> PSetImp  entryset;

  mut co<> MapImp   Init()
  {  this.entryset = new<elem=rep<fst=key<>, snd=value<>>> PSetImp();
     return val(this);
  }

  obs rep<fst=key<>, snd=value<>> Pair   find_entry(read<> Object k)
  {  rep<fst=key<>, snd=value<>> Pair  p;
     free<dest=rep<fst=key<>, snd=value<>>> PDataIt  entries;

     entries = val(this.entryset)<=elements();
     while( val(entries)<=current() != null<> )
     {  if( val(entries)<=current()<=first() == val(k) )
        {  p = val(entries)<=current();  }
        entries = destval(entries)<=Step();
     }
     return val(p);
  }

  mut co<> MapImp   Add(key<> Object k, value<> Object v)
  {  rep<fst=key<>, snd=value<>> Pair  p;

     // check for old entry with key k
     p = find_entry( val(k) );  // <- self call

     // if there is none, create new entry and insert it
     if( val(p) == null<> )
     {  p = new<fst=key<>,snd=value<>> Pair();
        val(this.entryset)<=Add( val(p) );
     }

     // set key and value of old/new entry
     val(p)<=Set( val(k), val(v) );
     return val(this);
  }

  mut co<> MapImp   Remove(read<> Object k)
  {  rep<fst=key<>, snd=value<>> Pair  p;

     p = find_entry( val(k) );  // <- self call
     if( val(p) != null<> ) { val(this.entryset)<=Remove( val(p) ); }
```

```
    return val(this);
  }

  obs value<> Object   lookup(read<> Object k)
  {  value<> Object  res;
     rep<fst=key<>, snd=value<>> Pair  p;

     p = find_entry( val(k) );  // <- self call
     if( val(p) != null<> ) { res = val(p)<=second(); }
     return val(res);
  }

  obs free<dest=rep<fst=key<>, snd=value<>>> PDataIt   entries()
  {
     return val(this.entryset)<=elements();
  }
}
```

B.3 In Sugared Generic JaM

Below, the map example with iterators is implemented in sugared JaM (§7.2.1) with interfaces and class parameters (§7.2.2).

```
/********************** Interfaces package **********************************/
interface Iterator<T> {
   mut void     Step();
   obs dest T   current();
}
interface Set<T> {
   mut void    Add(elem T e);
   mut void    Remove(read T e);
   obs elem T  contains(read T e);
   obs free<dest=elem> Iterator<T>  elements();
}
interface Map<K,V> {
   mut void     Add(key K k, value V v);
   mut void     Remove(read K k);
   obs value V  lookup(read K k);
   obs free<dest=rep<fst=key, snd=value>> Iterator<Pair<K,V>>  entries();
}
/********************* DSComponents package ********************************/
// standard pair class
class Pair<A,B> {
   fst A      fst;
   snd B      snd;

   mut void   Set(fst A a, snd B b)  { this.fst = a; this.snd = b; }
   obs fst A  first()                { return this.fst; }
   obs snd B  second()               { return this.snd; }
```

```
}

// single linked nodes
class Node<T> {
   co Node<T>     next;
   data T         data;

   mut void       SetNext(co Node<T> n) { this.next = n; }
   mut void       SetData(data T p)     { this.data = p; }
   obs co Node<T> next()                { return this.next; }
   obs data T     data()                { return this.data; }
}

/******************** DSIterators package **********************************/
// iterator over single-linked list of Nodes
class NodeIt<T> implements Iterator<Node<T>> {
   dest Node<T>     curnode;

   mut void         StartAt(dest Node<T> n)
                              { this.curnode = n; }
   mut void         Step()    { this.curnode = this.curnode.next(); }
   obs dest Node<T> current() { return this.curnode; }
}

// iterator over data in the Nodes from an interator
class DataIt<T> implements Iterator<T> {
   rep<dest=read<data=dest>> Iterator<Node<T>>  nodes;

   mut void     Wrap(free<dest=read<data=dest>> Iterator<Node<T>> nn)
                             { this.nodes = nn; }
   mut void     Step()       { this.nodes.Step(); }
   obs dest T   current()    { dest T res;
                               if( this.nodes.current() != null )
                               {  res = this.nodes.current().data();  }
                               return res;
                             }
}

/******************** DSCollectionImp package ******************************/
// set implemented with single-linked list
class SetImp<T> implements Set<T> {
   rep<data=elem> Node<T>  anchor;

   mut void   Add(elem T e)
   {
      if( this.contains(e) == null )
      {  this.anchor = new<data=elem> Node<T>();
         this.anchor.SetData( e );
         this.anchor.SetNext( this.anchor );
      }
   }
```

```
  mut void   Remove(read T e)
  {  rep<data=elem> Node<T>  prenode;

     if( this.anchor != null )
     {  if( this.anchor.data() != e )
        {  prenode = this.anchor;
           while( prenode.next() != null )
           {  if( prenode.next().data() == e )
              {  prenode.SetNext( prenode.next().next() );  }
              prenode = prenode.next();
           }
        }
        if( prenode == null )  // equivalent to 'else'
        {  this.anchor = this.anchor.next();  }
     }
  }

  obs elem T   contains(read T e)
  {  elem T  res;
     rep<data=elem> Node<T>  node;

     node = this.anchor;
     while( node != null )
     {  if( node.data() == e ) { res = node.data(); }
        node = node.next();
     }
     return res;
  }

  obs free<dest=elem> Iterator<T>  elements()
  {  free<dest=rep<data=elem>> NodeIt<T> nn;

     nn = new<dest=rep<data=elem>> NodeIt<T>().StartAt( this.anchor );
     return new<dest=elem> DataIt<T>().Wrap( nn );
  }
}

// standard map implementation with an entry-set object
class MapImp<K,V> implements Map<K,V> {
  rep<elem=rep<fst=key, snd=value>> Set<Pair<K,V>>  entryset;

  mut void   Init()
  {  this.entryset = new<elem=rep<fst=key, snd=value>> SetImp<Pair<K,V>>();
  }

  obs rep<fst=key, snd=value> Pair<K,V>   find_entry(read K k)
  {  rep<fst=key, snd=value> Pair<K,V>  p;
     free<dest=rep<fst=key, snd=value>> DataIt<Pair<K,V>>  entries;

     entries = this.entryset.elements();
```

```
    while( entries.current() != null )
    {  if( entries.current().first() == k )
       {  p = entries.current();  }
       entries.Step();
    }
    return p;
  }

  mut void   Add(key K k, value V v)
  {  rep<fst=key, snd=value> Pair<K,V>  p;

    // check for old entry with key k
    p = find_entry(k);  // <- self call

    // if there is none, create new entry and insert it
    if( p == null )
    {  p = new<fst=key,snd=value> Pair<K,V>();
       this.entryset.Add( p );
    }

    // set key and value of old/new entry
    p.Set( k, v );
  }

  mut void   Remove(read K k)
  {  rep<fst=key, snd=value> Pair<K,V>  p;

    p = find_entry(k);  // <- self call
    if( p != null ) { this.entryset.Remove(p); }
  }

  obs value V   lookup(read K k)
  {  value Object  res;
     rep<fst=key, snd=value> Pair<K,V>  p;

     p = find_entry(k);  // <- self call
     if( p != null ) { res = p.second(); }
     return res;
  }

  obs free<dest=rep<fst=key, snd=value>> Iterator<Pair<K,V>>  entries();
  {
     return this.entryset.elements();
  }
}
```

Bibliography

[AC96] M Abadi, L Cardelli: *A Theory of Objects*; Springer 1996.

[ASS96] Harold Abelson, Gerald Jay Sussman, Julie Sussman: *Structure and Interpretation of Computer Programs* (2*nd.* ed.); MIT Press 1996.

[AW+92] M Aksit, K Wakita, et al.: *Abstracting object interactions using composition filters*; Project Report of TRESE group; Univ. of Twente, the Netherlands 1992.

[AKC01] Jonathan Aldrich, Craig Chambers, David Notkin: *ArchJava: Connecting Software Architecture to Implementation*; Submitted for publication, 2001. http://citeseer.nj.nec.com/aldrich01archjava.html

[ACN02] J Aldrich, V Kostadinov, C Chambers: *Alias annotations for program understanding*; *OOPSLA'02*; ACM 2002.

[Alm97] Paulo Sérgio Almeida: *Balloon Types: Controlling Sharing of State in Data Types*; *ECOOP'97*; LNCS 1241; Springer 1997.

[Ar+96] Alessandro Artale, Enrico Franconi, Nicola Guarino, Luca Pazzi: *Part-whole relations in object-centered systems: An overview*; *Data & Knowledge Engineering* **20**; Elsevier 1996.

[Ame87] Pierre America: *Inheritance and subtyping in a parallel object-oriented language*; *ECOOP'87*; LNCS 276; Springer 1987.

[Ast96] Hernán Astudillo R: *Reorganizing Split Objects*; 138–149 in *OOPSLA'96*; ACM 1996.

[Bak95] Henry G Baker: *'Use-Once' Variables and Linear Objects – Storage Management, Reflection and Multi-Threading*; 45–52 in *SIGPLAN Notices* **30**(1); ACM 1995.

[BN02] Anindya Banerjee, David A Nauman: *Representation Independence, Confinement and Access Control*; 166–177 in *POPL'02*; ACM 2002.

[BW99] Boumedine Belkhouche, Joel Wu: *Behavioral Specification and Analysis of Object-Oriented Designs*; *JOOP* **11**(8); SIGS 1999.

[BLM97] J C Bicarregui, K C Lano, T S E Maibaum: *Objects, Associations and Subsystems: A Hierarchical Approach to Encapsulation*; 324–343 in *ECOOP'97*; LNCS 1241; Springer 1997.

[Bi+80] G M Birtwistle, O-J Dahl, B Myhrhaug, K Nygaard: *Simula begin* (2*nd.* ed.); Studentlitteratur, Bratt-Institut für Neues Lernen, and Cartwell-Brat 1980.

[BC87] Edwin Blake, Steve Cook: *On Including Part Hierarchies in Object-Oriented Languages, with an Implementation in Smalltalk*; 41–50 in *ECOOP'87*; LNCS 276; Springer 1987.

[Bla99] Bruno Blanchet: *Escape analysis for object-oriented languages: Applications to Java*; *OOPSLA'99*; ACM 1999.

[BO98] Conrad Bock, James Odell: *A more complete model of relations and their implementations: aggregation*; *JOOP* **11**(5); SIGS 1998.

[Boo94] Grady Booch: *Object-Oriented Analysis and Design with Applications* (2*nd.* ed.); Addison-Wesley 1994. (first published 1991).

[Bos96] J Bosch: *Object Acquaintance Selection and Binding*; Research Report 13/96 ISRN HKR-RES–96/13–SE; University of Karlskrona/Ronneby, Department of Computer Science and Business Administration 1996. Appeared in *Theory and Practice of Object Systems* **4**(3); 1998.

[BNR01] John Boyland, James Noble, William Retert: *Capabilities for Sharing: A Generalisation of Uniqueness and Read-Only*; 2–27 in *ECOOP'01*; LNCS 2072; Springer 2001.

[Boy01] John Boyland: *Alias Burying: Unique Variables Without Destructive Reads*; 533–553 in *Software - Practice and Experience* **31**(6); Wiley 2001.

[Bre91] R Breu: *Algebraic Specification Techniques in Object Oriented Programming Environments*; LNCS = 562 ; Springer 1991.

[Bro87] F P Brooks: *No Silver Bullet: Essence and Accidents of Software Engineering*; *Computer* April 1987; IEEE 1987. First published in *Information Processing '86*; Elsevier 1986.

[Bru96] Kim B Bruce: *Typing in object-oriented languages: Achieving expressiveness and safety*; Technical Report; Williams College 1996.

[BPV98] Kim B Bruce, Leaf Petersen, Joseph Vanderwaart: *Modules in LOOM: Classes are not enough*; 1998. ftp://ftp.cs.williams.edu/pub/kim/modules.dvi.gz

[BPF97] Kim B Bruce, Leaf Petersen, Adrian Fiech: *Subtyping is not a good "Match" for object-oriented languages*; 104–127 in *ECOOP'97*; LNCS 1241; Springer 1997.

[CR00] Ciarán Bryce, Chrislain Razafimahefa: *An Approach to Safe Object Sharing*; 367–381 in *OOPSLA'00*; ACM 2000.

[Bud95] Tim Budd: *Multiparadigm Programming in Leda*; Addison-Wesley 1995.

[Bun79] Mario Bunge: *Ontology I: The Furniture of the World* (*Treatise on Basic Philosophy* 3); D Reidel Publishing 1979.

[Ca+89] Peter Canning, William Cook, Walter Hill, Walter Olthoff, John C Mitchell: *F-bounded polymorphism for object-oriented programming*; 273–280 in *Functional programming languages and computer architecture*); 1989.

[CW85] Luca Cardelli, Peter Wegner: *On Understanding Types, Data Abstraction, and Polymorphism*; *Computing Surveys* **17**(4); ACM 1985.

[Car97] Luca Cardelli: *Type Systems*; *Handbook of Computer Science and Engineering* Chapter 103; CRC Press 1997.

[Cas97] Guiseppe Castagna: *Object-Oriented Programming: A Unified Foundation*; Birkhäuser 1997.

[CH88] Roger Chaffin, Douglas J Herrmann: *The nature of semantic relations: a comparison of two approaches*; 289–334 in Martha Walton Evens (ed.): *Relational models of the lexicon: representing knowledge in semantic networks*; Cambridge 1988.

[Cha91] Dennis de Champeaux: *Object-Oriented Analysis and Top-Down Software Development*; 360–376 in *ECOOP'91*; LNCS 512; Springer 1991.

[CLF93] Dennis de Champeaux, Doug Lea, Penelope Faure: *Object-Oriented Software Development*; Addison-Wesley 1993.

[CLF92] Dennis de Champeaux, Doug Lea, Penelope Faure: *The Process of Object-Oriented Design*; 45–62 in *OOPSLA'92*; ACM 1992.

[Civ93] Franco Civello: *Roles for composite objects in object-oriented analysis and design*; 376–393 in *OOPSLA'93*; ACM 1993.

[Cla01] David Clarke: *Object Ownership & Containment*; PhD thesis; Univ. of New South Wales 2001.

[CPN98] David G Clarke, John M Potter, James Nobel: *Ownership Types for Flexible Alias Protection*; 48–64 in *OOPSLA'98*; ACM 1998.

[CJ75] Ellis Cohen, David Jefferson: *Protection in the Hydra Operating System*; 141–160 in *SIGOPS* **9**(5); ACM 1975.

[Coo90] William R Cook: *Object-Oriented Programming Versus Abstract Data Types*; *Foundations of Object-Oriented Languages*; LNCS 489; Springer 1991.

[CHC90] William R Cook, Walter Hill, Peter S Canning: *Inheritance is not subtyping*; *POPL'90*; ACM 1990.

[CWM99] Karl Crary, David Walker, Greg Morrisett: *Typed memory management in a calculus of capabilities*; *POPL'99*; ACM 1999.

[DB+96] John Daly, Andrew Brooks, James Miller, Marc Roper, Murray Wood: *Evaluating Inheritance Depth on the Maintainability of Object-Oriented Software*; 109–132 in *Empirical Software Engineering* **1**(2); 1996.

[DLN98] David L Detlefs, K Rustan M Leino, Greg Nelson: *Wrestling with rep exposure*; SRC Research Report 156; DEC 1998.

[DL97] Krishna Kishore Dhara, Gary T Leavens: *Forcing behavioral subtyping through specification inheritance*; *ICSE'96* ; IEEE 1996. Also in a revised version as Technical Report TR#95-20c at Iowa State Univ., 1997.

[DD95a] Jin Song Dong, Roger Duke: *The Geometry of Object Containment*; 41–63 in *Object Oriented Systems* **2**; CHampman & Hall 1995.

[DD95b] Jin Song Dong, Roger Duke: *Exclusive Control within Object Oriented Systems*; 123–132 in *TOOLS Pacific'95*; Prentice Hall 1995.

[DE97] Sophia Drossopoulou, Susan Eisenbach: *Java is Type Safe — Probably*; 387–418 in *ECOOP'97*; LNCS 1241; Springer 1997.

[EKW92] David W Emley, Barry D Kurtz, Scott N Woodfield: *Object-Oriented Systems Analysis: A Model-Driven Approach*; Yourdon Press 1992.

[WF94] A K Wright, M Felleisen: *A syntactic approach to type soundness*; 38–94 in *Information and Computation* **115**; 1994.

[FM90] J Fiadeiro, T Maibaum: *Describing, Structuring and Implementing Objects*; *Foundations of Object-Oriented Languages*; LNCS 489; Springer 1991.

[FFA99] Jeffrey S Foster, Manuel Fähndrich, Alexander Aiken: *A Theory of Type Qualifiers*; *PLDI'98;SIGPLAN Notices* **34**(5); ACM 1999.

[Ga+95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: *Design Patterns: Elements of Resuable Object-Oriented Software*; Addison-Wesley 1995.

[GM93] Dipayan Gangopadhyay, S Mitra: *ObjChart: Tangible Specification of Reactive Object Behavior*; *ECOOP'93*; LNCS 707; Springer 1993.

[GH75] John D Gannon, J J Horning: *Language Design for Programming Reliability*; *Trans. SE* **1**(2); IEEE 1975.

[GM97] Andreas Gawecki, Florian Matthes: *Integrating Subtyping, Matching and Type Quantification: A Practical Perspective*; 26–47 in *ECOOP'97*; LNCS 1241; Springer 1997.

[GTZ98] Daniela Genius, Martin Trapp, Wolf Zimmermann: *An Approach to Improve Locality Using Sandwich Types*; *Types In Compilation'98*; LNCS 1473; Springer 1998.

[GP95] Peter Gerstl, Simone Pribbenow: *Midwinters, end games, and body parts: a classification of part-whole relations*; 865–889 in *Int. J. Human-Computer Studies* **43**; Academic 1995.

[GL95] Joseph Gil, David H Lorenz: *Environmental Acquisition – A New Inheritance-Like Abstraction Mechanism*; TR LPCR-9507; Technion, Israel Institute of Technology, Haifa 1995. A shorter version appeared in *OOPSLA'96*; ACM 1996.

[GR83] Adele Goldberg, David Robson: *Smalltalk-80: The Language and its Implementation*; Addison-Wesley 1983.

[Gor00] Mike Gordon: *Christopher Strachey: Recollections of His Influence*; 65–67 in *Higher-Order and Symbolic Computation* **13**; Kluwer 2000.

[GJS00] James Gosling, Bill Joy, Guy Steele, Gilad Bracha: *The Java Language Specification* (2*nd.* ed.); Sun Microsystems 2000.

[GB99] Aaron Greenhouse, John Boyland: *An Object-Oriented Effects System*; 205–229 in *ECOOP'99*; LNCS 1628; Springer 1999.

[GPV01] Christian Grothoff, Jens Palsberg, Jan Vitek: *Encapsulating Objects with Confined Types*; *OOPSLA'01*; ACM 2001.

[Gun92] Carl A Gunter: *Semantics of programming languages: structures and techniques*; MIT 1992.

[HLS00] Harri Hakonen, Ville Leppänen, Tapio Salakoski: *Object Integrity while Allowing Aliasing*; 91–96 in ICS'2000.

[HGP92] M Halper, J Geller, Y Perl: *An OODB "Part" Relationship Model*; 602–611 in *CIKM'92* ; 1992.

[Ham97] Graham Hamilton (ed.): *JavaBeans* (1.01); Sun Microsystems 1997.

[HG97] David Harel, Eran Gery: *Executable Object Modeling with Statecharts*; 31–42 in *Computer* **30**(7); IEEE 1997. Early version in *ICSE'96*); IEEE 1996.

[HJS92] T Hartmann, R Junghand, G Saake: *Aggregation in a Behavior Oriented Object Model*; *ECOOP'92*; LNCS 615; Springer 1992.

[Hau93] Franz J Hauck: *Inheritance Modeled with Explicit Bindings: An Approach to Typed Inheritance*; *OOPSLA'93*; ACM 1993.

[HM95] Ian J Hayes, Brendan P Mahony: *Using Units of Measurement in Formal Specifications*; 329–347 in *Formal Aspects of Computing* **7**(3); 1995.

[HHG90] R Helm, I M Holland, D Gangopaghyay: *Contracts: specifying behavioral compositions in object-oriented systems*; 169–180 in *OOPSLA/ECOOP'90;SIGPLAN Notices* **25**(10); ACM 1990.

[HB99b] B Henderson-Sellers, F Barbier: *What is this thing called aggregation?*; 236–250 in *TOOLS 29* ; IEEE 1999.

[Hen97] B Henderson-Sellers: *OPEN Relationships: Composition and Containment*; *JOOP* **10**(7); SIGS 1997.

[HHN92] Laurie Hendren, Joseph Hummel, Alexandru Nicolau: *Abstractions for Recursive Pointer Data Structures: Improving the Analysis and Transformation of Imperative Programs*; 249–260 in *PLDI'92;SIGPLAN Notices* **27**(7); ACM 1992.

[Hoa72] C A R Hoare: *Proof of correctness of data representations*; 271–281 in *Acta Informatica* **1**(4); Springer;1972.

[Hog91] John Hogg: *Islands: Aliasing Protection In Object-Oriented Languages*; *OOPSLA'91*; ACM 1991.

[Ho$^+$92] John Hogg, Doug Lea, Alan Wills, Dennis deChampeaux, Richard Holt: *The Geneva Convertion On The Treatment of Object Aliasing*; Follow-up report on ECOOP'91 workshop "Object-Oriented Formal Methods"; 11–15 in *OOPS Messenger* **3**(2); ACM 1992.

[HHN94] Joseph Hummel, Laurie Hendren, Alexandru Nicolau: *A Language for Conveying the Aliasing Properties of Dynamic, Pointer-Based Data Structures*; 208–216 in 8*th.* International Parallel Processing Symposium 1994.

[IPW99] Atsushi Igarishi, Benjamin Pierce, Philip Wadler: *Featherweight Java: A minimal core calculus for Java and GJ*; *OOPSLA'99*; ACM 1999.

[ILE88] Madelyn Anne Iris, Bonnie E Litowitz, Martha Evens: *Problems of the part-whole relation*; 261–288 in Martha Walton Evens (ed.): *Relational models of the lexicon: representing knowledge in semantic networks*; Cambridge 1988.

[ISO98] *Programming Languages – C++*; ISO/IEC 1998.

[ISO95] *Programming Languages – Ada*; ISO/IEC 1995.

[Ja$^+$94] Ivar Jacobson, Magnus Christerson, Patril Jonsson, Gunnar Övergaard: *Object-Oriented Software Engineering*; Addison-Wesley 1994. (first published 1992).

[JO93] Ralph E Johnson, William F Opdyke: *Refactoring and aggregation*; 264–278 in *International Symposium on Object Technologies for Advanced Software*; LNCS 742 ; Springer 1993.

[JHC84] P N Johnson-Laird, D J Herrmann, R Chaffin: *Only Connections: A Critique of Semantic Networks*; 292–315 in *Psychological Bulletin* **96**(2); APA 1984.

[JL76] A K Jones, B H Liskov: *A Language Extension for Controlling Access to Shared Data*; 277–285 in *Software Engineering* **2**(4); IEEE 1976.

[KS92] Gerti Kappel, Michael Schrefl: *Local referential integrity*; 41–61 in *Entity-Relationship Approach'92*; LNCS 645 ; Springer 1992.

[Ken94] Andrew Kennedy: *Dimension types*; 348–362 in in *Programming Languages and Systems ESOP'94*; LNCS 788 ; Springer 1994.

[Ken97] Andrew Kennedy: *Relational parametricity and units of measure*; 442–455 in in *POPL'97*; ACM 1997.

[KM95] S Kent, I Maung: *Encapsulation and Aggregation*; *TOOLS Pacific'95*; Prentice Hall 1995.

[KR94] H Kilov, J Ross: *Information Modeling: An Object-Oriented Approach*; Prentice Hall 1994.

[Ki$^+$87] Won Kim, Jay Banerjee, Hong-Tai Chou, Jorge F Garza, Darrell Woelk: *Composite Object Support in an Object-Oriented Database System*; *OOPSLA'87*; ACM 1987.

[Ki$^+$88] Won Kim, Nat Ballou, Hong-Tai Chou, Jorge F Garza, Darrell Woelk, Jay Banerjee: *Integrating An Object-Oriented Programming System With a Database System*; *OOPSLA'88*; ACM 1988.

[KS93] Nils Klarlund, Michael I Schwartzbach: *Graph Types*; 196–205 in *POPL'93*; ACM 1993.

[Kni96] Günter Kniesel: *Encapsulation = Visibility + Accessibility*; Technical Report IAI-TR-96-12; CS Dept III, Univ. Bonn 1996.

[KT99] Günter Kniesel, Dirk Theisen: *JAC – Java with Transitive Readonly Access Control*; should be in: Object-Oriented Technology: ECOOP'99 Workshop Reader, LNCS 1743, 1999.

[Kni99] G Kniesel: *Type-safe delegation for run-time component adaption*; *ECOOP'99*; LNCS 1628; Springer 1999.

[Kol99] M Kolp: *A Metaobject Protocol for Integrating Full-Fledged Relationships into Reflective Systems*; PhD thesis; INFODOC, Universite Libre de Bruxelles, Belgium 1999.

[Kri94] B B Kristensen: *Complex Associations: Abstractions in Object-Oriented Modeling*; *OOPSLA'94*; ACM 1994.

[KM96] B B Kristensen, Daniel C M May: *Activities: Abstractions for Collective Behavior*; *ECOOP'96*; LNCS 1098; Springer 1996.

[Lam93] John Lamping: *Typing the Specialization Interface*; *OOPSLA'93*; ACM 1993.

[Lam73] B W Lampson: *A Note on the Confinement Problem*; *Comm of the ACM* **16**(10); ACM 1973.

[La+77] B W Lampson, et al.: *Report on the programming language Euclid*; *SIGPLAN Notices* **12**(2); ACM 1977.

[Lea99] Gary T Leavens: *Larch/C++ Reference Manual* (Revision 5.41); Iowa State Univ. 1999.

[LM88] Ole Lehrmann Madsen, Birger Møller-Pedersen: *What Object-Oriented Programming May Be – and What It Does Not Have To Be*; 1–20 in *ECOOP'88*; LNCS 322; Springer 1988.

[Lei95] K Rustan M Leino: *Toward Reliable Modular Programs*; PhD thesis, Technical Report TR-95-03; California Institute of Technology 1995.

[Lei01] K Rustan M Leino: *Extended Static Checking: A Ten-Year Perspective*; 157–175 in *Informatics: 10 Years Back, 10 Years Ahead*; LNCS 2000 ; Springer 2001.

[LN00] K Rustan M Leino, Greg Nelson: *Data abstraction and information hiding*; Research Report 160; Compaq SRC 2000.

[LS97] K Rustan M Leino, Raymie Stata: *Virginity: A contribution to the specification of object-oriented software*; Technical Note 1997-001; DEC SRC 1997.

[LL97] Clarence I Lewis, Cooper H Langford: *Symbolic Logic*; Dover Publications 1932. Reprinted in Irving M Copi, James A Gould: *Contemporary Readings in Logical Theory*; Macmillan 1967.

[LH89] Karl J Lieberherr, Ian Holland: *Assuring good style for object-oriented programs*; 38-48: *IEEE Software* **6**(5); IEEE 1989.

[Lif93] Rainer H Liffers: *Inheritance versus Containment*; *SIGPLAN Notices* **28**(9); ACM 1993.

[LZ75] Barbara H Liskov, Stephen N Zilles: *Specification Techniques for Data Abstractions*; *SE* 1(1); IEEE 1975.

[Lis92] Barbara H Liskov: *A History of* CLU; Technical Report; MIT 1992.

[LW94] Barbara H Liskov, Jeanette M Wing: *A Behavioral Notion of Subtyping*; 1811–1841 in *TOPLAS* **16**(1); ACM 1994.

[Lis88] Barbara H Liskov: *Data Abstractions and Hierarchy*; 17–34 in *SIGPLAN Notices* **25**(5); ACM 1988.

[Liu92] Ling Liu: *Exploring semantics in aggregation hierarchies for object-oriented databases*; 116–125 in *ICDE'92* ; IEEE 1992.

[Lou94] Kenneth C Louden: *Programmiersprachen – Grundlagen, Konzepte, Entwurf*; Thomson 1994. Original title: ¡Programming Languages – Principles and Practice¿ 1993.

[LG88] John M Lucassen, David K Gifford: *Polymorphic effect systems*; *POPL'88*; ACM 1988.

[LV95] David C Luckham, James Vera: *An Event Based Architecture Definition Language*; *Trans. Software Engineering* **21**(9); IEEE 1995.

[Mar96] Robert C Martin: *The Liskov Substitution Principle*; *C++ Report*; SIGS March 1996.

[MA79] J R Mc Graw, G R Andrews: *Access Control in Parallel Programs*; 1–9 in *Software Engineering* **5**(1); IEEE 1979.

[Mey88] Bertrand Meyer: *Object Oriented Software Construction*; Prentice Hall 1988.

[Mez98] Mira Mezini: *Variational Object-Oriented Programming Beyond Classes And Inheritance*; Kluwer 1998.

[Mic02] *MSDN Library*; web-pages; Microsoft 2002. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/com/

[MD95] *The Component Object Model Specification* (0.9); Microsoft, DEC 1995.

[MSI90] Hafdeh Mili, John Sibert, Yoav Intrator: *An Object-Oriented Model Based on Relations*; 139–155 in *The J. of Systems and Software* **12**(2); Elsevier 1990.

[Mi+97] Robin Milner, Mads Tofte, Robert Harper, David MacQueen: *The Definition of Standard ML (revised)*; MIT Press 1997.

[Min96] Naftaly H Minsky: *Towards Alias-Free Pointers*; 189–209 in *ECOOP'96*; LNCS 1098; Springer 1996.

[MZ92] Guido Moerkotte, Andreas Zachmann: *Multiple Substitutability Without Affecting the Taxonomy*; 120–135 in *Advances in Database Technology EDBT'92*; LNCS 580; Springer 1992.

[MC94] Ana M D Moreira, Robert G Clark: *Complex Objects: Aggregates*; TR-CSM-123; Univ. Stirling, Scotland 1994.

[MP01] Peter Müller, Arnd Poetzsch-Heffter: *Universes: A Type System for Alias and Dependency Control*; *Informatik Berichte* 279; Fernuniversität Hagen 2001.

[MP99a] Peter Müller, Arnd Poetzsch-Heffter: *Universes: A Type System for Controlling Representation Exposure*; *Programmiersprachen und Grundlagen der Programmierung, 10. Kolloquium*;*Informatik Berichte* **263**; FernUniversität Hagen 1999/2000.

[MP99b] Peter Müller, Arnd Poetzsch-Heffter: *Modular Specification and Verification Techniques for Object-Oriented Software Components*; G T Leavens, M Sitaraman (ed.): *Foundations of Component-Based Systems*; Cambridge Univ. Press 1999.

[Nic99] Sheldon Nicholi: *Wixins: a new object model*; *JOOP* **12**(7); SIGS 1999.

[Nob99] James Noble: *The Objects of Aliasing*; presented at Intercontinental Workshop on Aliasing in Object Oriented Systems at ECOOP'99.

[NVP98] James Noble, Jan Vitek, John Potter: *Flexible Alias Protection*; 158-185: *ECOOP'98*; LNCS 1445; Springer 1998.

[Ode94] JJ Odell: *Six Different Kinds of Composition*; *JOOP* **5**(8); SIGS 1994.

[Ohe01] David von Oheimb: *Analyzing Java in Isabelle/HOL: Formalization, Type Safety and Hoare Logic*; dissertation; Technische Universität München 2001.

[OMG00] *The Common Object Request Broker: Architecture and Specification* (2.4); OMG 2000.

[OMG00] *OMG Unified Modeling Language Specification* (1.3); OMG 2000.

[OM01] Klaus Ostermann, Mira Mezini: *Object-Oriented Composition Untangled*; *OOPSLA'01*; ACM 2001.

[Par72] D L Parnas: *On the Critera To Be Used in Decomposing Systems into Modules*; *Communications of the ACM* **15**(12); ACM 1972.

[Par94] Chris Partridge: *Modelling the real world: Are classes abstractions or objects?*; *JOOP* **7**(7); SIGS 1994.

[PNC98] John Potter, James Noble, David Clarke: *The Ins and Outs of Objects*; presented at ¡ASWEC'98¿ (Adelaide); 1998.

[Pre97] Wolfgang Pree: *Komponentenbasierte Softwareentwicklung mit Frameworks*; dpunkt 1997.

[Pun97] Franz Puntigam: *Coordination Requirements Expressed in Types for Active Objects*; *ECOOP'97*; LNCS 1241; Springer 1997.

[Qui95] Klaus Quibeldey-Cirkel: *Das Objekt, Paradigma in der Informatik*; B G Teubner 1994.

[RBF98] D Ramazani, G v Bochmann, P Flocchini: *Object Naming and Object Composition*; Publication #1135; Département d'informatique et de recherche opérationnelle, Université de Montréal Novembre 1998.

[RC00] Derek Rayside, Gerard T Campbell: *An Aristotelian Understanding of Object-Oriented Programming*; *OOPSLA'00*; ACM 2000.

[Rey94] John C Reynolds: *User-Defined Types and Procedural Data Structures as Complementary Approaches to Data Abstraction*; Carl A Gunter, John C Mitchell (ed.): *Theoretical Aspects of Object-oriented Programming: types, semantics, and language design*; MIT Press 1994. First published 1975.

[Rey78] John C Reynolds: *Syntactic control of interference*; *POPL'78*; ACM 1978.

[RL00] Clyde Ruby, Gary T Leavens: *Safely Creating Correct Subclasses without Seeing Superclass Code*; 208–228 in *OOPSLA'00*; ACM 2000.

[Rum94a] James Rumbaugh: *Virtual worlds: Modeling at different levels of abstraction*; *JOOP* **6**(8); SIGS 1994.

[Rum94c] James Rumbaugh: *Building boxes: Composite objects*; *JOOP* **7**(7); SIGS 1994.

[Rum95] James Rumbaugh: *Taking things in context: Using composites to build models*; *JOOP* **8**(7); SIGS 1995.

[Rum97] James Rumbaugh: *OO Myths: Assumptions from a language view*; *JOOP* **9**(9); SIGS 1997.

[Ru+91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, William Lorensen: *Object-Oriented Modeling and Design*; Prentice-Hall 1991.

[SFL98] Monika Saksena, Robert B France, María M Larrondo-Petrie: *A Characterization of Aggregation*; International Journal of Computer Systems Science and Engineering; 1998.

[Sat95] Ulrike Sattler: *A concept language for an engieering application with part-whole relations*; 119–123 in *DL-95* International Workshop on Description Logics; Universita di Roma 1995.

[SIP97] Secure Internet Programming Group: *HotJava 1.0 Signature Bug*; web-page; Princeton Univ. 1997. http://www.cs.princeton.edu/sip/news/april29.html

[SPL98] Linda M Seiter, Jens Palsberg, Karl J Lieberherr: *Evolution of Object Behavior using Context Relations*; 79–92 in *Transactions on Software Engineering* **24**(1); IEEE 1998.

[Set97] Ravi Sethi: *Programming Languages: Concepts & Constructs* (2*nd.* ed.); Addison-Wesley 1997.

[SG96] Mary Shaw, David Garlan: *Software Architecture: perspectives on an emerging discipline*; Prentice-Hall 1996.

[Sim95] A J H Simons: *Rationalising Eiffel's Type System*; C Mingins, R Duke, B Meyer (ed.): *TOOLS'95*; Prentice-Hall 1995.

[Sim87] Peter Simons: *Parts: a study in ontology*; Oxford 1987.

[SS77] J M Smith, D C Smith: *Database abstaction: Aggregation and generalization*; 105-133: *Transactions on database systems* **2**(2); ACM 1977.

[Sny86] Alan Snyder: *Encapsulation and Inheritance in Object-Oriented Programming Languages*; 38–45 in *OOPSLA'86*; ACM 1986.

[Sny93] Alan Snyder: *The Essence of Objects: Concepts and Terms*; 31–42 in *IEEE Software* **10**(1); IEEE 1993.

[Som95] Ian Sommerville: *Software Engineering* (5*th.* ed.); Addison-Wesley 1995.

[SNH95] Dilip Soni, Robert L. Nord, Christine Hofmeister: *Software Architecture in Industrial Applications*; 196–207 in ICSE'95, IEEE 1995.

[Sta97] Raymie Stata: *Modularity in the Presence of Subclassing*; Research Report 145; DEC/SRC 1997.

[SB85] M Stefik, D G Bobrow: *Object-oriented programming: themes and variations*; 40–62 in *The AI Magazine* **6**(4); 1985.

[SM95] Patrick Steyaert, Wolfgang De Meuter: *A Marriage of Class- and Object-Based Inheritance Without Unwanted Children*; 127–144 in *ECOOP'95*; LNCS 952; Springer 1995.

[St+96] Patrick Steyaert, Carine Lucas, Kim Mens, Theo D'Hondt: *Reuse Contracts: Managing the Evolution of Resuable Assets*; 268–285 in *OOPSLA'96*; ACM 1996.

[Str94] Bjarne Stroustrup: *The Design And Evolution of C++*; Addison-Wesley 1994.

[SM97] Kevin J Sullivan, Mark Marchukov: *Interface Negotiation and Efficient Reuse: A Relaxed Theory of the Component Object Model*; CS-97-11; Univ. of Virginia 1997.

[Sun00] *Java 2 Platform API Specification* (Standard Edition, v 1.3); Sun Microsystems 2000.

[Sym97] D Syme: *Proving JavaS Type Soundness*; Tech. Report; Comp. Lab., University of Cambridge 1997.

[Szy92] Clemens A Szyperski: *Import is not inheritance; why we need both: Modules and classes*; 19–32 in *ECOOP'92*; LNCS 615; Springer 1992.

[Tai96] A Taivalsaari: *On the Notion of Inheritance*; 439–479 in *Computing Surveys* **28**(3); ACM 1996.

[TT94] Mads Tofte, Jean-Pierre Talpin: *Implementing the call-by-calue lambda-calculus using a stack of regions*; 188–201 in *POPL'94*; ACM 1994.

[US87] David Ungar, Randall B Smith: *Self: The Power of Simplicity*; 227–241 in *OOPSLA'87*; ACM 1987.

[Utt96] Mark Utting: *Reasoning about aliasing*; TR 96-37; Software Verification Research Center, Univ. of Queensland 1996.

[Utt92] M Utting: *An Object-Oriented Refinement Calculus with Modular Reasoning*; PhD thesis; University of New South Wales 1992.

[Var96] Achille C Varzi: *Parts, Wholes, and Part-Whole Relations: The Prospects of Mereotopology*; 259—86 in *Data and Knowledge Engineering* **20**; 1996.

[VMO99] S Vauttier, M Magan, C Oussalah: *Extended Specification of Composite Objects in UML*; *JOOP* **12**(2); SIGS 1999.

[VB99] Jan Vitek, Boris Bokowski: *Confined Types*; 82–96 in *OOPSLA'99*; ACM 1999.

[VCCH99] T VonEicken, C-C Chang, G Czajkowski, C Hawblitzel: *J-Kernel: A Capability-Based Operating System for Java*; 369–394 in LNCS 1603; Springer 1999.

[Wad90] Philip Wadler: *Linear Types can change the world!*; *Programming Concepts and Methods*; Elsevier, North-Holland 1990.

[WM00] David Walker, Greg Morrisett: *Alias Types for Recursive Data Structures*; Int'l Workshop on Types in Compilation, Montreal, Canada, Sep. 2000.

[Weg90] Peter Wegner: *Concepts and Paradigms of Object-Oriented Programming*; 7–87 in *OOPS Messenger* **1**(1); ACM 1990.

[Wil92] Alan Cameron Wills: *Formal Methods applied to Object-Oriented Programming*; PhD thesis; Univ. Manchester 1992.

[WH87] Morton E Winston, Douglas Herrmann: *A Taxonomy of Part-Whole Relations*; 417–444 in *Cognitive Science* **11**; Ablex 1987.

[Wir83] Niklaus Wirth: *Programming in Modula-2* (2*nd*. ed.); Springer 1983.

[WB+95] Murray Wood, Andrew Brooks, James Miller, Marc Roper: *Empirical Evaluation of Software Quality Attributes*; EFoCS-9-95; Univ. of Strathclyde, Glasgow 1995. www.cs.strath.ac.uk/CS/Research/EFOCS/Research-Reports/EFoCS-9-95.ps.Z