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

CENTRE FOR NEWFOUNDLAND STUDIES

TOTAL OF 10 PAGES ONLY
MAY BE XEROXED

(Without Author's Permission)

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
NOTICE:
The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author’s permission.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

AVIS:
L’auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l’Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L’auteur conserve la propriété du droit d’auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n’y aura aucun contenu manquant.
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

Abstract ii

Contents iv

List of Figures vii

List of Symbols ix

1 Introduction 1
  1.1 Summary 1
  1.2 Contributions 12
  1.3 Outline 14

2 Abstraction in Object-Oriented Programming 15
  2.1 The Importance of Abstraction 15
  2.2 Abstraction Hierarchies 18
  2.3 Object-Oriented View of the Runtime System 20
  2.4 Complexity in the Large in Object Systems 23
  2.5 Composite Objects and Structured Systems 25
  2.6 Managing Dynamic Complexity: The Map Example 30
  2.7 Origin of the Notion of Composite Object 33

3 Encapsulation in Object-Oriented Programming 36
  3.1 The Importance of Modularity 36
  3.2 Information Hiding and Encapsulation 40
  3.3 The Need to Encapsulate Composite Objects 43
  3.4 Directions of Research in Encapsulation Units 47
  3.5 External Access despite Encapsulation? 52
  3.6 Review of Proposed Encapsulation Policies 54

4 Related Work 59
  4.1 Encapsulation Approaches 59
  4.2 Discussion 68
7.2.1  Syntactic Sugar ........................................... 207
7.2.2  Subclass Polymorphism and Class Inheritance ............ 208
7.2.3  Unlimited Calling? ....................................... 212
7.2.4  More Mutable Modes: Shared, Inside-Out, Borrowed ....... 215
7.3   Some Applications and More Examples ........................ 218
7.3.1  Behavioral Type Checking ................................ 218
7.3.2  Mode/Effects System and the Observer Pattern .......... 220
7.3.3  Domain Modeling: The Car Example ........................ 222
7.3.4  Transfer Across Abstraction Boundaries: The Lexer/Reader Example ............................................. 224
7.3.5  Transfer of Multiple Objects at Once .................... 226
7.3.6  The Builder Pattern: Bottom-Up Creation with Free Fields .... 227

8  Conclusion  .............................................. 229

A  The Definition of JaM ................................... 231
   A.1  Syntactic Structures .................................... 231
   A.2  Type System .............................................. 232
   A.3  Semantic Structures ..................................... 234
   A.4  Small Step Semantics .................................. 235

B  JaM Code of the Map Example .................................. 237
   B.1  In Basic Desugared JaM ................................... 237
   B.2  Refactored with Self-Calls .............................. 241
   B.3  In Sugared Generic JaM .................................. 242

Bibliography ................................................... 246
# List of Figures

1.1 Set-of-objects and node-based realization in JaM .................................. 11

2.1 Space of programming paradigms ................................................................. 16
2.2 Flat, and structured class model of an ATM-banking system ....................... 19
2.3 Managing the complexity of the object system through packages ............... 24
2.4 Managing runtime complexity through composite objects .......................... 25
2.5 Class model of composite object-structured system (adapted from [Kri94]) ...... 27
2.6 Class packaging orthogonal to object composition, with dependencies .......... 28
2.7 Composite object and expansion to elementary objects .............................. 28
2.8 Unstructured lookup collaboration ............................................................... 30
2.9 lookup collaboration structured with composite objects ............................ 31
2.10 lookup collaboration at intermediate level of detail .................................... 33

3.1 Encapsulation models in object-oriented programming ............................... 42
3.2 Encapsulated composite objects ..................................................................... 45
3.3 Nested principal-with-proxies aggregate, nested collective aggregate .......... 52

5.1 Syntax of base-JaM programs ........................................................................ 71
5.2 Program’s meaning as defining object classes ............................................ 73
5.3 Runtime model ............................................................................................... 75
5.4 Handle source consistency ............................................................................ 76
5.5 Semantic values, types, and type-consistency .............................................. 77
5.6 Top-level reduction rules ............................................................................. 79
5.7 Reduction of expression reduces ................................................................. 81
5.8 Reduction of statement reduces .................................................................... 83
5.9 Potential access paths in object graphs labeled with base-modes ................ 88
5.10 Base-JaM integrity invariants ........................................................................ 90
5.11 Composition of composite objects in JaM .................................................. 91
5.12 Legal base-JaM programs ............................................................................ 94
5.13 Typing rules for program terms ................................................................... 96
5.14 Typing rules for runtime terms and consistency with extended context ....... 97
5.15 Mode-specific definitions for base-JaM ...................................................... 100
5.16 Dependency of proven properties ............................................................... 107
# List of Symbols

- $c$: class named $c$
- $o$, $o_2$, $o_3$: object named $o$, object $o$ of class $c$, unnamed object of class $c$
- $x$: variable $x$ with value $v$
- $m()$: message $m()$ sent via object reference
- $\rightarrow$: dependency between classes
- $\rightarrow$: generalization, or Java's `extends` relationship between classes
- $\rightarrow$: realization, or Java's `implements` relationship between classes
- $X \doteq Y$: formulae $X$ and $Y$ are defined and $X = Y$
- $X[y/x]$: substitution of $y$ for $x$ in term, sequence, or set $X$
- $\rightarrow$: partial map
- $\cup$, $\setminus$: multiset-union and multiset-subtraction
- $\epsilon$: empty path, empty sequence of association roles
- $X^*$: set of sequences of $X$'s (Kleene closure)
- $\overline{\mathbf{d}}$, $\overline{\tau x}$: sequences $x_1,\ldots,x_n$ and $\tau_1 x_1,\ldots,\tau_n x_n$
- $\pi_1 \circ \pi_2$, $\overline{\alpha \cdot \gamma}$: concatenation of paths, and of association role sequences
- $\mu_\tau \circ \mu$: adaption of mode $\mu$ imported through $\mu_\tau$
- $\leq_m$: mode compatibility relation
- $\Sigma(c)$, $\Sigma(\mu c)$: signature of $c$-objects, and of $\mu c$-references
- $\alpha, \beta, \gamma \in A$: association roles, and the set of association roles
- $c, d \in C$: class names, and the set of class names
- $e$: expression, runtime term
- $\eta$, $\rho$: method-local and object-local binding environment
- $\Gamma$: type assignment, typing environment
- $g$: object graph
- $h$: object reference value (handle)
- $k$: kind of method
- $\mu \in M$: mode, and the set of valid modes
- $o, \omega, q, u, v, w \in \mathcal{O}$: object identifiers, and the set of object identifiers
- $\pi$: path of object references
- $Sanc(o)$, $StRep(o)$: $o$'s sanctuary and state representation
- $s$: store
- $t, \tau$: 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 sub-objects, 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)](image)

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 $o \in \{f1, f1', f2\}$. 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 \( StRep(o) \subseteq \mathcal{O} \) 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 \( f' \) to 33 or 34 makes it represent, respectively, 2 February 1971 or 3 February 1971. Since \( f' \) 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 \( \{ 1 \text{ February 1971}, 2 \text{ February 1971}, 3 \text{ February 1971} \} \). **Example 2.** Updating the data field of Node object \( n_4 \) to a new Date object \( f_3 \) 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 \( \{ 1 \text{ February 1971}, 2 \text{ February 1971}, 3 \text{ February 1971} \} \), and the set reified by \( S_2 \) to \( \{ f_1, f_1', f_2, f_3 \} \).

This dissertation will ensure that such side-effects of the change of \( f' \) or \( n_4 \) 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 \textit{get-value-of-x} and \textit{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 \textit{S}$_2$ typically return an abstract iterator object which yields one element after another to the client.\footnote{Iterators are an example that objects can not only be data abstractions but also \textit{behavioral} or \textit{process abstractions}: Rather than holding data, iterators reify the client’s iteration process over the data stored in another object, much like a coroutine.} The typical implementation of iterators would be for Set representative \textit{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 \textbf{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 \textit{mutator} or \textit{observer}, and on metaphorically associating each object \textit{o} with a protection domain, the \textit{sanctuary} \textit{Sanc}(o): An object’s fields may only be updated in its own \textit{mutator} methods (\textit{shallow state encapsulation}). And these mutators may be invoked on objects in \textit{o}’s sanctuary only from mutators of \textit{o} and of objects in \textit{o}’s
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 \( \omega \). 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 \( \omega \) in \( o \)'s sanctuary as well as the initiation of mutator executions in \( \omega \) by mutators of \( o \) will be based on certain types of paths \( o \rightarrow \omega \) of object references from \( o \) to \( \omega \) in the current object graph. This dissertation proposes a classification of paths into types called modes \( \mu \in M \). The basic classification is five-fold:
A **rep path** is a path $o \rightarrow w$ which means that $w$ is in $o$'s sanctuary and in all sanctuaries containing $o$, but in no other. The programmer adds $o$'s state-representing components $w$ to $o$'s sanctuary by classifying paths $o \rightarrow w$ from $o$ to $w$ 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 $w$, so that $o$ is $w$'s unique **owner**.

A **free paths** is a path $o \rightarrow w$ meaning that $w$ is in no object’s sanctuary (excluding rep paths to $w$), and that all free paths to $w$ must start with the first reference of $o \rightarrow w$ (so that $o$ is $w$’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 $o \rightarrow w$ which means that $w$ is in the same (nested) sanctuaries as $o$, and that the extension $q \rightarrow o \rightarrow w$ of any path $q \rightarrow o$ of mode $\mu$ by $o \rightarrow w$ is another path of mode $\mu$. 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 $o \rightarrow w$ that has no meaning for $w$'s status and does not extend other paths to a moded path to $w$. All paths which are none of the above are classified as **read**.

Rep and free paths $o \rightarrow w$ are both “ownership paths,” guaranteeing that $o$ is the unique owner of $w$. 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. 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 $w$ have (indirectly) been sent from $o$ to $w$ along one of its rep or free paths $\pi$. That is, if $\pi$ is $o = o_0 \rightarrow o_1 \rightarrow \ldots \rightarrow o_n = w$ 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 $w$.

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 $o \rightarrow o \rightarrow \omega$, association paths of role $\alpha$ represent links $o \rightarrow o \rightarrow \omega$ of public ordinary association $\alpha$, 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 $s \rightarrow n_1$ 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.

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 $n_1$ which returns the co reference $n_1 \rightarrow n_2$, then the reference $s \rightarrow n_2$ which $s$ obtains must not be a co reference, since $s$ cannot be in its own sanctuary $\text{Sanc}(s)$. The return of the co reference can be better understood as the mode-preserving shortening of two-references path $s \rightarrow n_1 \leftarrow n_2$ to a one-reference path $s \rightarrow n_2$: The reference which $s$ obtains is a rep reference $s \rightarrow n_2$. Should, on the other hand, one node $n_1$ call next() on its co-object $n_2$, then the returned reference's mode is not adapted, since the return simply shortens co path $n_1 \leftarrow n_2 \leftarrow n_3$ to $n_1 \leftarrow n_3$.

Analogously, the mode of references passed as parameters has to be adapted: If $s$ has created a new Node object $n_0$ in its sanctuary, then it should supply to $n_0$'s SetNext operation (expecting a co reference) one of its rep references, namely $s \rightarrow n_1$, and not a reference $s \leftarrow n'$ 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 $\mu_r$ from the mode $\mu$ 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 $f : \mu_i d_i \rightarrow \mu d$ 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 $\mu_r$-references to $c$-objects have the signature $\Sigma(\mu_r c)$ with modes from $c$-objects’ signature $\Sigma(c)$ adapted relative to call-link mode $\mu_r$. Its entries $f : \mu_r \circ \mu_i d_i \rightarrow \mu_r \circ \mu d$ 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 $\mu_r c$, are type-checked.

This adaption is comparable in C++ to the signature $\Sigma(\text{const } c)$ 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 $T$ of each field to const $T$.

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 $ni \rightarrow fj$ 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 $ni \rightarrow fj$ must be classified as association references extending $s$’s paths to the nodes to paths which can extend reference $s' \rightarrow s$ to paths $s' \Rightarrow f_1, f'_1, f_2$ 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 $\alpha \in A$ 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 \( m \in B = \{ \text{free, rep, co, read} \} \cup \mathcal{A} \) 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 \( \alpha_i = \mu_i \) that specify that the extensions of \( \mu \)-paths by \( \alpha_i \)-paths have mode \( \mu_i \). Syntactically, a full mode thus has the general form \( m<\alpha_1=\mu_1, \ldots, \alpha_n=\mu_n> \).

In the example, the nodes' data-references could be given association mode \( \text{data}<> \) and \( s \)'s anchor reference to the first node the mode \( \text{rep}<\text{data}=<\text{elem}<> \rangle \), so that \( s \)'s reference paths \( s \longrightarrow f_1, f_1', f_2 \) to the objects in the nodes have association mode \( \text{elem}<> \). These paths represent the \( \text{elem} \) 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 \( \text{rep}<\text{elem}=\text{rep}<> \rangle \), its extensions \( s' \longrightarrow f_1, f_1', f_2 \) by \( s \)'s \( \text{elem} \) paths are given the desired mode \( \text{rep} \).

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 \( \text{Node} \) class with data references of mode \( \text{data}<> \) could also be used in the date-set composite \( C_1 \) instead of those with \( \text{co}<> \) data (cf. paragraph 6): Only change the mode of \( s \)'s anchor reference to \( \text{rep}<\text{data}=\text{rep}<> \rangle \).

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 \texttt{mut} and \texttt{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, \texttt{mut} and \texttt{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 \( \text{stored} \) in a variable (field, local variable, parameter) have their modes fixed to the mode \( \mu \) which qualifies the class name \( c \) in the reference type \( c \in \mu \) declared as the variable’s range. Second, the temporary reference \( o \rightarrow \omega \) which the sender \( o \) obtains when the receiver \( q \) returned reference \( q \rightarrow \omega \) has the mode \( \mu_r \circ \mu \) that is an adaption of \( \mu \) relative to the mode \( \mu_r \) of the reference \( o \rightarrow q \) 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 (\texttt{new}) and to \texttt{null}.
interface Set {
    obs boolean contains(read<> Object o);
    mut void Add(elem<> Object o);
    mut void Remove(elem<> Object o);
}

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 \( \mu \)-path \( o \rightarrow q \) by a co- or \( \alpha \)-path \( q \rightarrow \omega \) have, respectively, the mode \( \mu \) or the mode \( \mu' \) if \( \mu = m<..., \alpha=\mu',...> \).

In analogy to this, JaM’s typing rules infer, besides the target class \( c \), the modes \( \mu \) 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 \( \mu_r c \) is the result type \( \mu_r \circ \mu d \) of the corresponding operation in the signature \( \Sigma(\mu_r c) \) of call-links of type \( \mu_r c \). Restrictions are imposed by the typing rules on the use of object references as values in order to preserve the properties of \texttt{rep} and \texttt{free} paths which entailed the safety of permitting mutator calls (as described further below): Object references assigned to \( \mu \)-variables must have a \textit{compatible} mode \( \mu' \leq_m \mu \). Object references supplied as actual parameter to operations with formal parameter mode \( \mu \) in the signature \( \Sigma(\mu_r c) \) of call-links of type \( \mu_r c \) must have a compatible mode \( \mu' \leq_m \mu \). (Simplified, \texttt{free} mode \texttt{free}<\delta> is compatible to any mode \( m<\delta> \), any mode \( m<\delta> \) is compatible to the \texttt{read} mode \texttt{read}<\delta> with the same correlations, and \texttt{read} modes are compatible to \texttt{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 \texttt{mut}-methods, assignments to the fields of \texttt{this} and mutator invocations through references of base-modes \texttt{rep}, \texttt{free}, and \texttt{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, CROO] 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 \(\mu\) 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 explains 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 Scalene; 
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 ab­
straction in object-oriented programming. It may be skipped by readers already fa­
miliar 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 com­
posite objects with objects as components will be used for structuring the system into a hierarchy of nested objects. The foundational data/behavior dualism of (com­
posite) 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.
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/operator 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 [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 ‘∈’ and class-inclusion ‘⊂’ 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 subclasing 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 [DB96]. 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 [Ame97, 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” [Ca89, 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 hierarchical 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: DICHTOMIC 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 fashion: 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 \(o \in \mathcal{O}\) (assigned to it when it is created). If an object (identified by) \(o\) has among the values in its fields or methods the identifier \(\omega \in \mathcal{O}\) of another object, then \(o\) is said to have, at that moment, an object reference (link, handle, pointer) to \(\omega\), in symbols, \(o \dashrightarrow \omega\). In this reference, \(o\) is called the source and \(\omega\) 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 stor­
age 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”

2The 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.
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 all-
encompassing 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, OMOl] 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 sub-objects 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
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.
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 implement-
ing 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 \( CState(o) \). In this state, \( o \) can represent abstract data to implement a data abstraction. To \( CState(o) \) belong the states \( state(\ell) \) of the composite’s fields \( \ell \in \text{flds}(o) \) as well as the states \( CState(\omega) \) of certain components \( \omega \) of the composite which are accordingly called its state-representing components \( \omega \in StCmp(o) \). In short, the composite object \( o \)’s current state \( CState(o) \) is some kind of union of its fields’ and current state-representing components’ states:

\[
CState(o) = \bigcup_{\ell \in \text{flds}(o)} state(\ell) \cup \bigcup_{\omega \in \text{StCmp}(o)} CState(\omega)
\]

It is crucial for the power of composite objects (over elementary objects) and of the object composition technique (over inheritance) that in \( CState(o) \) not only each field and component’s state \( state(\ell) \) and \( CState(\omega) \) can change, but that also the set \( StCmp(o) \) of state-representing components is able to change dynamically as needed (unlike the set \( \text{flds}(o) \) 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 \text{MapImp} \ of an abstract \text{Map} data structure, i.e., a variable mapping from key objects to value objects, may represent \text{Map} states by storing each key:value pair of the map in a variable number of component objects of class \text{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 \texttt{lookup} operation on the abstract \text{Map} data structure that will be elaborated in detail in the next section. For \texttt{lookup}, a \text{MapImp} composite \( d \) has to iterate over its entry components of class \text{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 \texttt{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 \texttt{lookup} operation allows no change of \( d \)’s state \( CState(d) \), whereas Iterator \( i \) must change to progress the iteration during \texttt{lookup}. Hence \( i \) can only be a behavioral component of \( d \).
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 constituting 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 between 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 e3 are stored. The nodes belong to s, an object of implementation class Setlmp, 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 Maplmp 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 Nodelt 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. Maplmp 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 Nodelt 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).
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 it 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 o refines the part and 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 $\omega$ 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, Li93, 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 functioning 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 compu-
mentation. 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 adaptation 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 *insta-
ble* 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 program-
mers [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 struc-
ture 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 partition-
ing 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 run-
time context $S_O$ in which it is used. It can be transferred 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 im-
plementation.

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 transferred
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

39
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 interferes 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].
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 \textit{(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 \textit{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 as­
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

\begin{enumerate}
\item \textbf{Insufficiency: 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 \textit{“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’\textsc{’}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 vari­
ables. 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 [La77], or effects systems’ regions [LG88]). It was also employed for safe parallelization of programs with pointer data structures [HNN92, KS93, HNH94] and for optimized memory deallocation (linear types [Wad90, Bak95], regions [TT94], escape analysis [Bla99], calculus of capabilities [CWM99], alias types [WMOO]).

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
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 insufficient 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 encapsulation 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 \( \text{Object} \). 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 [GPVO01]: 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 \( w \) 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 \( w \) 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 \( w \) (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 \( \text{SetImp} \) objects, which create iterators over their elements (the Abstract Factory design pattern). The iterators from the \( \text{SetImp} \) object \( s \) used as the entry-set in a \( \text{MapImp} \) composite needs to be wrapped in a \( \text{FirstIt} \) 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 \textit{elements} polymorphic with class parameter \( T \). But since \( T = \text{FirstIt} \) is in a different package than \( \text{SetImp} \) (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 \( \text{AParser} \) 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 \( \text{Scanner<AParser>} \). But then no parser implementation is possible where the scanner object is a component of one of the \( \text{AParser} \)'s subcomponents. Also parser implementations in a different package than \( \text{AParser} \) 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 \textit{islands} [Hog91] or \textit{balloons} [Alm97].

A variation is to distinguish those reference fields whose target we want to be private, \textit{component fields}, from normal fields. Historically, this idea was implemented first without encapsulation, e.g., in the object-based KI system \textit{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 $\downarrow_1 \uparrow$, and $\oplus$ 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.\footnote{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 $w$ in it, means to create a clone of $w$. It is however not clear if also the reading of the this variable in methods of $\omega$ inherited from non-expanded superclasses creates a clone.}

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 \texttt{Set} with an internal, dynamically-growing, cyclically-linked storage structure would be impossible to encapsulate.

3. \textbf{REAL COMPOSITE OBJECT ENCAPSULATION} establishes an encapsulation barrier around all the fields \textit{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 \textit{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 \textit{composition paths} whose final object is private to the initial object. This seems to be the unspoken idea behind \textit{flexible alias protection} [NVP98].

b) Also the class qualification approach can be developed further by assuming \textit{for every object $o$, a generic, ad-hoc subclass $c_\circ$ of any class $c$ all of whose instances are private to $o$}. Such classes have been described as $o$'s \textit{local classes} [KS92], classes or object types with (main) ownership parameter $o$ (\textit{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 \textit{local store} [Utt96], a so-called rep context (providing "a nested partitioning of the object store") [CPN98, Cla01],
"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 \( \omega \) "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 \( \omega \) have an implicit attribute, called their owner, to which they are private if it is non-null. For example, the component objects of \( \text{COM} \) ("inner objects") have an implicit owner attribute ("outer object"), which they return when asked for their \texttt{Unknown} 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 \texttt{Unknown} 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 \( \omega \) are destroyed, \( \omega \) 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 \texttt{transfer} [Utt96] or \texttt{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 \texttt{free}, and \texttt{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 $D_k$, 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 $\omega$ [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
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 $\mathcal{D}_\mathcal{L}$ [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 $e_1$ through $e_3$, could also be created in $D_2$, or in an extra child space $D_4$ 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

strict hiding $\Rightarrow$ strict encapsulation

relaxed hiding $\Rightarrow$ absolute encapsulation

strict hiding $\Rightarrow$ absolute encapsulation

52
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.

```plaintext
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, LeiOl]. 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 minimal 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 remaining 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 if 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* [ClaOl]. *Islands* [Hog91] and *Balloons* [Alm97] are a special form of Sandwiches without outbound 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 in 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 $\omega$ 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 $\omega$ having a notification message sent by $q$ if this would not give $\omega$ the opportunity of changing its state. But then there is a problem with registering a nested component object $\omega$ with an external event-dispatcher or Subject: $q$’s sending of the notifying mutator messages along an inbound reference would circumvent $\omega$’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. 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 inferring 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 distinction 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 [Ki87, Ki88], Sina [AW92], etc.), Kent and Maung’s object composition was represented by a hidden owner attribute in each object set at creation time. References were classified uniformly 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 hashtables 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 τ = Pair<ω:o, o1, o2>, which encodes o as ω’s owner and o1 and o2 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 ⇢ o, fst ⇢ o1, snd ⇢ o2. With this substitution σω, all ownership types t’ in class Pair are interpreted relative to ω as ownership structures τ’ = σω(t’). An object ω may be targeted from different objects q1, ..., qn by references typed with different static ownership types t1, ..., tn. But all their source-relative interpretations σq1(t1) must yield ω’s dynamic type T.

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

²Actually, 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.

³In 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 **ownershi­
ship 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 calcu­
lus, 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 encapsu­
lation") 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 limitation 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 de­
sired, 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 $\alpha$ bound to object $o$ grants the class’s instance the right to possess an $\alpha$-reference to an object targeted by $o$’s owned references and other objects’ $\beta$-references with $\beta$ 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 uncaptureable 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 limitation 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 [Kn96] 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 arith-
metics. 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 [BoyOl]. 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 α ∈ 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. $D \in D := \text{class } C \{ (T \text{Id})^* \text{Mth}^* \}$
method $\text{Mth} := \mathcal{K} \text{T Id}((T \text{Id})^*) \{ (T \text{Id})^* \text{S} \}$
method kind $\kappa \in K := \text{mut} \mid \text{obs}$
type term $t \in T := M \ C$
base-mode $\mu \in \mathcal{M} := \text{free} \mid \text{rep} \mid \text{co} \mid \text{read}$
statements $s \in S := SS \mid N = E; \mid \text{return } E; \mid \text{if}(E \Psi E)\{S\} \mid \text{while}(E \Psi E)\{S\}$
relational op. $\psi \in \Psi := == \mid !=$
expression $e \in E := \text{val}(N) \mid \text{destval}(N) \mid \text{null} \mid \text{new } C() \mid E \triangleq \text{Id}(E^*)$
variable $\nu \in N := \text{Id} \mid \text{this.Id}$

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

Figure 5.1: Syntax of base-JaM programs

(new), or an operation call.\textsuperscript{2} Keywords $\text{val}$ and $\text{destval}$ 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 ‘$\text{val}$’ and ‘$\text{destval}$’ make explicit the, respectively, non-destructive and destructive read access to a variable $\nu$. Destructive access resets the variable to $\text{null}$ after having read the value out of it. Non-destructive access copies the value out of it. In case of a $\text{free}$ reference value, the mode of the copy is weakened to $\text{read}$.

Added ‘$\text{obs}$’ or ‘$\text{mut}$’ 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 $\text{obs}$-methods cannot change non-$\text{free}$ objects’ states. It does not ensure that $\text{mut}$-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 $\text{rep}$, the corresponding reference to an object $\omega$ is defined to mean that $o$ is $\omega$’s owner and includes $\omega$ in its sanctuary. In the execution of legal base-JaM programs, it is ensured that $\omega$ has no other owner (the Unique Owner property).

\textsuperscript{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 \( \omega \)'s owner and that \( \omega \) is in no sanctuary. In legal base-JaM programs, it is ensured that \( \omega \) has no other owner and indeed belongs to no sanctuary, and that no second free reference can target \( \omega \) or start free reference paths to \( \omega \) (the Unique Head property).
• By mode co, the reference is defined to mean that \( \omega \) 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 \( t_i \) 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 \( \text{new } c_n() \_\text{main}() \) 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
reduction semantics. Such a semantics defines the stepwise transformation (reduction, evaluation) of program terms in the context of a stack \( \eta \) of environments for the ongoing method invocations, a store \( s \) 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 \( o \) from the object (identified by) \( o \) to the object (identified by) \( w \) is formalized not simply by the object identifier \( w \) (in \( o \)'s fields or methods) but by the triple \((o, \mu, \omega)\), 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 \( g \) 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 \( FlsMths(c_i) = (\Gamma_i, F_i) \) of the instance record type \( \Gamma_i \) 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 \( \Gamma_i \) is the collection of the names \( x_i \) and range types \( \tau_i \) of the fields defined in \( D_i \) to the type assignment \( \{x_i: \text{ref} \tau_i\} \). (\text{ref} is added to the fields' types since the fields are not \( \tau_i \)-values but \( \tau_i \)-variables, i.e., \( x_i \) denotes a location in the store that contains a \( \tau_i \)-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 \( e, \eta, s, om, g \Rightarrow e', \eta', s', om', g' \) transforming the term \( e \) in the implicit, static context of the program \( p \), and in the dynamic contexts \( \eta, s, om, g \) (environment stack, store, object-map, object graph). It starts with the start-up expression \( e_0 =_{def} \text{new } c_n().\text{main()} \) in the initial contexts \( \eta_0, s_0, om_0, g_0 =_{def} \emptyset, O^{\text{obs}}, \emptyset, \emptyset \):

\[
\begin{align*}
\text{new } c_n().\text{main()}, O^{\text{obs}} & \Rightarrow e_1, \eta_1, s_1, om_1, g_1 \\
& \Rightarrow e_2, \eta_2, s_2, om_2, g_2 \\
& \Rightarrow \ldots
\end{align*}
\]
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** \( s \) is an abstract model of the current memory state which maps locations \( \ell \in \text{Loc} \) (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 \( x \in \text{Id} \) of local variables valid in term \( e \) are mapped by the **environment** \( \eta \) to the store locations \( \ell \) 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 \( x \in \text{Id} \) 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” \( g o \) 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 \( \text{red} e x \), by another term. In particular, locations \( \ell \in \text{Loc} \) are substituted for identifiers \( x \) (using \( \eta \)) and for field names \( \text{this} . x \) (using \( \eta \text{this} \)) as “l-values”, variables’ values \( v \in \mathcal{V} \) are substituted for read access expressions (using \( s \)) as corresponding “r-values”, and method bodies are substituted for operation call expressions (using \( \text{om} \)). 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 occurrences 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 \( \eta_i \) 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** \( \eta_i \in \text{Env} \) handles the identifiers at each method invocation nesting level. Formally, this stack is a sequence \( \eta_1, \ldots, \eta_n \) with \( \eta_1 \) as the bottom and \( \eta_n \) as the top element. The extension of \( \vec{\eta} \) by a new top (or, in the context rules, by a to-be-discarded bottom) \( \eta' \) will be written \( \vec{\eta} \cdot \eta' \)
environment \( \eta \in Env \) = \( Id \mapsto Loc \times K \times V \)
store \( s \in Store \) = \( Loc \mapsto V \)
object-map \( om \in Omap \) = \( O \mapsto ((Id \mapsto Loc) \times (Id \mapsto Mth)) \)
object graph \( g \in Graph \) = \( N^{O \times M \times O} \)
runtime term \( e \in R \) ::= \( R \mid R=R; \mid \text{if}(R \Psi R)\{S\} \mid \text{while}(E \Psi E)\{S\} \mid N \mid \text{val}(R) \mid \text{destval}(R) \mid \text{null} \mid \text{new} \mid C() \mid R \leftarrow Id(R^*) \mid Loc \mid V \mid \ll R \gg \)

with \( Loc, V \) from fig. 5.5; \( S, E, N, \Psi, C, Id \) from program syntax

Figure 5.3: Runtime model

(or \( \eta' \cdot \eta \)), using standard sequence concatenation ‘\( \cdot \)’. In order to formalize the integrity property Mutator Control (Path) (§5.3.2), actual environments \( \eta \) are extended to \( \eta \kappa \) by annotating them with the corresponding method’s kind \( \kappa \) and the call-link \( h \in V \) 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 \( \text{store} s \in Store \) maps locations \( \ell \in Loc \) to the values \( v \in V \) currently at these locations. In base-JaM, these values are always “handles,” the formalization of object references introduced below.

3. A growing \( \text{object-map} om \in Omap \) that maps identifiers \( o \in O \) of created objects to object “values”: a field environment \( g_o \) (mapping field names to locations), and a method suite \( F_o \) (mapping operation names to methods). ²

Additionally, the reduction rules update in parallel an object graph \( g \in Graph \) 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 \textit{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. \text{JAM’s Formalization of Reference Values}. Values—more precisely, \textit{first-class values}—are those things to which expressions can evaluate \textit{and} which can be stored in variables \textit{and} passed as parameters and results (and which are classified

²om’s graph can be understood as a \textit{set of implementation objects} formalized as triples \( \langle \rho, g_o, F_o \rangle \).
by the types \( t \) in the program). Java has \textit{primitive values} of boolean and numeric types, and \textit{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 \textbf{object identifier}: Each time a new object is created, a fresh identifier \( \omega \) is drawn from a given set \( \mathcal{O} \), and used henceforth to refer to that object. 4 And the notion of a null reference (denoted by \textit{null}) is formalized by the special value \( \text{null} \) that does not identify any object.

The base-JaM semantics uses an extended formalization of object references as so-called \textbf{handles}: A handle is not just the object-identifier \( \omega \) of the referred-to object (the reference’s \textit{target}), but a triple \( h = (o, \mu, \omega) \) which includes also the identifier \( o \) of the referring object (the reference’s \textit{source}) and the mode \( \mu \) of \( o \)’s reference to \( \omega \). This extension will simplify to specify which object graph edges \( o \xrightarrow{L} \omega \) 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 \( \ell = g_0(x) \) of fields \( x \) of object \( o \), we expect to find only handles \( s(\ell) = h \) whose source is \( o \). Then the object-map is source consistent, in symbols, \( \models_s \text{om} \). This is defined formally in figure 5.4. Analogously, at locations \( \ell = \eta_h(x) \) of local variables and parameters \( x \) in environments \( \eta_h \) of invocations with receiver \( r \), we expect to find only handles \( s(\ell) = h \) whose source is \( r \). Then the object-map is source consistent, \( \models_s \eta \). And at all method nesting levels in the runtime term \( e \) with corresponding receiver \( r \), we expect to find only handles \( h \) with source \( r \), and locations \( \ell \) containing handles \( s(\ell) = h \) with source \( r \). If this is the case then the runtime term is source consistent, in symbols, \( \models_s \eta \). In figure 5.4, this is defined inductively from the outermost method nesting (corresponding to the

---

4An object value \( \langle \omega, \mu, \omega \rangle \) 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: \( \langle \varnothing, \varnothing \rangle \). But then \texttt{new c()} == \texttt{new c'()}, contrary to the semantics of Java.
location (l-value) \( \ell \in \mathcal{L} \) = df \( \bigcup_{r \in \mathcal{M} \times \mathcal{C}} \mathcal{L}_r \)

handle (value) \( h \in \mathcal{V} \) = df \( (\mathcal{O} \cup \{nil\}) \times \mathcal{M} \times (\mathcal{O} \cup \{nil\}) \)

object-identifier \( o \in \mathcal{O} \) = df \( \bigcup_{c \in \mathcal{C}} \mathcal{O}_c \)

object value \( \langle \rho, F \rangle \in \mathcal{V} \) = df \( \langle Id \mapsto \mathcal{L} \rangle \times (Id \mapsto \mathcal{M}) \)

infinite countable sets \( \mathcal{O}_c \) given for all \( c \in \mathcal{C} \) and \( \mathcal{L}_r \) for all \( r \in \mathcal{M} \times \mathcal{C} \)

\[
\tau \in \mathbf{T} ::= \text{ref } \mathcal{M} \mathcal{C} [\text{ref } \mu \ c] =_{df} \mathcal{L}_c \mu \ c \\
| \mathcal{M} \mathcal{C} [\mu \ c] =_{df} (\mathcal{O} \cup \{nil\}) \times \{ \mu \} \times (\mathcal{O}_c \cup \{nil\}) \\
| \text{obj } \mathcal{C} [\text{obj } \ c] =_{df} \langle \rho, F \rangle | \top \text{FldsMths}(c) = \langle \Gamma, F \rangle \text{ and } \eta \models \Gamma \\
| \text{Cmd} [\text{Cmd}] =_{df} \{ \epsilon \} \]

\[
\eta \models \Gamma \iff_{df} \text{dom}(\eta) = \text{dom}(\Gamma) \land \forall x \in \text{dom}(\Gamma). \eta(x) \in \llbracket \Gamma(x) \rrbracket \\
\models s \iff_{df} \forall \tau \in \mathcal{M} \times \mathcal{C}, \ell \in \text{dom}(s). \ell \in \mathcal{L}_r \Rightarrow s(\ell) \in \llbracket \tau \rrbracket \\
\models \text{om} \iff_{df} \forall c \in \mathcal{C}, o \in \text{dom}(\text{om}). o \in \mathcal{O}_c \Rightarrow \text{om}(o) \in \llbracket \text{obj } c \rrbracket 
\]

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 \( (\mathcal{O} \cup \{nil\}) \times \mathcal{M} \times (\mathcal{O} \cup \{nil\}) \): Handles with nil \( \not\in \mathcal{O} \) 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 \( \eta_0 \) 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 \( \epsilon \) (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 \( \tau \) from the program syntax are generalized to type terms \( \tau \in \mathbf{T} \) shown in figure 5.5. The extensional interpretation (or denotation) \( \llbracket \tau \rrbracket \) of a type term \( \tau \) is the set of all conceivable values of type \( \tau \).

It is straightforward to define what it means for an environment \( \eta \) as a mathe-
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, \( \varphi \vdash \Gamma \). Both must be defined for the same identifiers \( x_i \), and the environment must assign them semantic values \( \eta(x_i) \) from the set \( \llbracket \tau_i \rrbracket \) denoted by the corresponding type assumption \( x_i : \tau_i \) in \( \Gamma \). Treating \( \Gamma \) as partial mapping, we can write
\[
\eta \vdash \Gamma \iff \text{dom}(\eta) = \text{dom}(\Gamma) \land \forall x \in \text{dom}(\Gamma) \cdot \eta(x) \in \llbracket \Gamma(x) \rrbracket.
\]
The set \( \llbracket \text{obj } c \rrbracket \) of possible object values for instances of class \( c \) consists of those tuples \( \langle \varrho, F \rangle \) where the field environment \( \varrho \) is consistent with class \( c \)'s instance record type \( \Gamma \) (assignment of types to field names), and where the method-suite \( F \) is precisely the one which \( c \) defines for its instances:
\[
\llbracket \text{obj } c \rrbracket = \{ \langle \varrho, F \rangle \mid \vdash \text{FldsMths}(c) = \langle \Gamma, F \rangle \text{ and } \varrho \vdash \Gamma \}.
\]
The set \( \Omega \) of object-identifiers is assumed to be partitioned according to class names \( c \in \mathcal{C} \) into disjoint subsets \( \Omega_c \) reserved as identifiers for instances of class \( c \). The object-map is type-consistent, written \( \models \text{om} \), if it maps object-identifiers in class \( c \)'s partition \( \Omega_c \) only to object values of \( c \)-instances:
\[
\models \text{om} \iff \forall c \in \mathcal{C}, o \in \text{dom}(\text{om}), o \in \Omega_c \Rightarrow \text{om}(o) \in \llbracket \text{obj } c \rrbracket.
\]
Handles \( h \in \mathcal{V} \) are classified into handle types \( \mu c \in \mathcal{M} \times \mathcal{C} \) according to their mode \( \mu \) and their target's object class \( c \). The extensional interpretation \( \llbracket \mu c \rrbracket \) of handle type \( \mu c \) is the set of handles with any object-identifier or nil as source, \( \mu \) as mode, and any object-identifier in partition \( \Omega_c \), or nil, as target.\(^5\)
\[
\llbracket \mu c \rrbracket = \text{at } (\Omega \cup \{\text{nil}\}) \times \{\mu\} \times (\Omega_c \cup \{\text{nil}\}).
\]
The set of store locations \( \ell \in \mathcal{L} \) is assumed to be partitioned into disjoint subsets \( \mathcal{L}_\tau \) according to the type \( \tau \) 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 \( \tau = \mu c \in \mathcal{M} \times \mathcal{C} \). Store \( s \) is type-consistent, written \( \models s \), if it maps locations in each type \( \tau \)'s partition \( \mathcal{L}_\tau \) only to values in type (term) \( \tau \)'s extension \( \llbracket \tau \rrbracket \):
\[
\models s \iff \forall \tau \in \mathcal{M} \times \mathcal{C}, \ell \in \text{dom}(s), \ell \in \mathcal{L}_\tau \Rightarrow s(\ell) \in \llbracket \tau \rrbracket.
\]
Given this organization of the store, variables ranging over \( \tau \)-values are represented in the store at locations \( \ell \in \mathcal{L}_\tau \). Hence the interpretation \( \llbracket \text{ref } \tau \rrbracket \) of the type of \( \tau \)-variables found in type assumptions \( x : \text{ref } \tau \) in \( \Gamma \), is the set \( \mathcal{L}_\tau \).
\[
\llbracket \text{ref } \mu c \rrbracket = \text{at } \mathcal{L}_\mu \mathcal{C}_c
\]
\[\models s, \models \text{om} \text{ can be written as short-hand for } \models s \text{ and } \models \text{om}.
\]
\(^5\)For the inclusion of subclass-polymorphism in JaM (see §7.2.2), it is necessary to clarify that \( \text{obj } c \) is the monomorphic type of the object values of the direct instances of class \( c \) (and not its subclasses), and that \( \Omega_c \) 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 \( \Omega_c \) in the definition of \( \llbracket \mu c \rrbracket \) to the subclass-closure \( \bigcup_{\mu \leq \mu} \Omega_c' \). 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 \( \llbracket \mu c \rrbracket \) to the set \( \bigcup_{\mu' \leq \mu} \{\mu'\} \) of all modes \( \mu' \) mode-compatible to \( \mu \) (see §5.4.2).
5.2.2 Computational Steps

6. Selection of the Redex. The definition of reduction steps \(\bar{e}, \bar{s}, om, g \rightarrow e', \bar{s}', om', g'\) 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 \(\bar{e}, \bar{s}, om, g \rightarrow e', \bar{s}', om', g'\). On the other side is the selection of the substitutable subterm in \(\bar{e}\) 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 \(\mathcal{E}^*\) is a runtime term “with a hole” symbolized by ‘\(\square\)’. A complete runtime term \(\hat{e} = \mathcal{E}^*[e]\) is obtained by filling a runtime term \(\bar{e}\) into the hole, i.e., by substituting \(\bar{e}\) for ‘\(\square\)’. Reduction steps then are written \(\mathcal{E}^*[e], \bar{s}, om, g \rightarrow \mathcal{E}^*[e'], \bar{s}', om', g'\). 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 \(\hat{s}, om, g\) change the same way.

In base-JaM, one rule handles substitution of a redex \(\bar{e}\) at the same method nesting level as \(\hat{e} = \mathcal{E}^*[e]\), while another rule handles substitution across one level of method nesting (see figure 5.6). For these rules we do not need the general, multilevel reduction contexts \(\mathcal{E}^*\), but single-level reduction contexts \(\mathcal{E} \in R^D_1\) that do not increase the inserted term’s method nesting level. (A general reduction context \(\mathcal{E}^*\) with the hole at method nesting level \(n\) corresponds to the nesting of \(n\) single-level reduction contexts \(\mathcal{E}_i \in R^D_1\): \(\mathcal{E}^* = \mathcal{E}_1[\mathcal{E}_2[\ldots[\mathcal{E}_n[\mathcal{E}_n]\ldots[\mathcal{E}_2]\ldots[\mathcal{E}_1]\ldots]]]\).)

1. If \(e\) is a redex reducing \(\bar{e}, \bar{s}, om, g \rightarrow e', \bar{s}', om', g'\), then the same reduction is possible in any single-level reduction context \(\mathcal{E} \in R^D_1\), i.e., from \(\mathcal{E}[e]\) to \(\mathcal{E}[e']\).

2. If \(e\) is reducible \(\bar{e}, \bar{s}, om, g \rightarrow e', \bar{s}', om', g'\), then \(e\)'s nesting in a method inlining \(\hat{e} = \mathcal{E}[e]\) can be reduced to \(\hat{e'} = \mathcal{E}[e']\) if the environment stack is
extended at the bottom by some environment $\eta^K_h$ for the new outer-most nesting level. Additionally, $\hat{e}$ and $\hat{e}'$ can be wrapped in a single-level context $\mathcal{E} \in R_1$.

The role of the reduction contexts $\mathcal{E} \in R_1$ 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 $\text{val}(\hat{e})$, $\text{destval}(\hat{e})$ or $\text{return } \hat{e}$ then the next reduction step must change the only proper subterm $e''$ in the unchanged context of $\mathcal{E} = \text{val}(e^\square)$, $\text{destval}(e^\square)$, or $\text{return } e^\square$ (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 $v_i \in V$ 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 $\epsilon$ and only $s_2$ is left: $\mathcal{E} = e^\square s_2$.
- Before the reduction step that can execute assignment $\hat{e} = e_1 = e_2$; itself, context $\mathcal{E} = e^\square = e_2$; directs the substitution to the left-hand side until $e_1$ is an irreducible location $\ell$. Then context $\mathcal{E} = \ell = e^\square$; 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 $\mathcal{E} = \text{if}(e^\square \Psi e)\{s\}$ and $\mathcal{E}' = \text{if}(h\Psi e^\square)\{s\}$ 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 $\eta^K_h$ only, except for return steps that works with the finished top-level environment $\eta^{K'}_{s,\sigma,r,t}$ and the environment $\eta^K_h$ to which the execution will return.

{var_1} The identifier $x \in \text{Id}$ 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 $\eta$ in annotated $\eta^K_h$ maps $x$. Nothing else changes.

{var_2} A field name $\text{this}.x$ reduces to that location $\ell'$ which is specified for $x$ in the field environment $\varrho$ of the current object, i.e., of the target $\omega$ of the handle $s(\ell)$ at the location $\ell = \eta(\text{this})$ denoted by $\text{this}$ in the top-level environment $\eta^K_h$.

{rdcp} Non-destructive read access $\text{val}(\ell)$ to a location $\ell$ copies the value from the store (at location $\ell$) 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
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 \( \mu[\text{read} / \text{free}] \) for the substitution of read for free in mode \( \mu \).

In base-JaM, substitution merely means to replace \( \mu \) to read and leave other \( \mu \)’s unchanged. In full JaM, it will mean to replace any occurrence of base-mode free in the full mode \( \mu = \mu[\text{read}] \) to read, and besides this leave \( \mu \) unchanged. The object graph transformation will be discussed in §5.3.

\{rd\_de\} Destructive read access \( \text{destval}(\ell) \) evaluates to the value at location \( \ell \), but resets the store at \( \ell \) 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 \( r \) 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 o, and evaluates to an initial, free handle from the current object r (the creator) to the new object o. Being fresh implies in particular that o is neither source nor target of any edge in the object graph. Let \( \{x_i : \text{ref } \mu_i \ c_i\} \) be the instance record type \( \Gamma \) and \( F \) the method suite which class c defines for its instances. Then instantiating c means to take fresh locations \( \ell_i \) of respective types \( \text{ref } \mu_i \ c_i \), initialize them to nil-handles with source o and modes \( \mu_i \), and map o to an object value \( \{\{x_i \mapsto \ell_i\}, F\} \) 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 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 \( \text{ref} \)-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 \( \eta_h \) below the current top-level environment. Then evaluation will continue in environment \( \eta_h \) 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 \( \ell \) and the right-hand side to a value \( \langle o, \mu, \omega \rangle \). It updates the store at \( \ell \) 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 \( E[c] \) will be next in the order of execution.

{if}, {iif} 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 \( \epsilon \) to let execution continue with the statement following the if
Figure 5.8: Reduction of statement redices

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 $e_0, \eta_0, s_0, o_{m0}, g_0 \Rightarrow^* e, \bar{\eta}, s, o_{m}, g$ then

$$\models_s o_{m} \land \models_s \bar{\eta} \land \models_s e$$

**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 $s_0 = \emptyset$ contains no handles, and term $e_0 = \text{new } c(). \text{main()}$ contains no handles and no locations. In the induction step $N \rightarrow N + 1$, reduction $e_0, \eta_0, s_0, o_{m0}, g_0 \Rightarrow^* e_N, \bar{\eta}_N, s_N, o_{mN}, g_N$ is continued $e_N, \bar{\eta}_N, s_N, o_{mN}, g_N \Rightarrow e, \bar{\eta}, s, o_{m}, g$. By induction hypothesis, $\models_{s_N} o_{mN}$ and $\models_{s_N} \bar{\eta}_N$ and $\models_{s_N, \bar{\eta}_N} e_N$. Consider $\models_s o_{m}$ and $\models_s \bar{\eta}$. Reductions with \{var\}, \{vary\}, \{rdep\}, \{null\}, \{if\}/\{if\} and \{wh\} change neither $s$ nor $o_{m}$ nor $\bar{\eta}$; \{ret\} neither adds to $s$ nor $\bar{\eta}$; and \{upd\} and \{rds\} do not change the source of the handle at the updated location. Hence still $\models_s o_{m}$ and $\models_s \bar{\eta}$ in all these cases. In case of \{new\} and \{call\}, $o_{m}$ does not change for old objects, and $\bar{\eta}$ does not change in old environments. $s$ 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 o, or the receiver r, respectively. Hence |=_s om and |=_s r again.

Consider |=_s,N e. In case of a reduction with \{var\}, the redex x in the maximal method nesting depth is replaced to location \(\eta(x)\) from the top-level environment \(\eta_h^n\). Hence \(|=_{s,N} \eta_N^n\) ensures that \(s(\ell)\)'s source is the target r of call-link h in top-level environment \(\eta_h^n\), and thus the right one for a location at maximal nesting depth: \(|=_{s,N} e\). And in case of \{var\}, we have a handle \(h = s(\ell) = (o, \mu, o)\) at the location \(\ell = \eta(\text{this})\) of this in top-level environment \(\eta_h^n\). On one hand, \(|=_{s,N} \eta_N^n\) ensures that h’s source o is the target r of call-link h in \(\eta_h^n\). On the other hand, \(|=_{s,N} \eta_m^n\) ensures that the handle \(s(\ell')\) at location \(\ell'\) of o's field x has o as source. Hence the location \(\ell'\) inlined in the runtime term at maximal nesting depth refers to a handle \(s(\ell')\) in the store with the necessary top-level environment’s receiver o = r as source.

In case of \{rdcp\} and \{rdcsts\}, the redices \(\text{val}(\ell)\) and \(\text{destval}(\ell)\) at nesting level \(n\) imply by induction hypothesis that \(s(\ell)\) 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 \{rdcsts\} does update the store, it does not change the source of the handle at location \(\ell\). \(|=_{s,N} e\) is preserved.

In case of \{null\} and \{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 \(|=_{s,N} e\).

Reductions with \{call\}, \{upd\}, \{if\}/\{if\} and \{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 (\{call\}) or without changing the source of the handle at the updated location. Hence still \(|=_{s,N} e\). ■

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 \((e, \eta, s, om)\) is a graph \(g\) 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 \( s \) (stored reference), call-link in \( \eta \) (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 \( s, \eta \), or \( e \). Harder is the removal of edge \( o \xrightarrow{\mu, \omega} \) exactly when handle \( \langle o, \mu, \omega \rangle \) exists nowhere in \( e, \eta \), and \( s \) any more. This can elegantly be handled if the object graph is not formalized as a set \( g \in 2^{O \times M \times O} \) of edges representing the existing of corresponding handles, but as a multiset \( g \in N^{O \times M \times O} \) of edges whose multiplicity represents the number of the corresponding handles’ occurrences in \( s, \eta \), or \( e \): Multiplicities of edges are increased and decreased in accordance with the addition and removal of handles to/from \( e, \eta \) and \( s \), so that the multiplicity of edge \( o \xrightarrow{\mu, \omega} \) in \( g \), written \( mult(o \xrightarrow{\mu, \omega}, g) \), reaches zero (meaning it disappears from the graph) exactly when the last occurrence of \( \langle o, \mu, \omega \rangle \) is removed from \( s, \eta \) and \( e \).

**Definition 1** An object graph is a multiset \( g \in \mathcal{G}raph =_{df} N^{O \times M \times O} \) of directed, mode-labeled edges \( o \xrightarrow{\mu, \omega} \in O \times M \times O \) between two object-identifiers \( o, \omega \in O \) 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 \((e, \eta, s, om)\) as an object graph which contains every edge as often as \( e, \eta \), and \( s \) contain the corresponding handle. It can be constructed from the current configuration by an abstract function ogr:

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

\[
ogr(e, \eta, s) =_{df} \biguplus_{h \in O \times M \times O} \biguplus_{i=1}^{n_h} \{h\}
\]

where \( \bigcup \) is the multiset union that adds up elements’ multiplicities.

The number \( num(h, s) \) of locations at which \( h \) occurs in \( s \in \mathcal{S}tore \) is

\[
num(h, s) =_{df} \{ \ell \in \text{dom}(s) \mid s(\ell) = h \}
\]

85
The number \(\text{num}(h, \eta)\) of environments with call-link \(h\) in stack \(\eta\) (of size \(n\)) is
\[
\text{num}(h, \eta) = \{ i \in \{1, \ldots, n\} \mid \eta_i = h \}
\]
The number \(\text{num}(h, e)\) of occurrences of \(h\) in runtime term \(e \in R\) can be determined inductively as follows:

\[
\begin{align*}
\text{num}(h, x) & = \text{df} 0 \\
\text{num}(h, \text{val}(e)) & = \text{df} \text{num}(h, e) \\
\text{num}(h, \text{destval}(e)) & = \text{df} \text{num}(h, s) \\
\text{num}(h, \text{return}(e)) & = \text{df} \text{num}(h, e) \\
\text{num}(h, \text{if}(\text{Ce}_1 \land \text{Je}_2)) & = \text{df} \text{num}(h, e_1) + \text{num}(h, e_2) \\
\text{num}(h, \text{new}(c)) & = \text{df} 0 \\
\text{num}(h, \text{null}) & = \text{df} 0 \\
\text{num}(h, \text{while}(e)) & = \text{df} \text{num}(h, e) + \text{num}(h, e) \text{f}(e_1, \ldots, e_n) = \text{df} \sum_{i=0}^{n} \text{num}(h, e_i)
\end{align*}
\]

The definition of \(\text{num}(h, e)\) 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 \(g \uplus h\) and removals \(g \ominus h\) 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" \(\uplus\) and "removal" \(\ominus\) used in the semantics are reduced as follows to multiset-union \(\uplus\) and multiset-subtraction \(\ominus\) (which add two multisets' element multiplicities, or subtract the second one's element multiplicities from those of the first one).

\[
g \uplus o \uplus \omega = \begin{cases} g & \text{if } \omega \in \{o, \omega\} \\
g \uplus \{o \uplus \omega\} & \text{otherwise}
\end{cases}
\]
\[
g \ominus o \ominus \omega = \begin{cases} g & \text{if } \omega \in \{o, \omega\} \\
g \ominus \{o \ominus \omega\} & \text{otherwise}
\end{cases}
\]

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

- \{\text{var}\}\}, \{\text{var}\}, \{\text{null}\}, and \{\text{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.

- \{\text{rd}_\text{det}\} leaves the object graph unchanged: The new occurrence of handle \(h = \langle o, \mu, \omega \rangle\) in the term is balanced by removing one occurrence from the store:

\[
\text{num}(h, e') + \text{num}(h, s') = \text{num}(h, e) + \text{num}(h, s).
\]

- \{\text{rd}_\text{cp}\} increases in the graph the multiplicity of the handle \(h = \langle o, \mu', \omega \rangle = o \uplus \omega\) read from the store with substitution of read for free (unless \(o\) or \(\omega\) is nil):

\[
\text{mult}(h, \mu') = \text{mult}(h, \mu) + 1.
\]

This models the redex's substitution to \(h\), which increases the number of \(h\)'s occurrences in the term: \(\text{num}(h, e') = \text{num}(h, e) + 1\).

- \{\text{new}\} adds creator object \(r\)'s initial reference to the new object \(\omega\) to the object graph (except if \(r\) is nil, as in the first step of evaluating \(e_0 \equiv \text{new} e_n() . \text{main}()\)):
\[ g' = g \oplus r \text{ free} \cdot \omega. \] This models the redex's substitution to \( \langle r, \text{ free}, \omega \rangle \).

\{call\} steps equip the receiver with a this reference \( r \overset{\omega}{\to} \) and with a parameter handle \( s \overset{\omega_i}{\to} \) supplied by the sender. That is, the multiplicity of \( r \overset{\omega}{\to} \) and edges \( s \overset{\omega_i}{\to} \) increases, while that of edges \( s \overset{\omega_{ii}}{\to} \) 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 \( o \overset{\omega}{\to} \) is not changed: Its disappearance from the term is balanced by its occurrence in the new top-level environment.

\{ret\} steps combine call-link \( \langle s, \mu_r, r \rangle \) and the edge \( s \overset{\omega}{\to} \) returned by the receiver to the edge \( s \overset{\mu_{p_{s_{\ell}}}}{\to} \) \( \omega \) in the sender, i.e., the former two edge's multiplicity decreases while the latter one's multiplicity increases. This matches the appearance of \( \langle s, \mu_r, \omega \rangle \) in the runtime term and the disappearance of handle \( r \overset{\omega}{\to} \) \( \omega \) from the term and of call-link \( \langle s, \mu_r, r \rangle \) (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 \( o \overset{\omega}{\to} \) \( \omega' \) to \( o \overset{\omega}{\to} \omega \), i.e., decrease the multiplicity of the first handle and increase that of the second one. This matches, respectively, the disappearance of the right-hand side handle \( \langle o, \mu', \omega' \rangle \) from the term and the appearance the handle \( \langle o, \mu, \omega \rangle = o \overset{\omega}{\to} \omega' \) at location \( \ell \) in the store. Additionally, the multiplicity of the old handle \( \langle o, \mu, \omega \rangle = o \overset{\omega}{\to} \omega \) at location \( \ell \) decreases since the update at location \( \ell \) overwrites it.

\{if\} and \{if\} 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 \( e_0, \eta_0, s_0, \omega_0, g_0 \Longrightarrow^* e, \eta, s, \omega, g \) then

\[ g = ogr(e, \eta, s) \]

**Proof by induction on the number \( N \) of reduction steps from \( e_0 \) to \( e' \):** In the base case \( N = 0 \), \( ogr(e_0, \eta_0, s_0) = \emptyset = g_0 \) since term \( e_0 = \text{ new } c() \cdot \text{ main }() \) and store \( s_0 = \emptyset \) contain no handles, and environment stack \( \eta_0 = \emptyset_{\text{obs}}^{\text{nil}, \text{read}, \text{nil}} \) contains only a nil-handle.

In the induction step, execution \( e_0, \eta_0, s_0, \omega_0, g_0 \Longrightarrow^* e_N, \eta_N, s_N, \omega_N, g_N \) is continued \( e_N, \eta_N, s_N, \omega_N, g_N \Longrightarrow e, \eta, s, \omega, g \). 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 occurrences 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 \( g = ogr(e, \eta, s) \) follows from the induction hypothesis's \( g_N = ogr(e_N, \eta_N, s_N) \).
5.3.2 Modeled 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 \( o \rightarrow \omega \) according to their ownership- and sanctuary-meaning (cf. §5.1) generalizes to paths from \( o \) to \( \omega \) (subsuming the case of object references as paths of length one):

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

Paths \( \pi \) of mode \( \mu \) between \( o \) and \( \omega \) can be written \( o \xrightarrow{\mu} \omega \) 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 \( \omega \). 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 \( \omega \) along the path which returns \( \omega \)'s \texttt{this-handle} \( \langle \omega, \text{co}, \omega \rangle \) to \( o \). If a path is classified as a \( \mu \)-path then this collapsed reference must be of mode \( \mu \). That is, the combined adaption \( \mu_1 \circ (\mu_2 \circ \ldots (\mu_{n-1} \circ (\mu_n \circ \text{co})) \ldots) \) of the returned handle’s mode should be \( \mu \). That is, while a \( \mu \)-reference is a \textit{means} for the source \( o \) to directly access target \( \omega \) (with some limitations imposed by the type system according to \( \mu \)), a \( \mu \)-\texttt{path} indicates \( o \)'s (\( \mu \)-bounded) \textit{right}, in principle, to directly access \( \omega \). Hence these paths will be called potential access \textit{paths}.

Let \( \text{PAP}(o, \mu, \omega) \) be the set of all those paths \( \pi \) from \( o \) to \( \omega \) in graph \( g \) which are potential access paths of mode \( \mu \). Which paths in \( g \) are potential access paths, and what their mode is, is controlled by the modes of the edges in \( g \). 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 \( o \xleftarrow{\mu} \omega \in g \) (that is, edges with \( \text{mult}(o \xleftarrow{\mu} \omega, g) > 0 \)), and the concatenation \( \pi_1 \cdot \pi_2 \) of a \( \mu \)-path \( \pi_1 \in PAP(o, \mu, q) \) and a \( \mu \)-path \( \pi_2 \in PAP(q, \mu, \omega) \) (see figure 5.9). This is easy to verify: A \( \text{rep} \)-path \( \pi_1 \) and co-path \( \pi_2 \) together imply that \( o \) is not only \( q \)'s but also \( \omega \)'s owner and that not only \( q \) but also \( \omega \) is in \( o \)'s sanctuary. This is exactly what the classification of \( \pi_1 \cdot \pi_2 \) as \( \text{rep} \) says about \( \omega \). If \( \pi_1 \) is \( \text{free} \) then the co-path \( \pi_2 \) between \( q \) and \( \omega \) implies that \( o \) is not only \( q \)'s but also \( \omega \)'s owner and that not only \( q \) but also \( w \) is in no object's sanctuary. Hence \( \pi_1 \cdot \pi_2 \) should be classified as \( \text{free} \). Co paths \( \pi_1 \) and \( \pi_2 \) together mean that \( o \) and \( \omega \) have the same owners and belong to the same sanctuaries, i.e., \( \pi_1 \cdot \pi_2 \) is \( \text{co} \).

Read path \( \pi_1 \) leaves \( q \)'s owners and sanctuary memberships unspecified, and co-path \( \pi_2 \) equates them with \( w \)'s owners and sanctuary-memberships. They are thus left unspecified by \( \pi_1 \cdot \pi_2 \), so that read is the right mode.

### 4. Ownership and Sanctuaries

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

However, potential access paths are only forward concatenations of handles, so that in the situation \( o \xleftarrow{\text{rep}} \omega \xleftarrow{\text{co}} \omega' \), object \( o \) would own \( \omega \) but not \( \omega' \). But when \( \omega' \) calls an operation on \( \omega \), like \( \text{SetPrev} \), to which it passes this as parameter of mode \( \text{co} \), then the co-handle is inverted and \( o \) now also owns \( \omega' \) through path \( o \xleftarrow{\text{rep}} \omega \xleftarrow{\text{co}} \omega' \). 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 \( \text{co} \)-symmetric one, in which both ends of \( \text{co} \)-handles and \( \text{co} \)-paths have the same owner and belong to the same representations.

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

**Definition 3** Let \( g^* = \text{df} g \uplus \{ \omega \xleftarrow{\text{co}} o \mid o \xleftarrow{\text{co}} \omega \in g \} \). Then

\[
PAP_g(o, \mu, q) = \text{df} \{ \pi \mid g^* \vdash \pi \in PAP(o, \mu, q) \}
\]

\[
\text{Osh}_g(o, \omega) = \text{df} PAP_g(o, \text{rep}, \omega) \cup PAP_g(o, \text{free}, \omega)
\]

\[
\text{Sanc}_g(o) = \text{df} \bigcup \omega \text{ su. th. } PAP_g(o, \text{rep}, \omega) \neq \emptyset \bigcup \{ \omega \} \cup \text{Sanc}_g(\omega)
\]

Object graph index \( g \) in \( PAP_g \), \( \text{Osh}_g \) and \( \text{Sanc}_g \) can be dropped where \( g \) is obvious.

The construction of \( PAP_g \) based on \( g^* \) means that any two objects \( o \) and \( \omega \) that are in \( g \) connected by an undirected path \( o \xleftarrow{\text{co}}^* a_1 \xleftarrow{\text{co}}^* \ldots \xleftarrow{\text{co}}^* a_n \xleftarrow{\text{co}} \omega \) of co-edges will be connected by a potential access paths of mode \( \text{co} \) \( (PAP_g(o, \text{co}, \omega) \neq \emptyset) \). Since
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 g and the call-links and method kinds of invocations on the environment stack i.

- The Unique Owner property UO is the property characteristic of object ownership in JaM: It holds in graph 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 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 g and stack i if mutators were invoked on receiver objects ri only through a sequence of calls along the edges hj, ..., hi of an an ownership path to ri.
- The Mutator Control property MC is the property characteristic of sanctuaries in JaM: It holds in graph g and stack i 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.

\[ g, \eta \models UO, UH, MCP, MC \text{ is short for } "g \models UO \text{ and } g \models UH \text{ and } g, \eta \models MCP \text{ and } g, \eta \models MC." \]

The move from g to g* strengthens the notion of uniqueness of owners and free path’s initial edges. For mutator control paths, this is irrelevant because the call-links
in $\vec{f}$ are always real edges in $g$ (Proposition 2). But by inversion, all the additionally owned objects and representation members, i.e., objects not reachable by any real ownership path in $g$, 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 $\text{composite}(o) = \text{rep}(o) = \cup \{o\} \cup \cup \text{composite}(o, \omega) \in \Omega \text{composite}(\omega)$. Interaction between the constituent 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 $\text{Sanc}(o)$ (see above) is the subset of the expansion $\text{composite}(o)$ 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 $\text{Sanc}(o)$), new rep paths can be established that add an object to $\text{Sanc}(o)$. 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 $\text{StRep}(o)$ can only consist, besides representative $o$ itself, of objects that are held in sanctuary $\text{Sanc}(o)$ 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 $fgr_{om}(s) \subseteq ogr(e, \tilde{r}, s)$ containing just the edges for handles $h = s(\ell)$ found in the store $s$ and at locations $\ell \in fds_{om}(o)$ that model an object’s fields.

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

$$fds_{om}(o) =_{df} \text{im}(\rho) \quad \text{for } om(o) = (\rho, F)$$
$$fgr_{om}(s) =_{df} \bigcup_{o \in \text{dom}(om)} \text{im}(s|_{fds_{om}(o)}) \cap \emptyset \times M \times \emptyset$$
$$StReps_{s,om}(o) =_{df} \{o\} \cup \bigcup_{\omega \text{ su. th. PAP}_{fgr_{om}(s)}(o, \text{rep}, \omega) \neq \emptyset} StReps_{s,om}(\omega)$$

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

$$fgr_{om}(s) \subseteq ogr(e, \tilde{r}, s)$$

Proof: $h \in fgr_{om}(s) \Rightarrow h \in \text{im}(s) \Rightarrow \text{num}(h, s) > 0 \Rightarrow h \in ogr(e, \tilde{r}, s)$. ■

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 $StReps_{s,om}(o) \subseteq \{o\} \cup Sanc_{gr_{om}(e, \tilde{r}, s)}(o)$

Proof: Membership $\omega \in StReps_{s,om}(o)$ presupposes a (possibly empty) sequence of rep paths from $o$ to $\omega$ in $fgr_{om}(s)$. This sequence exists also in $ogr(e, \tilde{r}, s) \supseteq fgr_{om}(s)$ (Proposition 3). If it is empty then $\omega = o$, otherwise $\omega \in Sanc_{gr_{om}(e, \tilde{r}, s)}(o)$. ■

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 $\ell \in fds(o)$, the composite state is ultimately represented in the store at all the locations $\ell \in fds(\omega)$ 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 $s$, to the corresponding location sets:

$$state_{s,om}(o) =_{df} s|_{fds_{om}(o)}$$
$$CState_{s,om}(o) =_{df} \bigcup_{\omega \in StReps_{s,om}(o)} state_{s,om}(\omega)$$
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 \( \text{composite}(o) \), as representatives of a (possibly primitive) composite object \( O' \). The components of composite \( O \) are those composite objects \( \Omega_i \) to whose representatives \( \omega_i \) 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 \texttt{rep} path in the field-subgraph. Correspondingly, one could also give inductive definitions of \( \text{composite}(o) \), \( \text{StRep}(o) \), and \( \text{CState}(o) \) 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 \( e, \eta(s, o, g) \rightarrow e', \eta(s', o, g') \) changes a composite’s state, i.e., \( \text{CState}_{s, o, g}(o) \neq \text{CState}_{s', o, g'}(o) \), then it is executing a mutator, i.e., there is an environment \( \eta_{(s, \mu, o)}^{\text{mut}} \in \eta \) of kind \texttt{mut} with receiver \( o \): 

\[ \forall o \in \text{dom}(om). \ CState_{s, o, g}(o) \neq \text{CState}_{s', o, g'}(o) \Rightarrow \exists s, \mu, \eta_{(s, \mu, o)}^{\text{mut}} \in \eta \]

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 \textit{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 \textit{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 \textit{legal program} (of typed base-JaM) whose execution starts with the evaluation of \( e_0 \equiv \text{new} \ c_n(). \text{main}() \), written \( \vdash p \text{ start } e_0 \), if it is \textit{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 \text{main}. For formal reason, this operation must be an \textit{observer}: In the initial environment \( O_{(\text{nil, read, nil})}^{\text{obs}} \), the only handle to the initially created \( c_n \)-instance \( o \) will be \( \langle \text{nil, free, o} \rangle \). Since its source is \( \text{nil} \not\in O \), in
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.\(^6\)

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

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

\[\text{[field] A field definition } M \equiv t x \text{ is a well-formed definition of member } x, \text{ written } \vdash M \text{ def } s x, \text{ if its declared range type } t \text{ is a valid type.}\]

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

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

---

\(^6\)Mutator main could be supported by reformulating the Mutator Control Path property or by assuming a given object \( o_0 \in \mathcal{O} \) as the receiver in the initial environment: \( \tilde{\eta}_0 = \mathcal{O}_{\text{init, read, } o_0} \)
Identifier $c \in C$ is a valid class name, written $\vdash c \text{ ok}$, 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 $\kappa$ and has static type $\tau$ in the context of type assumptions $\Gamma$ for local variables. (The assumptions are met if they are type consistent with top-level environment $\eta^\kappa$, i.e., $\eta \models \eta^\kappa \pi \Gamma$.) 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 $\tau' \leq_m \tau$, the signature $\Sigma(\mu c)$ of $\mu c$-handles, and the set $\mathsf{Wr}(\kappa)$ of handle modes with write permission, will be deferred to §5.4.2.

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

Field expressions $\text{this}.x$ evaluate to the location of field $x$ of the object referenced by $\text{this}$. Hence they are assigned the type $\tau$ which instance record type $\mathsf{co} c$ specifies for $x$, where $c$ is the target class in the range type $\mathsf{co} c$ of $\text{this}$'s assumed type $\Gamma(\text{this})$.

Expressions of read access to a variable named $\nu$ are normally assigned the type $\tau$ of the value range in the variables' type $\mathsf{ref} \tau$ determined for $\nu$. In case of mode-preserving non-destructive read, this is always legal. Destructive read expressions $\text{destval}(\nu)$ are legal only if the variable $\nu$ is not $\text{this}$, and legal if $\nu$ 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 $\{\text{rdcp}\}$ in paragraph 7). Correspondingly, the type inferred for non-destructive read expressions is the substitution $\tau[\text{read}/\text{free}]$ of read for free in the read variable's range type.

Since $\text{null}$ evaluates to a free nil-handle, it can be assigned free handle types with any valid class name $c$.

Since object creation expression $\text{new } c()$ evaluates to free handles targeting new $c$-objects, it gets type free $c$. It is legal if $c$ is a valid class name.\footnote{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.}

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 $\kappa$ permits mutator invocations (see §5.4.2).
Figure 5.13: Typing rules for program terms

[upd] Assignment statements reduce to ε and are therefore given the special type \(\text{Cmd}\). 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 \(\tau\) of the left-hand side. The left-hand side must not be \(\text{this}\), and a field expression only inside a method of mutator kind.

[ret] The (return) type of a \texttt{return} statement is the type of its return expression. The typing rules for sequences and \texttt{if} and \texttt{while} will imply that \texttt{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 \(\tau\). To be legal, the first statement must be of the type \(\text{Cmd}\) of continuing statements, i.e., a statement not returning from the current method.

[if] If statements reduce to \(\varepsilon\) or to their then-branch. They are given the type \(\text{Cmd}\) 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.,
Figure 5.14: Typing rules for runtime terms and consistency with extended context

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:

\[ \text{[val]} \quad \frac{v \in [\tau]}{\Gamma, \kappa \vdash \tau} \quad \text{[nest]} \quad \frac{\Gamma, \kappa \vdash \mu \in c}{\Gamma', \kappa' \vdash \mu', \nu, \kappa, \tau \vdash s : \mu' \circ \mu \in c} \]

\[ \eta \models \kappa_i \mid \mu_i, \Gamma_i, \kappa_i \iff \forall \eta_i \models \Gamma_i \land \kappa_i = \kappa_i \land \exists \alpha_i, \omega_i, \mu_i = (\alpha_i, \mu_i, \omega_i) \]

[wh] While loops reduce to if statements and therefore have type \( \text{Cmd} \). 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).

Note however that irreducible "term" \( \epsilon \) is not an element of \( R \).
The original rules with '1-' can then be seen as the special case '1-\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 $\bar{\eta} = \bar{\eta}_{h_i}$ is type consistent with a sequence $X = \mu_i, \Gamma_i, \kappa_i$ of type assignments, method kinds, and call-link modes, written $\bar{\eta} \models X$, under the following conditions: Each environment is type consistent with its corresponding type assignment. The sequences of method kinds in $\bar{\eta}$ and $X$ are the same. And the modes of the call-links in $\bar{\eta}$ 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 $\mu[\text{read}/\text{free}]$ in non-destructive read and mode import $\mu_r\sigma\mu$ 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\_det] (fig. 5.13) permit assignment to fields and destructive read of fields only within methods declared mutator ($\kappa = \text{mut}$). The invocation of methods declared mutator is limited in rule [call] through $\text{Wr}$ defined in figure 5.15 to enforce shallow and composite state encapsulation:

- Calling mutators through free handles is always permitted ($\text{free} \in \text{Wr}(\kappa)$) 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 ($\text{rep} \in \text{Wr(mut)}$) 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 (κ = 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 (co ∈ Wr(mut)). How­
ever, 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 τ' |= μ' | c' is mode-compatible to τ |= μ c, written
τ' |<="m τ, if c' = c and μ' is mode-compatible to μ, written μ' |<="m μ, 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 ini­
tial 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 com­
patible in assignments and calls would not generally be safe:

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

\[ \text{rep, co or read } \leq_m \text{ free would allow an object } o \text{ to convert a non-free handle } h \text{ to free } h_f \]
and then convert a copy of \( h \) also to \( h_f \), thus violating Unique Head. \( \text{read } \leq_m \text{ rep or co would allow } o \text{ to convert a read handle } h \text{ to rep or co, thus making, respectively, itself or its own owner } q \text{ to the}
owner of } h \text{ 's target } w. \text{ However, } w \text{ may already have an owner, and this owner is not guaranteed to}
be } o \text{ or } q, \text{ respectively, so that Unique Owner could be violated. } \text{rep } \leq_m \text{ co and co } \leq_m \text{ rep would allow } o \text{ to convert a co handle } h \text{ to a rep handle } h' \text{ or vice versa. } h' \text{ and old copy of } h \text{ make } o \text{ an owner as well as a co-object of } h \text{ 's target } w. \text{ Owning } w, o \text{ owns } w \text{ 's co-object } o. \text{ However, } o \text{ may}
already have an owner that is not } o, \text{ so that Unique Owner could be violated.}
can be supported by allowing to create \texttt{read} copies of \texttt{free} handles in \texttt{free} variables through non-destructive read access. Calling mutators like \texttt{Step()} is possible only if the \texttt{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 \texttt{free} handle to the receiver and can use it for further calls. Hence there is a solution for both observer and mutators calls to \texttt{free} objects.

Observe that, while \texttt{free} is compatible to \texttt{read}, mode compatibility alone is not a sufficient reason: A copy weakened only to \texttt{rep} or \texttt{co} would, respectively, still violate Unique Head or risk violating Unique Owner if the converting object has an owner. While \texttt{free} and \texttt{read} handles between the same objects can coexist (true inclusion polymorphism, “submoding”), a conversion of a \texttt{free} handle to \texttt{rep} and \texttt{co} is only safe because no \texttt{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 \(\mu\) 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 \(\mu_r \circ \mu\) that is calculated from \(\mu\) relative to the call-link’s mode \(\mu_r\) and is mode-compatible to all other adaptions that might be desirable. This adaption, called the \texttt{import} of \(\mu\) through \(\mu_r\) and written \(\mu_r \circ \mu\), is defined in figure 5.15:

- A returned \texttt{read} handle can only remain \texttt{read} since it provides no information that would make another mode a safe choice.
- The sender can safely import a \texttt{free} handle from the receiver as \texttt{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 \texttt{free} handle from the graph.
- If the receiver returns a \texttt{rep} handle, however, the receiver may still possess further \texttt{rep} handles with the same target, and thus remain the target’s owner. Hence the sender cannot import the handle as \texttt{free} or \texttt{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 $\mu_r$ of the call-link: If $\mu_r$ 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 $\mu_r$ 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 $\mu_r$ 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 $\mu_r$ 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 $\tau \Rightarrow \tau$ of $f$ in the signature $\Sigma(\mu_r, c)$ of handles of the receiver expression’s type $\mu_r$ $c$. The operations which can be called through a handle of type $\mu_r$, $c$ 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 $\mu_r$, $c$ handles must have the mode $\mu_r \sigma \mu$ to which the mode of returned $\mu$-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 $\mu = \text{read}$ 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 $\mu_r \sigma \text{read} = \text{read}$.
- If the $c$-object expects $\mu = \text{free}$ parameter values then only $\mu_r \sigma \text{free} = \text{free}$ 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 $\mu = \text{rep}$ 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 $\mu = \text{co}$ 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 \( \mu_r = \text{read} \) 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 \( \mu_r = \text{co} \), a \( \mu_r \circ \text{co} = \text{co} \) handle of the sender is just right, since the \( \mu_r = \text{co} \) means that sender and receiver have the same owner and are in the same sanctuaries, and \( \mu_r \circ \text{co} = \text{co} \) means that sender and target have the same owner and are in the same sanctuaries. And if the call-link is of mode \( \mu_r = \text{rep} \) or \( \text{free} \) then only a, respectively, \( \text{rep} \) or \( \text{free} \) 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, \( o \), 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 \((= s, om)\), 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 \( \text{FldsMths}(c) \) by adapting the modes \( \mu_i \) in it to \( \mu_i \circ \mu_i \).

### 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 \( e, \bar{\eta}, s, om, g \mapsto e', \bar{\eta}', s', om', g' \) is a reduction step defined relative to a program \( p \) that is legal, i.e., \( \vdash p \) start \( e_0 \), then

\[
\begin{align*}
\Gamma, \kappa \vdash_x e : \tau & \quad \land \quad \bar{\eta} \models \bar{\mu}, \Gamma, \kappa, X \quad \land \quad \models s, om
\Rightarrow \quad \exists X'. \quad \Gamma, \kappa \vdash_x e' : \tau \quad \land \quad \bar{\eta}' \models \bar{\mu}, \Gamma, \kappa, X' \quad \land \quad \models s', om'
\end{align*}
\]

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

Lemma 2 \( \Gamma, \kappa \vdash_x (s, \bar{\mu}, r) : \bar{\mu}' c \land om(r) = \langle g_r, F \rangle \land \models om \Rightarrow \bar{\mu} = \bar{\mu}' \land r \in \mathcal{O}_c \land \exists \Gamma_c \cdot FlsMths(c) = \langle \Gamma_c, F \rangle \land g_r \models \Gamma_c \)

Proof: First, \( \Gamma, \kappa \vdash_x (s, \bar{\mu}, r) : \bar{\mu}' c \Rightarrow (s, \bar{\mu}, r) \in \langle \bar{\mu}' c \rangle \Rightarrow \bar{\mu} = \bar{\mu}' \land r \in \mathcal{O}_c \Rightarrow \exists \Gamma_c \cdot FlsMths(c) = \langle \Gamma_c, F \rangle \land g_r \models \Gamma_c \)

Proof of the main lemma: \( e, \bar{\eta}, s, om, g \mapsto e', \bar{\eta}', s', om', g' \) means there is a multi-level context \( \mathcal{E}' \), a redex \( \bar{\epsilon} \) and a term \( \epsilon' \) such that \( e = \mathcal{E}'[\bar{\epsilon}] \) and \( \epsilon' = \mathcal{E}'[\epsilon'] \), and postfixes \( \bar{\eta} \) and \( \bar{\eta}' \) of \( \bar{\eta} \) and \( \bar{\eta}' \) such that \( \bar{\epsilon}, \bar{\eta}, s, om, g \mapsto \epsilon', \bar{\eta}', s', om', g' \). 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 \( \mathcal{E}' \). In the base case, \( \mathcal{E}' \) contains no inlined method, i.e., \( \mathcal{E}' = \mathcal{E} \in R^D \), \( \bar{\eta} = \bar{\eta}' \) and \( \bar{\eta}' = \bar{\eta}' \), and \( X = \epsilon \). In the simplest case, \( \mathcal{E} = \Box \) and \( \bar{\epsilon} = e \). 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: \( \bar{\eta} = \bar{\eta}' = \eta \). Then \( \eta \models \bar{\mu}, \Gamma, \kappa, X \models \eta \models \Gamma \). First, we derive that the new term \( \epsilon' \) that can be typed as \( \tau \) or, if it is an irreducible value, that it belongs to \( \tau \)'s extension \( \lbrack \tau \rbrack \), from which \( \Gamma, \kappa \vdash_x \epsilon' : \tau \) follows immediately for annotation \( X' = \epsilon = X \).

\{
\text{var}_I\}: \quad e = x \overset{\epsilon'}{\rightarrow} \Gamma(x) = \tau \quad \overset{\eta\models\epsilon}{\Rightarrow} \eta(x) = \epsilon' \in \lbrack \tau \rbrack
\{
\text{var}_r\}: \quad e = \text{this}.x \overset{\epsilon'}{\rightarrow} \left\{ \begin{array}{l}
\Gamma(\text{this}) = \text{ref co} c \overset{\eta\models\epsilon}{\Rightarrow} \eta(\text{this}) = \ell \in \text{Loc}_{\text{co} c} \\
\Rightarrow \overset{\text{defd}}{s(\ell) \in \lbrack \text{co} c \rbrack} \overset{\text{defd}}{o \in \mathcal{O}_c} \overset{\text{defd}}{\text{om}(o) \in \lbrack \text{obj} c \rbrack} \Rightarrow \overset{\text{defd}}{g' \in \Gamma_c \land g'(x) \in \lbrack \tau \rbrack}
\end{array} \right.
\}
\{
\text{rd}_\text{st}\}: \quad e = \text{destval}(\ell) \overset{\epsilon'}{\rightarrow} \Gamma, \kappa \vdash_x \ell : \text{ref} \Rightarrow \ell \in \text{Loc}_r \overset{\eta\models\epsilon}{\Rightarrow} s(\ell) = \epsilon' \in \lbrack \tau \rbrack
\{
\text{rd}_\text{cp}\}: \quad e = \text{val}(\ell) \overset{\epsilon'}{\rightarrow} \exists \tau, \Gamma, \kappa \vdash_x \ell : \text{ref} \land \tau = \ell[\text{read}, \text{free}] \Rightarrow \ell \in \text{Loc}_r \\
\Rightarrow \overset{\eta\models\epsilon}{\Rightarrow} s(\ell) \in \lbrack \tau \rbrack \Rightarrow s(\ell)[\text{read}, \text{free}] = \epsilon' \in \lbrack \tau \rbrack
\}
\{
\text{null}\}: \quad e = \text{null} \overset{\epsilon'}{\rightarrow} \tau = \text{free} c \Rightarrow \epsilon' = \langle r, \text{free}, \text{nil} \rangle \in \lbrack \tau \rbrack
\}
\{
\text{new}\}: \quad e = \text{new} c() \Rightarrow o \in \mathcal{O}_c \land \tau = \text{free} c \Rightarrow \epsilon' = \langle r, \text{free}, o \rangle \in \lbrack \tau \rbrack
\}

103
\{upd\}: \quad e = \ell = (r, \mu, \omega); \; \xrightarrow{\text{ct}} \; \tau = \text{Cmd} \Rightarrow e' = e \in \tau

\{if\}: \quad e = \text{if}(h_1 \psi h_2)\{s\}; \; \xrightarrow{\text{ct}} \; \tau = \text{Cmd} \Rightarrow e' = e \in \tau

\{if\}': \quad e = \text{if}(h_1 \psi h_2)\{s\}; \; \xrightarrow{\text{ct}} \; \tau = \text{Cmd} \land \Gamma, \kappa \vdash_X s : \text{Cmd} \; \xrightarrow{\text{ct}} \; \Gamma, \kappa \vdash_X e' : \tau

\{wh\}: \quad e = \text{while}(h_1 \psi h_2)\{s\}; \; \xrightarrow{\text{ct}} \; \tau = \text{Cmd} \land \Gamma, \kappa \vdash_X s : \text{Cmd} \land \Gamma, \kappa \vdash h_1 : \mu_1 c_1

\land \Gamma, \kappa \vdash_X h_2 : \mu_2 c_2 \Rightarrow \Gamma, \kappa \vdash_X \text{if}(h_1 \psi h_2)\{s\} e \in \text{Cmd} \; \xrightarrow{\text{ct}} \; \Gamma, \kappa \vdash_X e' : \tau

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

In case of \{new\}, nil-handles \(\langle o, \mu, \text{nil} \rangle\) are filled into the store at locations in \(\text{ref} \mu_1 c_1\) = Loc\(\mu_1 c_1\) reserved for handles of these modes. Hence \(\models s'\). And the new \(o \in \text{Obj}_c\) 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 \{rd\(\text{std}\)\} and \{upd\}, \(om' = om\), so that \(\models om'\). By \(\models s, s(\ell) = \langle o, \mu, \omega \rangle\) means that for some class \(c, \ell \in \text{Loc}_{\mu_1 c}\) and \(\langle o, \mu, \omega \rangle \in \llbracket \mu c \rrbracket\). But then the new store value \(s'(\ell) = \langle o, \mu, \text{nil} \rangle\) of the \{rd\(\text{std}\)\}-case is in \(\llbracket \mu c \rrbracket\), so that \(\models s'\). In the \{upd\}-case, we have to consider what class the target \(\omega' \neq \text{nil}\) of the new store value \(s'(\ell)\) is. \(\ell \in \text{Loc}_{\mu_1 c}\) (see above) means that \(\Gamma, \kappa \vdash_X \ell : \text{ref} \mu c\). And having a typing for \(e \equiv \ell = (r, \mu', \omega')\) means \(\Gamma, \kappa \vdash (r, \mu', \omega') : \mu' \circ c\) with \(\mu' \cdot c' \leq c_1\). Hence \langle r, \mu', \omega' \rangle \in \llbracket \mu' \circ c \rrbracket\) with \(\mu' \leq c_1\) and \(c' = c\), and thus \(\omega' \in \text{Obj}_c \cup \{\text{nil}\}\). But then also \(s'(\ell) = \langle o, \mu, \omega' \rangle \in \llbracket \mu c \rrbracket = \text{Loc}_{\mu_1 c}\), so that \(\models s'\).

\{ret\} Return is the case where the environment shrinks from \(\eta' = \eta^*_{h'} \cdot \eta^*_{(s, \mu, r)} \) to \(\eta^*_{h'}\).

\(\models \ldots \Rightarrow X = \mu^* \cdot \Gamma^*, \kappa^*, \epsilon\)

\(\xrightarrow{\text{ct}} \quad \Gamma^*, \kappa^* \vdash X = \mu^* \cdot \Gamma^*, \kappa^*, \epsilon\)

\(\xrightarrow{\text{ct}} \quad \Gamma, \kappa \vdash_X \text{return } (r, \mu, \omega) ; \; \mu' \cdot c \land \tau = \mu^* \circ \mu' \cdot c\)

\(\xrightarrow{\text{ct}} \quad \Gamma, \kappa \vdash_X (r, \mu, \omega) ; \; \mu' \cdot c\)

\(\xrightarrow{\text{ct}} \quad \Gamma, \kappa \vdash_X \text{return } (r, \mu, \omega) ; \; \mu' \cdot c\)

\(\xrightarrow{\text{ct}} \quad \Gamma, \kappa \vdash_X \text{return } (r, \mu, \omega) ; \; \mu' \cdot c\)

\(\xrightarrow{\text{ct}} \quad \Gamma, \kappa \vdash_X \text{return } (r, \mu, \omega) ; \; \mu' \cdot c\)

\(\models \ldots \Rightarrow X = \mu^* \cdot \Gamma^*, \kappa^*, \epsilon\)

\(\models \ldots \Rightarrow X = \mu^* \cdot \Gamma^*, \kappa^*, \epsilon\)

\(\models \ldots \Rightarrow X = \mu^* \cdot \Gamma^*, \kappa^*, \epsilon\)

\(\models \ldots \Rightarrow X = \mu^* \cdot \Gamma^*, \kappa^*, \epsilon\)

\(\models \ldots \Rightarrow X = \mu^* \cdot \Gamma^*, \kappa^*, \epsilon\)

\(\models \ldots \Rightarrow X = \mu^* \cdot \Gamma^*, \kappa^*, \epsilon\)

\(\models \ldots \Rightarrow X = \mu^* \cdot \Gamma^*, \kappa^*, \epsilon\)

\(\models \ldots \Rightarrow X = \mu^* \cdot \Gamma^*, \kappa^*, \epsilon\)

Since the new term \(e'\) is \(\llbracket s \rrbracket\), this gives us the desired \(\Gamma, \kappa \vdash_X e' : \tau\) with \(X' = \mu, \Gamma^*, \kappa^*, \epsilon\). The new stack \(\eta'\) is \(\eta^*_{h'} \cdot \eta^*_{(s, \mu, r)}\), where the new top-level environment \(\eta^*\) maps identifiers \(x_i\) to locations in \(\text{ref} \tau_i\). The types \(\tau_i\) are determined from the
declarations in method $F(f)$ or $\tau_i$ is, in case of $x_i = \text{this}$, the type $\text{co c}$ since $r \in \mathbb{O}_c$.

The type assignment $\Gamma^\ast$ maps the same identifiers to corresponding type terms $\text{ref } \tau_i$. Hence $\eta^\ast \models \Gamma^\ast$. Combined with $\overline{\eta} = \eta_k^\ast \models \tilde{\mu}, \Gamma, \kappa, X$, therefore $\overline{\eta} = \eta_k^\ast \cdot \eta^\ast_{(s, \tilde{\mu}, r)} \models \tilde{\mu}, \Gamma, \kappa, \tilde{\mu}, \Gamma^\ast, \kappa^\ast, \epsilon = \tilde{\mu}, \Gamma, \kappa, X'$. Since $om$ is unchanged, $\models om'$ by assumption. Consider the extensions of the store: this’s location $\ell \in [\text{ref co c}]$ is mapped to a corresponding handle $(r, \text{co, r}) \in [\text{co c}]$. The local variables’ locations $\ell_j \in [\text{ref } \mu'_j \text{ c}_j]$ are mapped to corresponding nil-handles $(r, \mu'_j, \text{nil}) \in [\text{ref } \mu'_j \text{ c}_j]$. The parameters’ locations $\ell_i \in [\text{ref } \mu_i \text{ c}_i]$ are mapped to handles $(r, \mu_i, \omega_i)$. The target classes match because of the typing of $e = (s, \tilde{\mu}, r) \leftarrow f((s, \mu_i, \omega_i))$: Derived from the types $\mu_i$ declared in $F(f)$, the handle signature is $(f : \mu \circ \mu_i \circ \text{nil}) \in \Sigma(\tilde{\mu} \ c)$. Hence the argument expressions $(s, \mu_i, \omega_i)$ must be typed with a subtype of $\mu \circ \mu_i$, i.e., some $\mu'_i \ c_i$. This means $(s, \mu_i, \omega_i) \in [\mu'_i \ c_i]$, and thus $\omega_i \in c_i$. Hence $(r, \mu_i, \omega_i) \in [\mu_i \ c_i]$. This shows that $\models s'$.

If the case that the single-level context $\mathcal{E}^\ast = \mathcal{E}$ is not empty ($\mathcal{E} \neq \square$), consider that typing $e = \mathcal{E}[\tilde{e}]$ required to have a typing for all of its subterms, in particular redex $\tilde{e}$. Since $\mathcal{E}$ contains no inlined method, the typing of $\tilde{e}$ must have been in the same context. That is, $\Gamma, \kappa \vDash_X \tilde{e} : \tilde{\tau}$ for some $\tilde{\tau}$. In conjunction with $\tilde{e}, \tilde{\eta}, s, om, g \rightarrow \tilde{e}', \tilde{\eta}', s', om', g'$, the case of $\mathcal{E} = \square$ above allows one to conclude that there is an $X'$ such that $\Gamma, \kappa \vDash_X \tilde{e}' : \tau$ and $\tilde{\eta}' \models \tilde{\mu}, \Gamma, \kappa, X'$ and $\models s', om'$. Since $\tilde{e}$ and $\tilde{e}'$ have the same type in the same context, if a type is inferred for $\mathcal{E}[\tilde{e}']$ it must be the type $\tau$ of $\mathcal{E}[\tilde{e}]$. 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 $\tilde{e}'$ of a redex substitution can neither be, nor be contained in, ‘this’ nor ‘this.x’. Therefore $\Gamma, \kappa \vDash_X \tilde{e}' : \tau$.

In the induction step, $\mathcal{E}[\llbracket e'' \rrbracket]$, $\eta_k^\ast \cdot \tilde{\eta}, s, om, g \rightarrow \mathcal{E}[\llbracket e''' \rrbracket]$, $\eta_k^\ast \cdot \tilde{\eta}, s', om', g'$. The typing of $e = \mathcal{E}[\llbracket e'' \rrbracket]$ required to have a typing for all of its subterms, in particular $\llbracket e'' \rrbracket$. Since $\mathcal{E}$ contains no inlined method, the typing of this subterm must have been in the same context, i.e., $\Gamma, \kappa \vDash_X \llbracket e'' \rrbracket : \tilde{\tau}$ for some $\tilde{\tau}$. This typing requires that $\Gamma^\ast, \kappa^\ast \vDash_X e'' : \mu c$ with $\tilde{\tau} = \mu \circ \mu c$ and $X = \mu^\ast, \Gamma^\ast, \kappa^\ast, X^\ast$. And $\eta_k^\ast \cdot \tilde{\eta} \models \tilde{\mu}, \Gamma, \kappa, X$ means $\tilde{\eta} \models X$. With the induction hypothesis it follows that $\Gamma^\ast, \kappa^\ast \vDash_X e'' : \mu c$ and $\tilde{\eta} \models \mu^\ast, \Gamma^\ast, \kappa^\ast, X^\ast$ and $\models s', om'$. Since $e''$ and $e'''$ have the same type in the same context, $\Gamma, \kappa \vDash_X \llbracket e'' \rrbracket : \tau$ with $X = \mu^\ast, \Gamma^\ast, \kappa^\ast, X^\ast$ implies the desired $\Gamma, \kappa \vDash_X \llbracket e''' \rrbracket : \tau$ with $X = \mu^\ast, \Gamma^\ast, \kappa^\ast, X^\ast$. Finally, $\eta_k^\ast \cdot \tilde{\eta} \models \tilde{\mu}, \Gamma, \kappa, \mu^\ast, \Gamma^\ast, \kappa^\ast, X^\ast$ since $\overline{\eta} \models \mu^\ast, \Gamma^\ast, \kappa^\ast, X^\ast$ and $\tilde{\eta} \models \mu^\ast, \Gamma^\ast, \kappa^\ast, X^\ast$. \hfill \blacksquare

10. Type System Correctness. As corollary from Lemma 1 we get a standard property of typed programming languages: The type system, by assigning types $\tau$ to the program’s terms $e$ (static types), correctly predicts the types of the values $\tau$ to which these terms will evaluate (dynamic types) in environments $\tilde{\eta}$ consistent with the assumptions $\Gamma$ made in the typing rules.
Corollary 1 If \( \vdash p \text{ start } e_0 \) and \( \models s, \text{om} \) and \( \check{\eta} \models \mu, \Gamma, \kappa, X \) then
\[
\Gamma, \kappa \vdash_X e : \tau \land (e, \check{\eta}, s, \text{om}, g \Rightarrow^* v, \check{\eta}', s', \text{om}', g') \land v \in \text{Loc} \cup \mathcal{V} \cup \{e\} \Rightarrow v \in \underbracket{\tau}
\]

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 \( v \in \text{Loc} \cup \mathcal{V} \cup \{e\} \), this typing is only possible by \( v \in \underbracket{\tau} \).

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 \( v \in \text{Loc} \cup \mathcal{V} \cup \{e\} \) 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 \( e_0, \eta_0, s_0, \text{om}_0, g_0 \).

Theorem 1 If \( e_0, \eta_0, s_0, \text{om}_0, g_0 \Rightarrow^* e, \check{\eta}, s, \text{om}, g \) is a reduction defined relative to a program \( p \) with \( \vdash p \text{ start } e_0 \) then there is a \( \tau \) and an \( X \) such that
\[
\models s, \text{om} \land \emptyset, \text{obs} \vdash_X e : \tau \land \check{\eta} \models \text{read}, \emptyset, \text{obs}, X
\]

Proof by induction on the number \( N \) of reduction steps from \( e_0 \) to \( e \): In the base case \( N = 0 \), we have \( e, \check{\eta}, s, \text{om}, g = e_0, \eta_0, s_0, \text{om}_0, g_0 \). Empty store \( s_0 = \emptyset \) and object-map \( \text{om}_0 = \emptyset \) are trivially type-consistent. The type assignment matching the empty environment \( \eta_0 = \emptyset \models \emptyset \) is the empty set \( \emptyset \) of type assumptions. And the empty annotation \( X = \emptyset \) at the turnstile symbol matches the lack of further environments in the environment stack. Hence \( \eta_0 \models \text{read}, \emptyset, \text{obs}, X \).

Now consider the typing of the initial term \( e_0 \equiv \text{new } c() \cdot \text{main}() \): It is an operation call expression, which is typed by \( \text{call} \). Receiver expression \( \text{new } c() \) is typed by \( \text{new} \) as \( \text{free } c \) under condition \( \vdash c \text{ ok} \), which is satisfied since \( p \)'s legality, i.e., \( \vdash p \text{ start } e_0 \), guarantees \( \vdash D_n \text{ defs } c \) for \( p \equiv D_1 \ldots D_n \). Since \( \vdash p \text{ start } e_0 \) ensures that class \( c \) defines a method suite \( F \) containing some \( F(\text{main}) \equiv \text{mut } \tau' \text{ main}(\ldots) \) without parameters, \( \Sigma(\text{free } c) \) contains \( \text{main} : \epsilon \Rightarrow^* \tau \) with defined adaption \( \tau = \text{free} \circ \tau' \).

main’s kind \( \kappa \) is irrelevant since the receiver expression’s mode is \( \text{free} \). The lack of argument expressions in \( e_0 \) matches the lack of parameters in \( F(\text{main}) \). Hence \( e_0 \models \text{read}, \emptyset, X \).

In the induction step \( N \to N + 1 \), reduction \( e_0, \eta_0, s_0, \text{om}_0, g_0 \Rightarrow^* e', \check{\eta}', s', \text{om}', g' \) is continued \( e', \check{\eta}', s', \text{om}', g' \Rightarrow e, \check{\eta}, s, \text{om}, g \). From the induction hypothesis’s \( \models s', \text{om}' \) and \( \models \text{obs}', \text{obs} \vdash_X e', \tau' \) with \( \check{\eta}' \models \text{read}', e, \text{obs}, X' \), the theorem follows by type preservation (Lemma 1).
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.\(^\text{10}\)

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 \(e_0, \eta_0, s_0, om_0, g_0 \rightarrow^* e', \eta', s', om', g'\) is a reduction defined relative to a program \(p\) with \(\vdash p\) start \(e_0\) then

\[
g' \models \text{UH, UO}
\]

*Proof by induction on the number \(N\) of reduction steps from \(e_0\) to \(e'\):* In the base case \(N = 0\), \(g'\) is the empty graph \(g_0 = \emptyset\), which trivially satisfies UO and UH. In the induction step \(N \rightarrow N + 1\), reduction \(e_0, \eta_0, s_0, om_0, g_0 \rightarrow^* e, \eta, s, om, g\) is continued \(e, \eta, s, om, g \rightarrow e', \eta', s', om', g'\). By induction hypothesis, \(g \models \text{UO, UH}\). The question is whether the step to \(g'\) 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 $g^*$. A violation of UH or UO could at most be introduced in reduction steps that increase the multiplicity of such edges in $g$, i.e., if $g' = g \oplus o \perp \omega \ldots$ with $\mu \in \{\text{free, rep, co}\}$ and $o, \omega \in \mathcal{O}$. 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 $e^*$ surrounding the redex $e = e^*[\hat{e}]$. Hence we can move directly to a case analysis of the rule by which redex $e$ is reduced. In case of $\{\text{var}\}$, $\{\text{var}\}$, $\{\text{rddst}\}$, $\{\text{null}\}$, $\{\text{ift}\}$/$\{\text{if}\}$, 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 $\Gamma, \kappa \vdash e : \tau$ for the redex.

It follows from the typing $\Gamma, \kappa \vdash e : \tau$ guaranteed for the whole term (Theorem 1).

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

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

- In case of assignment with $e = e' = (o, [l, w])$ and $e' \in \text{Loc}_\mu c$, the multiplicity of $o \mu \omega$ is increased while that of $o \mu \omega$ is decreased (if $\omega \neq \text{nil}$). If $\mu = \hat{\mu}$, this means the only change from $g$ to $g'$ is the decrease of $\ell$'s old value's multiplicity. Handle $(o, \hat{\mu}, \hat{\omega})$ in term $e$ means that the multiplicity-decreased edge indeed existed in $g = \text{ogr}(e, \eta, s)$ (Proposition 2). Typing $\Gamma, \kappa \vdash e : \tau$ presupposes $\Gamma, \kappa \vdash \ell : \text{ref} \mu c$ and $\Gamma, \kappa \vdash (o, \hat{\mu}, \hat{\omega}) : \hat{\mu} c$ with $\hat{\mu} \geq_m \mu$. By the definition of $\leq_m$ (§5.4.2) then $\mu = \text{free}$ implies $\hat{\mu} = \text{free}$, $\mu = \text{rep}$ implies $\hat{\mu} = \text{rep}$ or free, and $\mu = \text{co}$ implies $\hat{\mu} = \text{co}$ or free. That is, in all cases of where an edge's multiplicity is increased because of $\mu = \hat{\mu}$, $\hat{\mu} = \text{free}$. But then the outer induction hypothesis $g \models \text{UH}$ guarantees that edge $o \mu \omega \in g$ is the head of all ownership paths to $\omega$ and its co-objects, and its multiplicity is 1. Consequently, in the intermediate graph $g'' = g \oplus o \mu \omega$ there is no ownership path to $\omega$ and its co-objects. Now consider the addition in $g' = g'' \oplus o \mu \omega$:
  - In case of $\mu = \text{rep}$, where there are no new co edges, all new ownership paths start with $o \rightarrow_{\text{rep}} \omega$ and go to $\omega$ and its co-objects. This cannot cause any UH- or UO-conflicts in $g'$ since $\omega$ and co-objects are unowned in $g''$.
  - In case of $\mu = \text{co}$, the addition of $o \rightarrow_{\text{co}} \omega$ entails the appearance also of its inverse $o \leftarrow_{\text{co}} \omega$ in $g''$. These two may give raise to new free or rep paths if they extend

108
old ones. \( o \xrightarrow{\text{old}} \omega \) cannot extend old ownership paths since there were no ownership paths to \( \omega \) in \( g'' \). Old ownership paths to \( o \) might be extended by \( o \xrightarrow{\text{old}} \omega \) and further co-paths to new ownership paths of the same mode that target \( \omega \), its old co-objects or, if further extended by inverse \( \omega \xrightarrow{\text{old}} o \), \( o \) and its co-objects.

\( g \models \text{UO} \) 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 \( \omega \), to \( o \), and to their co-objects have the source \( q \). There is no UO-conflict.

\( g \models \text{UH} \) 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 \( \omega \), \( o \), and to their co-objects, all ownership paths to them have head \( h \) of multiplicity one. There is no UH-conflict.

\begin{enumerate}
\item \textbf{In case of a return redex with} \( \hat{e} = \langle \text{return} \ (r, \mu, o); \rangle \) \textbf{and top-environment} \( \eta_h^n \) \textbf{with} \( h = (s, \mu_r, r) \), the multiplicity of \( s \xrightarrow{\mu \mu \nu \nu} o \) is increased, while those of \( r \xrightarrow{\mu \nu} o \) and \( s \xrightarrow{\mu \nu} r \) are decreased (if \( o \neq \text{nil} \)). Note that the decreased edges indeed exist in \( g = ogr(e, \eta, s) \) (Proposition 2) because of handles \( (r, \mu, o) \) in term \( e \) and \( (s, \mu_r, r) \) in the top-environment. Consider the mode \( \mu \): In case of \( \mu = \text{rep} \) or \( \text{read} \), the new edge has the harmless mode \( \text{read} \). In case of \( \mu = \text{free} \), the new edge has mode \( \text{free} \) and thus establishes free paths in \( g' \) from \( s \) to \( o \) and its co-objects. On the other hand, the receiver’s old free handle \( r \xrightarrow{\mu \nu} o \) was by \( g \models \text{UH} \) the initial edge in all ownership paths to \( o \) and its co-objects. All these are destroyed in \( g' \). Hence there can be no new UH- nor UO-conflict between new and unchanged potential access paths in \( g' \). Most complicated is the case of \( \mu = \text{co} \). Here, the new edge has mode \( \mu_r \text{co} = \mu_r \) and may, depending on this mode, be used to build new potential access paths \( \pi \in \text{PAP}_g(o', \mu', \omega') \). However, for each of them there is a precursor \( \pi' \in \text{PAP}_g(o', \mu', \omega') \):
\begin{itemize}
\item If \( \mu_r = \text{free} \), new \( \mu' \)-paths can only be extensions \( \pi = s \xrightarrow{\mu \nu \nu \nu} o \xrightarrow{\nu \nu} \omega \) of the new edge by co-edges in \( g' \) (actually, in \( g'' \)). The co-edges must be old, since the only new edge in \( g' \) has mode \( \mu_r \neq \text{co} \). Hence a free path \( \pi' = s \xrightarrow{\mu \nu \nu \nu} r \xrightarrow{\nu \nu} o \xrightarrow{\nu \nu} \omega \) existed already in \( g \). By \( g \models \text{UH} \) it ensured that the free call-link \( s \xrightarrow{\mu \nu \nu \nu} r \) is the initial edge of all ownership paths to \( \omega \) and has multiplicity one. Consequently, decreasing the call-link’s multiplicity in \( g' \) destroys all old ownership paths to \( o \) and \( \omega \). Hence the multiplicity of the new free edge \( s \xrightarrow{\mu \nu \nu \nu} o \) must be 1, and must be the start of all new ownership paths to \( \omega' \). There is neither a UH- nor UO-conflict.
\item If \( \mu_r = \text{rep} \) then, analogously, the new \( \mu' \)-paths can only be extensions \( \pi = s \xrightarrow{\mu \nu \nu \nu} o \xrightarrow{\nu \nu} \omega \) of the new edge by old co-edges in \( g' \), and there was an old rep path \( \pi' = s \xrightarrow{\mu \nu \nu \nu} r \xrightarrow{\nu \nu} o \xrightarrow{\nu \nu} \omega \) in \( g \). This path by \( g \models \text{UH} \) excludes any old free path \( \pi'' \) to \( \omega' \). Since the only new potential access paths in \( g' \) have mode \( \mu_r \neq \text{free} \), there is no new UH-conflict. And by \( g \models \text{UO} \), rep path \( \pi' \) ensured that all old
ownership paths to \( \omega' \) have source \( s \). Since also all new ownership paths \( \pi \) have source have source \( s \), there is no UO-conflict.

- If \( \mu_e = \text{co} \) then, besides the \( h = s \xrightarrow{co} o_i \), also the multiplicity of its implicit inverse \( h^{-1} = s \xrightarrow{co} o \) in \( g^* \) is increased. These edges have precursors \( \pi_h = s \xrightarrow{co} r \xrightarrow{co} o \) and \( \pi_h^{-1} = s \xrightarrow{co} r \xrightarrow{co} o \) in \( g^* \). All new ownership paths \( \pi \) in \( g^* \) must contain \( h \) or \( h^{-1} \) as non-head edge. But they all have a precursor in \( g^* \) with \( \pi_h \) and \( \pi_h^{-1} \) in place of \( h \) and \( h^{-1} \). There can be no new UH- nor UO-conflict.

\[
\{ \text{call} \} \quad \text{Operation calls are the most tedious case. If } \hat{e} = \langle s, \hat{\mu}, r \rangle \quad \Rightarrow \quad \text{om}(r) = \langle g_r, F \rangle \quad \text{and} \quad F(f) = \kappa^* \mu \quad \Rightarrow \quad (x_i \mu_i \xi_i \{ \ldots \} \{ \ldots \}) \quad \text{then the multiplicity of self-link } \xi_i \quad \text{and received handles } r \xrightarrow{\mu_i} o_i \quad \text{is increased while that of sent handles } s \xrightarrow{\text{sent}} o_i \quad \text{is decreased (unless } o_i = \text{nil}) \quad \text{Note that handles } (r, \hat{\mu}_i, o_i) \text{ in term } e \text{ mean that the removed edges indeed existed in } g = ogr(c, \vec{r}, s) \quad \text{Proposition 2). If there are } n \text{ parameters, the new graph } g' \text{ is the final graph } g_n \text{ in the sequence } g, g_0, \ldots, g_n \text{ of graphs with } g_0 = g \oplus r \xrightarrow{co} r \quad \text{and} \quad g_i = g_{i-1} \ominus s \xrightarrow{\mu_i} o_i \oplus r \xrightarrow{\mu_i} o_i \quad \text{for } i > 0 \quad \text{Show } g_i \models \text{UH, UO for } i = 1, \ldots, n \quad \text{by induction on the number } k \quad \text{of non-null arguments.}

\]

Let \( i \) be the index of the last, the \( k^{th} \) non-null argument, so that all graphs following \( g_i \) are not actually changed: \( g_i = g_{i+1} = \ldots = g_n = g' \). 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 \( \pi \in \text{PAP}_{g'}(\omega', \mu', \omega') \) containing it, there was already a potential access path \( \pi' \in \text{PAP}_{g'}(\omega', \mu', \omega') \) with the self-link cut out and the same head in \( g' \). In the induction step \( k-1 \rightarrow k \), the graph \( g_{i-1} \) with \( k-1 \) transferred handles still satisfies UH and UO by induction hypothesis. The question is, if \( g_i = g_{i-1} \ominus s \xrightarrow{\mu_i} o_i \oplus r \xrightarrow{\mu_i} o_i \), preserves them.

Let us derive how the received handle’s mode \( \mu_i \) must relate to the sent handle’s modes \( \mu_{i-1} \). Typing \( \hat{\Gamma}, \hat{\kappa} \vdash \hat{e} : \hat{\tau} \) of the redex means three things:

- \( \hat{\Gamma}, \hat{\kappa} \vdash \langle s, \hat{\mu}, r \rangle : \hat{\mu} \xrightarrow{c} \land \text{om}(r) = \langle g_r, F \rangle \)
- \( \langle f : \hat{\tau}, r \rangle \in \Sigma(\hat{\mu} \xrightarrow{c}) \quad \Rightarrow \quad \tau_i = \hat{\mu} \xrightarrow{\mu_i} d_i \quad \land \quad \mu_i \not\equiv \text{rep} \quad \land \quad \mu_i \leq_m \hat{\mu} \xrightarrow{\mu_i} d_i \quad \land \quad \hat{\mu} \xrightarrow{\mu_i} d_i \quad \leq_m \tau_i \)

Observe above that \( \mu_i \neq \text{rep} \). This leaves \( \mu_i = \text{free} \) and \( \text{co} \) as relevant cases.

- If \( \mu_i = \text{free} \) then \( \hat{\mu} \xrightarrow{\mu_i} \text{free} \); so that \( \mu_i \leq_m \hat{\mu} \xrightarrow{\mu_i} \mu_i \) must be free. But if sent handle \( s \xrightarrow{\text{sent}} o_i \) in \( g_{i-1} \) is free, then by induction hypothesis \( g_{i-1} \models \text{UH all ownership paths to } o_i \) and its co-objects started with this handle. All these paths will be destroyed in \( g_i \), by the argument links’ removal. And the only new ownership paths in \( g_i \) are those through free handle \( r \xrightarrow{\mu_i} o_i \). Hence there can be no new UH- nor UO-conflict in \( g_i \).

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

First, consider what the sent handle tells us about ownership paths to \( o_i \). On one side, the rule for handle-signature \( \Sigma (\mu, c) \) with parameter mode \( \mu_i = \text{co} \) ensures that neither the call-link’s mode \( \mu = \mu' \) nor the sent handle’s mode \( \mu_i \) are \( \text{read} \). On the other side, argument link \( s \cdot \mu \cdot o_i \) tells us about ownership paths to \( o_i \).

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

Second, consider the receiver expression \( (s, \mu, r) \) in \( e \). tells us about ownership paths to \( r \): By \( g = ogr( e, \bar{\mu}, s ) \) (Proposition 2), there must be a corresponding edge \( s \cdot \mu \cdot r \) in \( g \), the call-link. Since it is not removed, it still exists in \( g_{i-1} \):

- If \( \mu = \mu' = \text{free} \) then the \( \text{free} \) call-link \( s \cdot \mu \cdot r \) in \( g_{i-1} \) means by induction hypothesis \( g_{i-1} \models \text{UH} \) that it was the head of all old ownership paths to \( r \) and its co-objects. Hence all extensions \( \pi = \pi' \cdot \pi'' \) of unchanged ownership paths \( \pi' \) with target \( r \) must start with call-link \( s \cdot \mu \cdot r \). On the other side, \( \mu_i \) is \( \text{free} \). Hence, as shown above, there are no other new ownership paths, and the old ownership paths to \( o_i \) and its old co-objects have disappeared in \( g_i \). The only new ownership paths are \( \text{free} \) with initial edge \( s \cdot \mu \cdot r \), and the only unchanged ownership paths with the same targets (\( o_i \) and \( r \) and their co-objects) are \( \text{free} \) paths with initial edge \( s \cdot \mu \cdot r \) (targeting \( r \) and its co-objects). There is no new \( \text{UH} \)- or \( \text{UO} \)-conflict.
- If \( \mu = \mu' = \text{rep} \) then the \( \text{rep} \) call-link \( s \cdot \mu \cdot r \) in \( g_{i-1} \) means by induction hypothesis \( g_{i-1} \models \text{UH} \) that all unchanged ownership paths to \( r \) and its co-objects are \( \text{rep} \) paths. And by induction hypothesis \( g_{i-1} \models \text{UO} \), all these ownership paths must
have source s. Consequently, all extensions \( \pi = \pi' \cdot r \cdot \omega \cdot o_i \cdot \pi'' \) of unchanged \( r \)-targeting ownership paths \( \pi' \) are \texttt{rep} paths \( a \), and \( s \) is their source \( b \). On the other side, \( \hat{\mu}_i \leq_m \hat{\mu}' \cdot o_i \) means that \( \hat{\mu}_i \) is \texttt{rep} or \texttt{free}. In case of \texttt{rep}, as shown above, all unchanged ownership paths to \( o_i \) and its co-objects are \texttt{rep} paths \( a \), with source \( s \) \( b \), and also all new ownership paths by extending them are \texttt{rep} \( a \), and have source \( s \) \( b \). In case of \texttt{free}, as shown above, there are no other new ownership paths, and the old ownership paths to \( o_i \) and its old co-objects have disappeared. That is, in both cases, all new ownership paths are \texttt{rep} \( a \) with source \( s \) \( b \), and the unchanged ownership paths to their targets \( (o_i \text{ and } r \text{ and their co-objects}) \) are also \texttt{rep} \( a \) with source \( s \) \( b \). There is no new \texttt{UH}-conflict and no new \texttt{UO}-conflict.

- If \( \hat{\mu} = co \) then \( \hat{\mu}_i \leq_m \hat{\mu}' \cdot o_i \) \( = co \) means that \( \hat{\mu}_i \) is \texttt{co} or \texttt{free}. If \texttt{free} then, as shown above, the old ownership paths to \( o_i \) and its old co-objects have disappeared in \( g_i \) and cannot give raise to new ownership paths: All new ownership paths \( \pi \) are extensions of unchanged ownership paths \( \pi' \in P =_st \cup_{\delta'} \text{PAP}_{g_i}(o', \texttt{free}, r) \cup \text{PAP}_{g_i}(o', \texttt{rep}, r) \) to \( r \), and thus have the same initial edges \( h \in H =_st \text{first}(P) \) (there are no new \texttt{free} or \texttt{rep} edges in \( g_i \)). And all unchanged ownership paths \( \pi'' \) with the same targets, namely \( r \) and its old co-objects, also have initial edges \( h \in H \) since they are themselves \( r \)-targeting paths in \( P \), or since they can be extended by the old \texttt{co}-links between \( r \) and this co-object to an \( r \)-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 \texttt{UH}- nor \texttt{UO}-conflicts: Induction hypothesis \( g_{i-1} \models \text{UH} \) ensures that if one \( h \in H \) is a \texttt{free} handle, then there is no other handle in \( H \), and \( h \)’s multiplicity is one. And induction hypothesis \( g_{i-1} \models \text{UO} \) 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 \texttt{UH}- and no \texttt{UO}-conflict.

If \( \hat{\mu}_i = co \), then consider that the \texttt{co}-call-link \( s \xleftarrow{\omega} r \) has an inverse \( r \xleftarrow{\omega} s \) in \( g_{i-1} \). Hence every new potential access path \( \pi \in \text{PAP}_{g_i}(o', \mu', \omega') \) in \( g_i \) has a precursor \( \pi' \in \text{PAP}_{g_{i-1}}(o', \mu', \omega') \) in \( g_{i-1} \) where for the new parameter link \( r \xleftarrow{\omega} o_i \) one substitutes the pair \( r \xleftarrow{\omega} s \xleftarrow{\omega} o_i \) of the inverse call-link and the argument link, and for the new inverse parameter link \( o_i \xleftarrow{\omega} r \) one substitutes the pair \( o_i \xleftarrow{\omega} s \xleftarrow{\omega} r \) of the inverse argument link and the call-link. The precursor \( \pi' \) of a new \texttt{free} or \texttt{rep} path \( \pi \) moreover must have the same initial edge since the only new edges in \( g_i \) have mode \texttt{co}. Hence \( g_i \models \text{UH} \) and \( g_i \models \text{UO} \) follow directly from induction hypothesis \( g_{i-1} \models \text{UH} \) and \( g_{i-1} \models \text{UO} \).
5.5.2 Structural Integrity of Mutator Access

Theorem 3 If $e_0, \eta_0, s_0, om_0, g_0 \rightarrow^* e, \eta, s, om, g$ is a reduction defined relative to a program $p$ with $\vdash p$ start $e_0$ then

$$g, \eta \models MCP$$

Proof by induction on the number $N$ of reduction steps from $e_0$ to $e$: In the base case $N = 0$, $g$ is the empty graph $g_0 = \emptyset$ which trivially satisfies MCP with any environment stack. In the induction step $N \rightarrow N + 1$, execution $e_0, \eta_0, s_0, om_0, g_0 \rightarrow^* e_N, \eta_N, s_N, om_N, g_N$ is continued $e_N, \eta_N, s_N, om_N, g_N \rightarrow^* e, \eta, s, om, g$. Let $\eta = \eta_{1h_1} \cdots \eta_{nh_n}$ with $h_i = (\omega_{i-1}, \mu_i, \omega_i)$.

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

Consequently, in all reduction steps where $n = k$ or $n = k - 1$, the induction hypothesis guarantees $g, \eta \models MCP$. 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., $\kappa_n = \text{mut}$, if $\kappa^*$ in the called method $F(f) = \kappa^* t f (\ldots) \{\ldots\}$ for $om(r) = (\eta_{r}, F)$ is \text{mut}. The term's typing $\Gamma_{n}, \kappa \vdash X : \tau$ (Theorem 1) with $\mu_r, \Gamma, \kappa, X = \mu_r, \Gamma_i, \kappa_i$, implies a typing $\Gamma_n, \kappa_n \vdash \hat{e} : \hat{\tau}$ 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 $\Gamma_n, \kappa_n, e$ since $\eta_N \models \bar{\mu}_i, \Gamma, \kappa, X$ (Theorem 1). Typing the call expression $\hat{e}$ required a typing $\Gamma_n, \kappa_n \vdash (s, \mu_r, r) : \mu_r c$ for the receiver expression. Hence $om(r) = (\eta_{r}, F)$ with $\models om$ (Theorem 1) means by Lemma 2 that $\text{FldsMths}(c) = (\Gamma_c, F)$ and $\mu_r = \mu_r$. But then $(f : \ldots) \Rightarrow (\ldots) \Rightarrow \tau' \in \Sigma(\mu_r, c)$ with the same kind $\kappa^*$ as $F(f)$. Therefore typing $\hat{e}$ ensured in case of $\kappa^* = \text{mut}$ that $\mu_r = \bar{\mu}_r \in \text{Wr}(\kappa_n)$. This leaves two cases:

- If $\mu_r$ is free or rep, then the call-link $h = s \Rightarrow r$ in $g$ is the necessary ownership path for $r$: $h \in PAP_g(s, \mu_r, r)$. ($h \in g$ follows with $g = ogr(e, \eta, s)$ from $h$ as call-link in the new top-level environment in $\eta$.)

- $\mu_r$ can be co only if $\kappa_n$ is mut. But then induction hypothesis $g_N, \eta_N \models MCP$ ensures an ownership path $\pi = h_j \cdots h_n \in PAP_g(\omega_j, \mu', s)$. As explained above, $\pi$ still exists in $g$ since it consists of call-links in $\eta_N$. The co call-link in $g$ extends $\pi$ to the necessary ownership path for $r$: $\pi \cdot h \in PAP_g(\omega_j, \mu', r)$.

Theorem 4 If $e_0, \eta_0, s_0, om_0, g_0 \rightarrow^* e, \eta, s, om, g$ is a reduction defined relative to a program $p$ with $\vdash p$ start $e_0$ then

$$g, \eta \models MC$$

113
Proof by induction on the number \( N \) of reduction steps from \( e_0 \) to \( e \): In the base case \( N = 0 \), \( g \) is the empty graph \( g_0 = \emptyset \), which trivially satisfies MC with any environment stack. In the induction step \( N \to N + 1 \), execution \( e_0, \eta_0, s_0, o_{m_0}, g_0 \Rightarrow * e_N, \eta_N, s_N, o_{m_N}, g_N \) is continued \( e_N, \eta_N, s_N, o_{m_N}, g_N \Rightarrow e, \eta, s, o_m, g \). Let \( \eta = \eta_{h_1} \cdots \eta_{h_n} \) with \( h_i = (\omega_{i-1}, \mu_i, \omega_i) \).

Proceed by reductio ad absurdum. Assume a violation of MC by \( g \) and \( \eta \). Then there must be a call-level \( i \) in \( \eta \) at which the receiver \( \omega_i \) is executing a mutator \( (\kappa_i = \text{mut}) \), and there is a “non-controlling” representative \( o_1 \) which has \( \omega_i \) in its sanctuary \( Sanc_g(o_1) \) but does not execute a mutator at a lower call-level \( (o_1 \notin \{\omega_1, \ldots, \omega_i\}) \).

The inductive definition of \( \omega_i \in Sanc_g(o_1) \) based on \( \text{rep} \) paths obviously requires a non-empty sequence \( \pi_1 \cdots \pi_k \) of \( \text{rep} \) paths \( \pi_j \in \text{PAP}_g(o_j, \text{rep}, o_{j+1}) \) connecting \( o_1 \) with \( \omega_i = o_{k+1} \) via objects \( o_2, \ldots, o_k, o_1 \rightarrow \text{rep} \rightarrow o_2 \rightarrow \text{rep} \rightarrow \cdots \rightarrow \text{rep} \rightarrow o_{k+1} = \omega_i \).

Show by induction on the length \( k \) of the shortest connecting \( \text{rep} \) 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 \( \omega_i \) cannot be the non-controlling representative \( o_1 \) since \( \kappa_i = \text{mut} \).

In the induction step \( k-1 \to k \), sequence \( o_1 \rightarrow \text{rep} \rightarrow o_2 \cdots o_{k-1} \rightarrow \text{rep} \rightarrow o_k \) is extended by \( \pi_k = o_k \rightarrow \text{rep} \rightarrow o_{k+1} \in \text{PAP}_g(o_k, \text{rep}, \omega_i) \).

1. \( g_N, \eta_N \models \text{MCP} \) (Theorem 3) guarantees for \( \kappa_i = \text{mut} \) some \( j \) such that \( \pi = h_j \cdots h_1 \) is an ownership path from \( \omega_{j-1} \) to \( \omega_i \).

2. \( \pi_k \in \text{PAP}_g(o_k, \text{rep}, \omega_i) \) means by \( g \models \text{UO} \) (Theorem 3) that all ownership paths to \( \omega_i \) start with \( o_k \). Hence \( \omega_{j-1} = o_k \).

3. By \( g \models \text{UH} \) (Theorem 3), \( \text{rep} \) path \( \pi_k \) to \( \omega_i \) guarantees that there is no free path to \( \omega_i \). Consequently, \( \pi \in \text{PAP}_g(o_k, \text{rep}, \omega_i) \) and \( \omega_i \in Sanc_g(o_k) \).

4. If, on one hand, call-level \( i \) existed already in \( \eta_N \), i.e., \( i \leq n \), then \( \text{rep} \) path \( \pi = h_j \cdots h_1 \) already existed in \( \eta_N \), and thus in \( g_N \) by virtue of \( g_N = ogr(e_N, \eta_N, s_N) \) (Proposition 2). But then \( \omega_i \in Sanc_g(o_k) \), so that the outer induction hypothesis \( g_N, \eta_N \models \text{MC} \) guarantees a mutator-execution by \( o_k \) at a call-level \( j \leq i \) in \( \eta_N \), and thus in \( \eta \).

5. If, on the other hand, call-level \( i \) is new in \( \eta \), then it must be the new top-level \( i = n = n_N + 1 \) in a \{call\}-step: \( \eta = \eta_N \cdot \eta^*_{s, \mu, r} \) for redex \( \hat{e} = (s, \mu, r) \Rightarrow f(\ldots) \), with \( \kappa_i = \kappa_n = \kappa,r \) and \( \omega_i = \omega_n = r \) and \( \omega_{i-1} = \omega_n = s \). 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 \( \mu_r \in \text{Wr}(\kappa_n) \) if \( \kappa_i = \kappa_n = \kappa^* \) is \text{mut}. \( \mu_r \) cannot be free, since call-link \( (s, \mu, r) \) in \( \eta \) would by \( g = ogr(e, \eta, s) \) (Proposition 2) mean a free path \( s \mu, r \in \text{PAP}_g(s, \text{free}, \omega_i) \) to \( \omega_i \), in contradiction to step 3. If \( \kappa_n = \text{mut} \) and \( \mu_r = \text{rep} \), then the call-link \( s \rightarrow \text{rep} \rightarrow r \) in \( g \) is the path \( \pi_k \), i.e., \( o_k = s \rightarrow \text{rep} \rightarrow r = \omega_i \). This means that \( o_k = s = \omega_n \) is executing a mutator at level \( j = n \leq n \). And if \( \kappa_n = \text{mut} \) and \( \mu_r = \text{co} \), then \( \pi_k \) is extended by the call-link's inverse \( r \rightarrow \text{co} \rightarrow s \) to a \( \text{rep} \) path from \( o_k \) to \( s \) in \( g^* \), so that \( s \in Sanc_g(o_k) \). But then, as shown in the previous step, \( s = \omega_n \) with \( \kappa_n = \text{mut} \) 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 \( \eta \). Since \( o_k \in \text{SanC}_g(o_1) \) through rep paths \( \pi_1 \ldots \pi_{k-1} \), 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 \( e_0, \eta_0, s_0, om_0, g_0 \Rightarrow^* e, \eta, s, om, g \Rightarrow e', \eta', s', om', g' \) is a reduction defined relative to a program \( p \) with \( \vdash p \) start \( e_0 \) then for all \( o \in \text{dom}(om) \),

\[
\text{CState}_{s,om}(o) \neq \text{CState}_{s',om'}(o) \Rightarrow \exists i \leq n. r_i = o \land \kappa_i = \text{mut}
\]

where \( \eta = \eta_1^{h_1}, \ldots, \eta_n^{h_n} \) with \( h_i = (s_i, \mu_i, r_i) \).

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

\[
\text{CState}_{s,om}(o) \neq \text{CState}_{s',om'}(o)
\]

\[
\text{Lemma 3} \quad \exists \omega \in \text{StRep}_{s,om}(o). s |_{\text{fds}_{om}(\omega)} \neq s' |_{\text{fds}_{om}(\omega)}
\]

\[
\text{Lemma 4} \quad \exists \omega \in \text{StRep}_{s,om}(o). r_n = \omega \land \kappa_n = \text{mut}
\]

\[
\Rightarrow \quad r_n \in \text{StRep}_{s,om}(o) \land \kappa_n = \text{mut}
\]

\[
\text{Proposition 4} \quad r_n \in \{o\} \cup \text{SanC}_{\text{ogr}(e,\eta),s}(o) \land \kappa_n = \text{mut}
\]

\[
\Rightarrow \quad (r_n = o \land \kappa_n = \text{mut}) \lor (r_n \in \text{SanC}_g(o) \land \kappa_n = \text{mut})
\]

\[
\text{Theorem 4} \quad (r_n = o \land \kappa_n = \text{mut}) \lor (\exists i \leq n. r_i = o \land \kappa_i = \text{mut})
\]

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

**Lemma 3** If \( e_0, \eta_0, s_0, om_0, g_0 \Rightarrow^* e, \eta, s, om, g \Rightarrow e', \eta', s', om', g' \) is a reduction defined relative to a program \( p \) with \( \vdash p \) start \( e_0 \) then

\[
\text{CState}_{s,om}(o) \neq \text{CState}_{s',om'}(o) \Rightarrow \exists \omega \in \text{StRep}_{s,om}(o). s |_{\text{fds}_{om}(\omega)} \neq s' |_{\text{fds}_{om}(\omega)}
\]

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

\[
s |_{\bigcup_{\omega \in \text{StRep}_{s,om}(o)} \text{fds}_{om}(\omega)} \neq s' |_{\bigcup_{\omega \in \text{StRep}_{s',om'}(o)} \text{fds}_{om'}(\omega)}
\]
In the simple case, the domain of the restriction is unchanged: \( L = \bigcup_{\omega \in \text{StRep}_s, \text{om}(o)} \text{flds}_\text{om}(\omega) = \bigcup_{\omega \in \text{StRep}_s', \text{om'}(o)} \text{flds}_\text{om'}(\omega) = L' \). Then the composite state change \( s |_{L} \neq s' \big|_{L'} \) means that the store changed at some location \( \ell \in L = L' \): \( s(\ell) \neq s'(\ell) \). It must be the field location \( \ell \in \text{flds}_\text{om}(\omega) \) of some object \( \omega \in \text{StRep}_s, \text{om}(o) \). Since \( s(\ell) \neq s'(\ell) \), \( s \big|_{\text{flds}_\text{om}(\omega)} \neq s' \big|_{\text{flds}_\text{om}(\omega)} \) for this \( \omega \).

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

\[
\{ o \} \bigcup_{\text{PAP}_{\text{fgrom}(s)}(o, \text{rep}, \omega) \neq 0} \text{StRep}_s, \text{om}(o) \neq \{ o \} \bigcup_{\text{PAP}_{\text{fgrom'}(s')}(o, \text{rep}, \omega) \neq 0} \text{StRep}_s', \text{om'}(o)
\]

That is, there must be an object \( q \) that is reachable from \( o \) by a non-empty sequence \( o = o_0 \xrightarrow{\text{rep}} o_1 \xrightarrow{\text{rep}} \ldots \xrightarrow{\text{rep}} o_n = q \) of \( \text{rep} \) paths in field subgraph \( \text{fgrom}(s) \) but no such sequence in \( \text{fgrom'}(s') \), or vice versa. Each of the \( \text{rep} \) paths \( o_i \xrightarrow{\text{rep}} o_{i+1} \) is in base-JaM a \( \text{rep} \) edge followed by \( \text{co} \) edges: \( o_i = o_{i,0} \xrightarrow{\text{co}} o_{i,1} \xrightarrow{\text{co}} \ldots o_{i,k_i-1} \xrightarrow{\text{co}} o_{i,k_i} = o_{i+1} \). In order for the path sequence to exist in \( \text{fgrom}(s) \) but not in \( \text{fgrom'}(s') \), or vice versa, there must be a left-most \( \text{rep} \) or \( \text{co} \) edge \( o_{i,j} \xrightarrow{\text{rep}} o_{i,j+1} \) or \( o_{i,j} \xrightarrow{\text{co}} o_{i+1,0} \) that appears in, or disappears from, the field subgraph. That is, a handle \( (o_{i,j}, \mu, o_{i,j+1}) \) or \( (o_{i,j}, \mu, o_{i+1,0}) \) is captured in, or removed from, a field location \( \ell \). By source consistency \( \models_s \text{om} \) (Proposition 1), this field must belong to the handle’s source \( o_{i,j} \); \( \ell \in \text{flds}_\text{om}(o_{i,j}) \). Since non-empty prefixes of \( \text{rep} \) paths are also \( \text{rep} \) paths, there is an unchanged \( \text{rep} \) 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: \( o_{i,j} \in \text{StRep}_s, \text{om}(o) \cap \text{StRep}_s', \text{om'}(o) \). Since \( o_{i,j} \) has a changed field, \( s \big|_{\text{flds}_\text{om}(o_{i,j})} \neq s' \big|_{\text{flds}_\text{om}(o_{i,j})} \), it is the desired object \( \omega \).

**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 \( e_0, \eta_0, s_0, \text{om}_0, g_0 \xrightarrow{\ast} e, \eta, s, \text{om}, g \xrightarrow{\prime} e', \eta', s', \text{om}', g' \) is a reduction defined relative to a program \( p \) with \( \vdash p \) start \( e_0 \) then for \( \eta = \eta_1^{\kappa_1}, \ldots, \eta_n^{\kappa_n} \) with \( \kappa_n = \text{dom}(\text{om}) \), and for all \( \omega \in \text{dom}(\text{om}) \),

\[
s \big|_{\text{flds}_\text{om}(\omega)} \neq s' \big|_{\text{flds}_\text{om}(\omega)} \Rightarrow r_n = \omega \wedge \kappa_n = \text{mut}
\]

**Proof:** The proof is based on two invariants holding in configuration \( e, \eta, s, \text{om}, g \). Let \( \eta = \eta_1^{\kappa_1}, \ldots, \eta_n^{\kappa_n} \) with \( \kappa_n = \{ s_1, t_1, r_i \} \).
Fields are not aliased by local identifiers: \( \text{locals}(\vec{\eta}) \cap \text{flds}(om) = \emptyset \)
where \( \text{locals}(\vec{\eta}) \) is the set \( \bigcup_{\eta \in P} \text{im}(\eta) \) of locations of all local variables.

Field locations of object \( o \) occur at "mutable positions" only in mutators of \( o \):
\[ \text{mutlocs}(E_i) \cap \text{flds}_{om}(o) \neq \emptyset \Rightarrow r_i = o \land \kappa_i = \text{mut} \]
where \( \text{mutlocs}(E_i) \) 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 \( E_i \) 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 \( e, \vec{\eta}, s, om, g \Rightarrow e', \vec{\eta}', s', om', g' \) is defined must contain \( n \) nesting levels of inlined methods. Hence it can be decomposed by a series of reduction context \( \varepsilon_1, \ldots, \varepsilon_{n-1} \in R^\square_1 \) and an innermost runtime term \( e_n \) containing no inlined method body such that \( e = \varepsilon_1[\varepsilon_2[\ldots[\varepsilon_{n-1}[e_n]]\ldots]] \). For uniformity, let us write \( \varepsilon_n \) for \( e_n \).

The set \( \text{mutlocs}(e) \) 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.

\[
\begin{align*}
\text{mutlocs}(x) &= \begin{cases} 
\text{dr} & \text{if } x \neq \text{this} \cdot x \\
\emptyset & \text{otherwise}
\end{cases} \\
\text{mutlocs}((\text{null})) &= \begin{cases} 
\text{dr} & \text{if } e \neq \text{this} \cdot x \\
\emptyset & \text{otherwise}
\end{cases} \\
\text{mutlocs}(e_1 e_2) &= \text{dr} \cup \text{mutlocs}(e_1) \cup \text{mutlocs}(e_2)
\end{align*}
\]

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 \( e = e', \text{val}(e), \text{destval}(e) \) 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 \( e, \vec{\eta}, s, om, g \). In the base case \( N = 0 \), there can be no \( t \in \text{flds}(om) \) and no \( o \in \text{dom}(om) \) since \( om = om_0 = \emptyset \). Hence invariants I1 and I2 are trivial. In the induction step \( N \rightarrow N + 1 \), execution \( e_0, \eta_0, s_0, om_0, g_0 \Rightarrow e^* \) produces \( e_N, \vec{\eta}_N, s_N, om_N, g_N \) such that \( e_N, \vec{\eta}_N, s_N, om_N, g_N \Rightarrow e, \vec{\eta}, s, om, g \).

Ad I1. Since \( \text{locals}(\vec{\eta}_N) \cap \text{flds}(om_N) = \emptyset \) by induction hypothesis, condition \( \text{locals}(\vec{\eta}) \cap \text{flds}(om) = \emptyset \) 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 \( \text{locals}(\vec{\eta}) \cap \text{flds}(om) = \emptyset \).

Ad I2. The induction hypothesis means that
\[ e_N = \varepsilon_1[\varepsilon_2[\ldots[\varepsilon_{n-1}[e_{n-1}]]\ldots]] \]
with \( \text{mutlocs}(E_i) \cap \text{flds}_{om}(o) \neq \emptyset \Rightarrow r_i = o \land \kappa_i = \text{mut} \) for \( o \in \text{dom}(om_N) \). The
invariant is unaffected by all steps in which there are no new locations in the term e, and in which om and \( \eta \) are unchanged. In case of \{ new \}, there is a new object in \( \text{dom}(om) \), but the locations of its fields are fresh, and thus cannot occur in e. In case of \{ call \}, nothing is removed from \( \eta_N \). 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 \( r_n \) and method kind \( \kappa_n \) 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 \}, the only change is that e contains the additional location \( \text{TJ}_n(x) \).

By induction hypothesis \( I \), locals(\( \eta_N \)) \( \cap \) fds(om\( N \)) = \( \emptyset \), so that this location cannot be the location of any object’s field.

The really interesting case is \{ var \}. Here the redex \( \hat{e} = \text{this}.x \) in \( e_N \) reduces to the location \( \ell \in \text{fds}_{om}(o) \) of o’s x-field, where o is the target of the top-level this-handle: \( s(\eta_n(\text{this})) = (o, \mu, o) \). By source consistency \( \models_s \eta \) (Proposition 1), the source o of a handle at a location in top-level environment \( \eta_n^\kappa_n \) implies that o is the top-level receiver \( r_n \). If the redex \( \hat{e} \) in \( e_N \) is a right-hand side or destructively read expression, i.e., \( \mathcal{E}_n = \mathcal{E}[^\text{destval}(\hat{e})] \) or \( \mathcal{E}_n = \mathcal{E}[\ell = \ell_2] \), then \( \mathcal{E}_n' = \mathcal{E}[\text{destval}(\ell)] \) or \( \mathcal{E}_n' = \mathcal{E}[\ell = \ell_2] \), respectively means there is a new field location \( \ell \) in \( \text{mutlocs}(\mathcal{E}_n') \). The typeability of \( e_N \) guaranteed by Theorem 1 required the typeability, in particular, of redex \( \text{destval}(\hat{e}) \) or \( \hat{e} = \ell_2 \), respectively in the context of \( r_n, \kappa_n \). But then \( \hat{e} = \text{this}.x \) implied \( \kappa_n = \text{mut} \). Since \( r_n = o \) and \( \kappa_n = \text{mut} \), and since the other levels of term and environment stack are unchanged, invariant I2 is preserved.

With invariants \( I_1 \) and I2 holding in \( e, \eta, s, om, g \), it is now easy to show shallow state encapsulation: Consider the cases of step e, \( \eta', s, om, g \) \( \longrightarrow \) e', \( \eta', s', om', g' \) 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 s nor om. In case of \{ call \} and \{ new \}, om is unchanged or changes only for a fresh object identifier, and 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 \( \eta_n \in \eta \). This changes no object’s fields since locals(\( \eta \)) \( \cap \) fds(om\( N \)) = \( \emptyset \) by I1. Hence shallow state encapsulation is trivial.

The two steps where s |\( \text{fd}_{om}(\omega) \) \( \neq \) s' |\( \text{fd}_{om}(\omega) \) are \{ rd, \text{del} \}-steps with \( \hat{e} = \text{destval}(\ell) \) and \{ upd \}-steps with \( \hat{e} = \ell = \ell_2 \) such that \( \ell \in \text{fds}_{om}(\omega) = \text{fds}_{om}(\omega) \) for some \( \omega \). In this case, the redex’s reduction is \( \hat{e}, \eta, s, om, g \) \( \longrightarrow \) e', \( \eta', s', om', g' \) with \( \hat{e} \) at nesting level \( n \) in e and with \( \eta' = \eta_n \) and \( \eta' = \eta_n' \). The I2 guarantees the desired r\( n = o \) \( \wedge \kappa_n = \text{mut} \).
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 $\alpha \in A$ are added to the set of modes, and these base-modes are parameterized by correlations $\delta$ to specify the mode of $\mu$-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 $\mu$-paths to non-$\mu$ 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 $\alpha \overset{\text{co}}{\rightarrow} q \overset{\alpha}{\rightarrow} \omega$ and $\alpha \overset{\text{co}}{\rightarrow} q \overset{\alpha}{\rightarrow} \omega$ of co- and association paths by association paths, nor the extension $\alpha \overset{\text{free}}{\rightarrow} q \overset{\alpha}{\rightarrow} \omega$ of potential access paths by association paths to co- and free paths $\alpha \overset{\text{co}}{\rightarrow} \omega$ and $\alpha \overset{\text{free}}{\rightarrow} \omega$. This simplification is reflected in constraints on the nesting structure of mode-terms: Only modes free, rep and read are parameterized by correlations ($\text{free}^{\delta}$, $\text{rep}^{\delta}$ and $\text{read}^{\delta}$, but co$<> \text{ and } \beta<>$), and only by correlations to rep, read and association modes ($m<>\alpha=\text{rep}^{\delta}<>\ldots<>$, $m<>\alpha=\text{read}^{\delta}<>\ldots<>$ and $m<>\alpha=\gamma<>\ldots<>$).

- Implicit mode-conversions from free$<>$ to co$<>$ or $\alpha <>$ caused by assignment or parameter supply will not be considered. (Tedious invariants about all sequences $\omega \overset{\text{co}}{\rightarrow} q$ of association paths starting from targets of free$<>$ paths $\overset{\text{free}^{\delta}}{\rightarrow} \omega$ 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 $\preceq_m$. 

119
• 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 A of identifiers α called association roles to the set \( B = \{ \text{free, rep, co, read} \} \cup A \) of base-modes. Each base-mode \( m \in B \) in then parameterized by the annotation of a (possible) empty set \( \delta \) of correlations \( \alpha_1 = \mu_1 \) to a full mode \( \mu = m<\alpha_1 = \mu_1, \ldots, \alpha_n = \mu_n> \).

A correlation \( \alpha = \mu' \) on a reference or path \( \pi = q \overset{m_1 \cdots m_n}{\longrightarrow} o \) correlates o's \( \alpha \)-paths with \( q \)'s \( \mu' \)-paths, so that consequently \( \pi \) is extended by \( o \)'s association references and paths \( o \overset{\alpha}{\longrightarrow} \omega \) to paths \( q \overset{\mu}{\longrightarrow} \omega \). By itself, i.e., from the perspective of source object \( o \), classifying a reference or path \( o \rightarrow \omega \) by an association role \( \alpha \in A \) says nothing about target object \( \omega \). It does not fix \( \omega \)'s owners or sanctuary memberships. It only allows third objects \( q \) with a path \( q \overset{\mu}{\longrightarrow} \omega \) to fix \( \omega \)'s owner and sanctuary membership by correlating \( \alpha \) with an appropriate mode.

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

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

2. Example: Maps. Let \( d \) be an instance of the \texttt{MapImp} class with three entries represented as \texttt{Pair} components \( e_1, e_2 \) and \( e_3 \), and a entry-set component \( s \) of class
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 \( k_2 \). 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 \( n_1 \) is a state-representing component of set \( s \), which is a state-representing component of map \( d \); and iterator \( n_n \) 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 \( n_1 \). And the state of the map’s iteration over the set is represented in \( i \) and \( n_n \). 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 \( e_1, e_2, \) and \( e_3 \) belong to \( d \)’s state representation. Declaring the next-links of PNode objects to be rep would turn single-linked node \( n_1 \) into a composite object, that subsumes the next object \( n_2 \) 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
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 $S = \{e_1, e_2, e_3\}$ 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 s $\text{rep<elem}<$ s $\text{rep<elem}<\text{}}$. n1 specifies that the data of n1 and its co-objects are s’s elements. And the map representative’s entryset reference d $\text{rep<elem}<$ d $\text{rep<elem}<\text{}}$. 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 s $\text{rep<elem}<\text{}}$. n1 combines with n1 $\text{data}<$ e1 to a path s $\rightarrow$ e1 of mode elem<>. With this path, the entryset reference d $\text{rep<elem}<$ d $\text{rep<elem}<\text{}}$. s combines to a path d $\rightarrow$ e1 of mode $\text{rep<elem}<\text{}}$. This means that e1 is an component of d whose fst and snd paths extend d $\rightarrow$ 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 s $\text{rep<elem}<\text{}}$. n1 is extended by the next-links n1 $\text{co}$. n2 and
4. EXAMPLE DECLARATIONS. The mode-classification of the references in the object

The class of e1, e2, e3:
```java
class Pair {
    fst<> Object fst;
    snd<> Object snd;
    ...
}
```

The class of n1, n2, n3:
```java
class PNode {
    co<> PNode next;
    data<> Pair data;
    ...
}
```

The class of nn:
```java
class PNodeIt {
    dest<> PNode curdest;
    ...
}
```

The class of s:
```java
class PSetImp {
    rep<dest=read<data=dest<>>> PNode anchor;
    ...
}
```

The class of i:
```java
class PDataIt {
    rep<dest=read<data=dest<>>> PNodeIt nodes;
    ...
}
```

The class of d:
```java
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 →n2, n3 to paths s → n2 and s → n3 of the same mode. They combine with n2 →e2 and n3 →e3 to elem paths s → e2 and s → e3. With these paths, the entryset reference combines to paths d → e2 and d → 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 → 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<>>>.
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 “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 $\mathcal{M}$ that will be defined in paragraph 6): Object graph edges $(o, \mu, \omega)$, handles $(o, \mu, \omega)$ in store, runtime-term and environments; source consistencies $\models_s om$, $\models_s \eta$, and $\models_s \eta \epsilon$; term reduction contexts; handle types $\mu \epsilon$, location-partitions $\mathcal{Loc}_{\mu \epsilon}$, type extensions $[\tau]$, and type consistencies $\eta \models \Gamma$, $\models \xi$, $\models om$, and $\models \eta \models \Gamma, \kappa, \mu \epsilon, \chi$. 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

$$\text{new}<> c_n().\text{main}(), \mathcal{O}_{\text{inst,read}<>\text{,nil}}, \emptyset, \emptyset \Rightarrow e_1, \eta_1, s_1, om_1, g_1 \Rightarrow \ldots$$

*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 $\ell \in \mathcal{Loc}_{\text{co}<\epsilon}$ (instead of a base-JaM-location $\ell \in \mathcal{Loc}_{\text{co}, c}$), and a store with a handle $(r, \text{co}<>\epsilon, r)$ (instead of $(r, \text{co}, r)$) at location $\ell$, and edge $r \xrightarrow{\text{co}<>\epsilon} r$ added to the object graph (instead of $r \xrightarrow{\text{co}, r}$). Correspondingly, the typing rule [meth] types a method body in the context of type assumption *this*: ref $\text{co}<>\epsilon$ (instead of ref $\text{co} c$).
mode-term $\mu \in M := (\text{free} | \text{rep} | \text{co} | A_{x} | \text{read})^{<\Delta>}$

correlations $\delta \in \Delta := (A=M)^{*}$

expression $e \in E := \text{val}(N) | \text{destval}(N) | \text{null}^{<\Delta>} | \text{new}^{<\Delta>} C() | E \leftarrow Id(E^{*})$

$$r \in O_{c}, \quad \text{om}(r) \doteq \langle \ldots, F \rangle, \quad F(f) \doteq \kappa^{*} \tau f (\mu_{i} c_{i} y_{i}) \{\mu_{j} c_{j} z_{j}\}; s\}$$
$$\text{fresh } \ell \in [\text{ref } co<> c]], \quad \text{fresh } \ell_{o}^{q} \in [\text{ref } \mu_{i} c_{i}], \quad \text{fresh } \ell_{s}^{p} \in [\text{ref } \mu_{j} c_{j}]$$
$$\eta^{*} = \{\text{this } \mapsto \ell, y_{i} \mapsto \ell_{o}^{q}, z_{j} \mapsto \ell_{s}^{p}\}$$
$$\delta' = s[\ell \mapsto \langle r, co<> r \rangle, \ell_{o}^{q} \mapsto \langle r, \mu_{i}, o_{i} \rangle, \ell_{s}^{p} \mapsto \langle r, \mu_{j}, nil \rangle]$$

$$g' = g \ominus s \mu_{i}^{o} \mapsto o_{i} \oplus r \mu_{j}^{o} \mapsto o_{i}$$

$$\langle s, \mu_{r}, r \rangle \doteq f(\langle s, \mu_{r}^{o}, o_{i} \rangle), \quad \eta_{h}^{*}, s, \text{om}, g \longrightarrow \ll s\gg, \quad \eta_{h}^{*} \cdot \eta^{*}(s, \mu_{r}, r), s', \text{om}, g'$$

{null} $h \doteq \langle s, \mu_{r}, r \rangle$

$$\text{null}^{<\delta>}, \quad \eta_{h}^{*}, s, \text{om}, g \longrightarrow \langle r, \text{free}^{<\delta>}, \text{nil} \rangle, \quad \eta_{h}^{*}, s, \text{om}, g$$

{new} $c()$, $\eta_{h}^{*}, s, \text{om}, g \longrightarrow h', \quad \eta_{h}^{*}, s[\ell \mapsto h_{i}], \text{om}[o \mapsto \langle p, F \rangle], \quad g \ominus h'$

\[\Gamma, k \vdash \text{this:ref co<> c, } \{x_{i}: \text{ref } \mu_{i} c_{i}\}, \{x_{j}: \text{ref } \mu_{j} c_{j}\} \vdash \Gamma \text{ ok} \quad G, k \vdash \text{def } f\]

\[\Gamma, k \vdash \text{null}^{<\delta>} : \text{free}^{<\delta>} c \quad \text{[new]} \quad \Gamma, k \vdash \text{new}^{<\delta>} c() : \text{free}^{<\delta>} c\]

\[\Gamma, k \vdash \text{null}^{<\delta>} : \text{free}^{<\delta>} c \quad \text{[new]} \quad \Gamma, k \vdash \text{new}^{<\delta>} c() : \text{free}^{<\delta>} c\]

\[\Gamma, k \vdash \text{this:ref co<> c, } \{x_{i}: \text{ref } t_{i}, \{x_{j}: \text{ref } t_{j}\} \vdash \Gamma \text{ ok} \quad k \vdash \text{def } f\]

\[\Gamma, k \vdash \text{this:ref co<> c, } \{x_{i}: \text{ref } t_{i}, \{x_{j}: \text{ref } t_{j}\} \vdash \Gamma \text{ ok} \quad k \vdash \text{def } f\]

\[\Gamma, k \vdash \text{null}^{<\delta>} : \text{free}^{<\delta>} c \quad \text{[new]} \quad \Gamma, k \vdash \text{new}^{<\delta>} c() : \text{free}^{<\delta>} c\]

Figure 6.4: Changed syntax, reduction, and typing rules

- The syntax of the null-expression is extended so that it specifies the correlation-set $\delta$ to be added to the free mode of its value, the nil-handle. Reduction rule {null} reduces $e = \text{null}^{<\delta>}$ to $\langle r, \text{free}^{<\delta>}, \text{nil} \rangle$ (instead of $\langle r, \text{free}, \text{nil} \rangle$), and typing rule [null] assigns to $e = \text{null}^{<\delta>}$ the JaM-type $\text{free}^{<\delta>} c$ (instead of $\text{free} c$).

- The syntax of the creation expressions is extended to specify the correlation-set $\delta$ to be added to the free mode of its value, the handle to the new object. Reduction rule {new} reduces $e = \text{new}^{<\delta>} c()$ to $\langle r, \text{free}^{<\delta>}, \omega \rangle$ (instead of $\langle r, \text{free}, \omega \rangle$), and adds to the object-graph the edge $r \text{free}^{<\delta>} \omega$ (instead of $r \text{ free}, \omega$). Correspondingly, typing rule [new] assigns to $e$ the type $\text{free}^{<\delta>} c$ (instead of $\text{free} c$).

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 $\text{free}^{<\star>}$ that is mode-compatible to
any mode \texttt{free}\langle\delta\rangle 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 \(\leq_m\).

In the rule \([\text{rtype}]\) 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, \(\mu\) is required to be a valid mode, \(\vdash \mu\ \text{ok}\). This restriction of the mode-terms derived from nonterminal \(\mathcal{M}\) will be defined in paragraph 6. For emphasis, the rule could be rewritten:

\[
[\text{rtype}] \quad \vdash \mu\ \text{ok} \quad \vdash c\ \text{ok} \quad \vdash \mu\ c\ \text{ok}
\]

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

\[
(m\langle\delta\rangle\ c)[\text{read}/\text{free}] =_{ar} m[\text{read}/\text{free}]\langle\delta\rangle\ c\quad m[\text{read}/\text{free}] =_{ar} \begin{cases} \text{read} & \text{if } m = \text{free} \\ m & \text{otherwise} \end{cases}
\]

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 \(\text{FldsMths}(c)\) by adapting the modes \(\mu_i\) in it to \(\mu o\mu_i\).

\textbf{Proposition 5} If \(e_0, \eta_0, s_0, o_0, g_0 \Longrightarrow^* e', \eta', s', o', g'\) is a reduction defined relative to a program \(p\) with \(\vdash p\) \texttt{start} \(e_0\) then

\[
g' = \text{ogr}(e', \eta', s')
\]

\textbf{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 \(e', \eta', s'\) 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.

\textbf{Lemma 5 (Type preservation)} If \(e, \eta, s, o, g \Longrightarrow e', \eta', s', o', g'\) is a reduction defined relative to a program \(p\) with \(\vdash p\) \texttt{start} \(e_0\) then

\[
\Gamma, \kappa \vdash_X e : \tau \land \eta' \models \mu, \Gamma, \kappa, X \land s, o_m \Rightarrow \exists X'. \Gamma, \kappa \vdash_{X'} e' : \tau \land \eta' \models \mu, \Gamma, \kappa, X' \land s', o_m
\]

\textbf{Proof:} The proof from the base-JaM-version of this theorem (Lemma 1) can be copied here, except that the mode of \texttt{this}, \texttt{null}, and \texttt{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 \( e_0, \eta_0, s_0, om_0, g_0 \xrightarrow{e, \eta, s, om, g} e \), \( \text{if, s, om, g} \) is a reduction defined relative to a program \( p \) with \( \vdash p \) start \( e_0 \) then \( \exists X, \tau \).

\[
\vdash s, om \land \emptyset, \text{obs} \vdash_X \; e : \tau \land \eta \vdash \text{read}\langle\rangle, \emptyset, \text{obs}, X
\]

Proof: The proof goes the same as that for its base-JaM version (Lemma 4). The difference is that the mode is \texttt{read}\langle\rangle instead of just \texttt{read}, that the receiver expression \texttt{new}\langle\rangle \; c() in operation call expression \( e_0 \) has mode \texttt{free}\langle\rangle instead of just \texttt{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 \( \pi \in \text{PAP}(o, \mu, \omega) \) and in the definition of objects' sanctuaries \( \text{Sanc}(o) \). There are three rules for potential access paths in an object graph \( g \) labeled with full modes which are listed in figure 6.5: The base-case is an edge \( o \xrightarrow{\mu} \omega \in g \). As in base-JaM (cf. fig. 5.9 on page 88), the extension \( \pi_1 \cdot \pi_2 \) of a \( \mu \)-path \( \pi_1 \in \text{PAP}(o, \mu, q) \) by a co-path \( \pi_2 \in \text{PAP}(q, \text{co}\langle\rangle, \omega) \) is another potential access path in \( \text{PAP}(o, \mu, \omega) \).\(^1\) New is the rule for the extension by association paths. An association path \( \pi_2 \in \text{PAP}(q, \alpha\langle\rangle, \omega) \) extend a path \( \pi_1 \in \text{PAP}(o, m\langle\ldots, \alpha=\mu, \ldots, \rangle, q) \) with the necessary correlation to a potential access path \( \pi_1 \cdot \pi_2 \in \text{PAP}(o, \mu, \omega) \).

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

\[
\begin{align*}
g^* &= \text{df} \; g \uplus \{ \omega \; \text{co}\langle\rangle, o \mid o \; \text{co}\langle\rangle, \omega \in g \} \\
\text{PAP}_g(o, \mu, q) &= \text{df} \; \{ \pi \mid g^* \vdash \pi \in \text{PAP}(o, \mu, q) \} \\
\text{Osh}_g(o, \omega) &= \text{df} \; \text{PAP}_g(o, \text{rep}\langle\ldots\\>\rangle, \omega) \cup \text{PAP}_g(o, \text{free}\langle\ldots\\>\rangle, \omega) \\
\text{Sanc}_g(o) &= \text{df} \; \omega \; \text{su. th.} \; \text{PAP}_g(o, \text{rep}\langle\ldots\\>\rangle, \omega) \neq \emptyset \\
\text{StRep}_{s, om}(o) &= \text{df} \; \{ o \} \cup \bigcup_{\omega \; \text{su. th.} \; \text{PAP}_{\text{from}}(o, \text{rep}\langle\ldots\\>\rangle, \omega) \neq \emptyset} \text{StRep}_{s, om}(o)
\end{align*}
\]

Other definitions do not change at all, like composite state \( \text{CState}_{s, om}(o) \), field locations \( \text{flds}(o) \) and field subgraph \( \text{fg}_{om}(s) \).

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

\(^1\)Note that in valid modes, which paragraph 6 will define, base-modes \texttt{co} and \( \alpha \in A \) never come with any correlations.
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 \( \mathcal{M} = \{\text{free}, \text{rep}, \text{co}, \text{read}\} \), 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, \( \mu \)-paths were the extensions \( o, q, \ldots, w \) of a \( \mu \)-edge by an optional sequence of co-edges. In full JaM, potential access paths are the (potentially trivial) further extensions \( o, q, \ldots, w \) of a base-JaM path by optional sequences of association paths \( \pi_i \in \text{PAP}(q_i, \alpha_i, q_{i+1}) \) ending in \( q_{n+1} = w \). The mode of this path depends solely on the initial edge’s mode \( \mu \) 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 \( \pi \in \text{PAP}(o, \mu, w) \) has shape \( "o, q, \ldots, w" \) if \( \pi = o, q, \ldots, w \) if \( \pi \) starts with the edge \( o, q \) followed by an optional sequence \( q, \ldots, q_{i-1}, q_i \) of co-edges, and then a sequence of association paths \( \pi_i \in \text{PAP}(q_i, \alpha_i, q_{i+1}) \) with some end-point \( q_{n+1} = w \). Two paths \( \pi_1 \) and \( \pi_2 \) 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 \( m \in \{\text{free}, \text{rep}, \text{co}, \text{read}\} \cup \mathcal{A} \) clearly is still the main classification of potential access paths. The annotated correlations \( \delta \) 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 \( \pi \) as a \( \mu \)-path is a classification according to the base-modes \( m_{\alpha_1 \ldots \alpha_n} \) of \( \pi \)’s extensions by \( \alpha_1 \ldots \alpha_n \)-sequences of association paths to potential access paths. Therefore, full mode \( \mu \) 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 \( \mu(\alpha_1 \ldots \alpha_n) \). In particular, extensions of edges \( o \xleftarrow{\omega} q \) by association path-sequences \( q \xrightarrow{\alpha_1 \ldots \alpha_n} \omega \), i.e., paths of shape \( o \xleftarrow{\omega} q \xrightarrow{\alpha_1 \ldots \alpha_n} \omega \), are paths with base-mode \( \mu(\alpha_1 \ldots \alpha_n) \).

**Definition 8** The use of a mode-term \( \mu \) as a mapping \( \mu : A^* \rightarrow B_\perp \) is defined as follows:

\[
\mu(\varepsilon) =_\text{at} m \quad \text{if} \quad \mu = m<\ldots>
\]

\[
\mu(\alpha.a) =_\text{at} \begin{cases} 
\mu'(\alpha') & \text{if} \quad \mu = m<\ldots, \alpha=\alpha', \ldots> \\
\perp & \text{otherwise}
\end{cases}
\]

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

\[
\mu_{\text{entryset}}(\varepsilon) = \text{rep}
\]

\[
\mu_{\text{entryset}}(\text{elem}) = \text{rep}
\]

\[
\mu_{\text{entryset}}(\text{elem.fst}) = \text{key}
\]

\[
\mu_{\text{entryset}}(\text{elem.snd}) = \text{value}
\]

**Remark 4.** Obviously the order of the correlations in the mode-term \( \mu \) and possible repeated occurrence of the same correlation \( \alpha=\alpha' \) 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 \( \mu : A^* \rightarrow B_\perp \). 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:

\[
\mu \equiv \mu' \iff \forall \alpha \in A^* \cdot \mu(\alpha) = \mu'(\alpha)
\]

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 \( \mu \).

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 \( g \) 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 \( g \) 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 \( g \) and stack \( \vec{r} \) if *mutators* were invoked on receivers \( r_i \) only through a sequence of calls along the edges \( h_j, \ldots, h_i \) of an ownership path to \( r_i \).

\(^2\)Actually, this can only be a mapping if \( \mu \) 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).
The Mutator Control property MC holds in graph $g$ 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: $m \in \{\text{co}\} \cup A \Rightarrow n = 0$.
3. There are no correlations to a co or free mode: $\mu_i(e) \notin \{\text{co}, \text{free}\}$.
4. The correlations’ modes must be valid modes: $\vdash \mu_i$ ok.

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

**Condition 1** generally simplifies the formal treatment through uniqueness: Valid modes $\mu$ specify a unique mode for the extension $\pi \cdot \pi'$ of $\mu$-paths $\pi$ by association paths $\pi'$ (cf. paragraph 1). Valid modes can rightfully be considered mappings to unique base-modes $\mu(\alpha)$. 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
would require some invariant about the consistency between these correlations and the correlations of paths $\pi' \in \text{PAP}(q', \mu', o)$ extensible to $\pi' \cdot \pi$. 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 $g^*$, with inverted co-edges. This ensured that each new path-extending co-path $\pi$ with the new, received parameter handle had a precursor $\pi'$ 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 $o <\text{elem}=(\text{co}<\text{co}>)$, s <\text{elem}>, e, which $o$ derives from set object $s$'s association paths to the set elements. This co-path would mean that $o$ and $e$ should have the same owners. But if $d$ owns the element $e$ through the ownership path $d <\text{rep}<...>$, $e$, it is not guaranteed that there is also an ownership path from $d$ to $o$, since the co-path $o <\text{elem}=(\text{co}<\text{co}>)$, s <\text{elem}>, e has no inverse for extending $d <\text{rep}<...>$, $e$ 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 $s \xrightarrow{\mu} r$, it is only relevant from what kind of method the call is made, and what the base-mode of $\mu$ are (correlations in $\mu$ 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 $\mu(a)$ of the correlation for association role $a$ can become relevant in an effects system extension of JaM that uses effects region this.$a$ for handing mutator calls through $a$-paths (§7.3.2).
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: free<...> ∈ Wr(κ) for κ = mut and obs. Through rep and co call-links, the source may call mutators on the target from within mutators: rep<...>, co<> ∈ Wr(mut). Through read call-links, mutators may never be called: read<...> ∉ Wr(κ) for κ = mut and obs.

Handles of the new base-mode α ∈ 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: α<...> ∉ Wr(κ) for κ = mut and obs.

8. MODE COMPATIBILITY. If two base-modes m and m' were treated as compatible in base-JaM (m ≤_m m'), it should be possible to extend this to full modes that have the same correlations (μ = m<δ> ≤_m m'<δ> = μ'). 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 free<> ω₁ to co-handles o co<> ω₁ and association handles o a<> ω₁ is problematic: It means not only the potential extension of o-targeting paths π = q ↛ o to ω₁ and its co-object, as in base-JaM. In JaM, these paths may be furthermore extended, depending on μ, by association path sequences ω₁ ↛ ω₂ ↛ ω₃ ... to more distant objects ωₖ 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 ω₁ ↛ ... ↛ ωₙ₊₁ of association paths starting from targets ω₁ 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 a<> (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 n1<br> and data-handles n1<br> from set representative s through s<br> n1<br> data<> n2 and s<br> n1<br> data<> e1 if s calls n1’s methods SetNext and SetData and supplies handles s<br> rep data<> n2 or s<br> elem<> e2, respectively.

Second, depth-compatibility is problematic since a change from mode \(\mu_r = m<\alpha=\mu>\) to \(\mu'_r = m<\alpha=\mu'>\) on a handle \(h\) means not only in the higher-level view a harmless mode change of \(h\)’s extensions by \(\alpha\)-paths from \(\mu\)-paths to \(\mu' \geq_m \mu\)-paths, but also means in the type system a change in the handle’s signature from the target’s \(\alpha\)-moded result and parameter types from \(\mu_r\alpha<> = \mu\) to \(\mu'_r\alpha<> = \mu' \geq_m \mu\). 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 \(\tau\) 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 \(\tau' \rightarrow \sigma\) (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 \(\alpha=\mu\) on \(h\) can not cause any problems if through the converted handle no operations with \(\alpha\)-parameters can be invoked. Hence we can obtain a minimum of depth-conversion by treating handles of mode read<\alpha=\mu> as compatible to handles of mode read<\alpha=\mu'> if \(\mu \leq_m \mu'\) if we disallow the supply of \(\alpha\)-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 \(\mu\)-handles as \(\alpha\)-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 \(\mu = m<\delta>\) to \(\mu' = m<>\) on a handle \(h\) implies in the handle’s signature a mode change of the target’s co-moded result and parameter types from \(\mu\co<> = \mu\) to \(\mu'\co<> = \mu' \geq_m \mu\). 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.
To sum up, besides being compatible to themselves,

- all modes are compatible to read-modes with the same correlations,
- all free modes are compatible to 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 is replaced by a mode \( \mu' \) to which \( \mu \) is compatible.

\[ \mu \circ \text{read} =_{\text{at}} \mu \circ \text{read} \]
\[ \mu \circ \text{free} =_{\text{at}} \mu \circ \text{free} \]
\[ \mu \circ \text{rep} = \text{at} \mu \circ \text{rep} \]
\[ \mu \circ \text{co} \]
\[ \mu \circ \alpha \]
\[ =_{\text{at}} \mu' \] if \( \mu = \mu_1 < \ldots, \alpha = \mu', \ldots > \)

\[ \vdash \text{FLdsMths}(c) = (\Gamma, F) \quad F(f) = \mu \circ f \circ \mu_1 \circ \mu_2 \{ \ldots \} \]
\[ \forall \alpha, (\mu_1 \circ \mu_2)(\alpha) = \text{read} \Rightarrow \mu_1(\alpha) = \text{read} \wedge (\mu_1(\alpha) \in \{\text{co}\} \cup \mathbb{A} \Rightarrow \mu_1(\alpha) \notin \{\text{read}\} \cup \mathbb{A}) \]
\[ \forall \alpha, \mu(\alpha) = \text{free} \wedge \mu(\alpha, \alpha) \in \mathbb{A} \wedge \mu \circ \mu(\alpha, \alpha) \neq \text{read} \Rightarrow \mu_1(\alpha) \neq \text{read} \]
\[ \vdash (f : \mu \circ \mu_1 \circ \mu_2 \circ \mu_3 \circ \mu_4) \in \Sigma(\mu_5, c) \]

Figure 6.9: Mode-specific definitions for JaM (part 3)

\[ \mu \circ \text{read} =_{\text{at}} \mu \circ \text{read} \]
\[ \mu \circ \text{free} =_{\text{at}} \mu \circ \text{free} \]
\[ \mu \circ \text{rep} = \text{at} \mu \circ \text{rep} \]
\[ \mu \circ \text{co} \]
\[ \mu \circ \alpha \]
\[ =_{\text{at}} \mu' \] if \( \mu = \mu_1 < \ldots, \alpha = \mu', \ldots > \)

\[ \vdash \text{FLdsMths}(c) = (\Gamma, F) \quad F(f) = \mu \circ f \circ \mu_1 \circ \mu_2 \{ \ldots \} \]
\[ \forall \alpha, (\mu_1 \circ \mu_2)(\alpha) = \text{read} \Rightarrow \mu_1(\alpha) = \text{read} \wedge (\mu_1(\alpha) \in \{\text{co}\} \cup \mathbb{A} \Rightarrow \mu_1(\alpha) \notin \{\text{read}\} \cup \mathbb{A}) \]
\[ \forall \alpha, \mu(\alpha) = \text{free} \wedge \mu(\alpha, \alpha) \in \mathbb{A} \wedge \mu \circ \mu(\alpha, \alpha) \neq \text{read} \Rightarrow \mu_1(\alpha) \neq \text{read} \]
\[ \vdash (f : \mu \circ \mu_1 \circ \mu_2 \circ \mu_3 \circ \mu_4) \in \Sigma(\mu_5, c) \]

To sum up, besides being compatible to themselves,

- all modes are compatible to read-modes with the same correlations,
- all free modes are compatible to 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 is replaced by a mode \( \mu' \) to which \( \mu \) is compatible.

\textbf{Definition 11} The mode compatibility relation \((\leq_m) \subseteq \mathcal{M} \times \mathcal{M}\) in JaM is the transitive reflexive closure of the relation \(\leq_{m}^{1}\) defined in figure 6.8.

\[ \vdash (f : \mu \circ \mu_1 \circ \mu_2 \circ \mu_3 \circ \mu_4) \in \Sigma(\mu_5, c) \]
returns" a potential access path \( \pi' \) of mode \( \mu_i \), then the mode of the corresponding imported path in the sender should be \( \mu_{\pi} \circ \mu_i \). Since this mode is derived from the correlation in the mode \( \mu_{\pi} \circ \mu \) of the returned handle \( s \overset{\mu_{\pi} \circ \omega}{\Rightarrow} \), this means that the \( \alpha_i \)-correlation in the handle's mode must have the mode \( \mu_{\pi} \circ \mu_i \). That is, the import operator \( \circ \) should be associative: \( \mu_{\pi} \circ (\mu_{\pi} \circ \mu_i) = (\mu_{\pi} \circ \mu_i) \circ \alpha_i \). In other words, mode import \( \circ \) is defined by recursively importing the modes of correlations in the imported mode \( \mu \).

10. SIGNATURE OF HANDLES. As in base-JaM, the definition of handles' signatures \( \Sigma(\mu_{\pi}\ c) \) is based on the adaption of FlsMsths(c), which implicitly contains the signature \( \Sigma(c) \) of c-objects, by the use of mode import "o" (see figure 6.9). As an example, figure 6.10 shows how the signature \( \Sigma(PSetImp) \) of PSetImp objects is adapted to the signature \( \Sigma(\mu_{\pi}, PSetImp) \) of MapImp objects' entryset handles with mode \( \mu_{\pi} = rep<elem=rep<fst=key<>>, snd=value<>\rangle \).

More detailed considerations are necessary for the conditions under which a method \( f \) of c-objects can be used through handles of type \( \mu_{\pi} \ c \) and with signature \( \Sigma(\mu_{\pi}\ c) \):

Regarding the new case of association moded parameters in JaM, observe that a parameter of mode \( \alpha<> \) means that the receiver \( r \) 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 read parameter). The receiver expects a parameter object owned by that object \( o \), should it exist, which is "destined" to own all its \( \alpha \)-objects since there is a path of edges from \( o \) to \( r \) that is extended by any \( \alpha \)-path to an ownership path.

1. \( \mu_{\pi} \circ \mu_i = \text{read}<...> \Rightarrow \mu_i = \text{read}<...> \). All cases where a non-read parameter in \( \Sigma(c) (\mu_i \neq \text{read}<...>) \) becomes a read parameter in \( \Sigma(\mu_{\pi}\ c) (\mu_{\pi} \circ \mu_i = \text{read}<...>) \)
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 $\mu_r \circ \text{rep} = \text{read}$ or $\text{read} \circ \text{co} = \text{read}$ point to objects with the owner desired by the receiver. In JaM, the same problem additionally arises with the supply of $\alpha$-parameters through call-links correlating $\alpha$ to a read-mode, i.e., where $\mu_r \circ \alpha <> = \text{read} <>$.

2. $\mu_t = \text{co} <> \Rightarrow \mu_r \neq \alpha <>$. Like the targets of two read-handles, the targets of two $\alpha$-handles do not necessarily have the same owner: There is no guarantee that there is any object “destined” to own all $\alpha$-objects of the caller. But without this one, each of the $\alpha$-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 $\alpha$-targets by the supply of an $\alpha$-handle as co-parameter through an $\alpha$-call-link.

3. $\mu_t = \alpha <> \Rightarrow \mu_r \neq \text{read} < >$. 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. $\mu = \text{free} < \alpha = \beta < >, \ldots > \land \mu_t \circ \mu(\alpha) \neq \text{read} \Rightarrow \mu_r \neq \text{read} < >$. 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 $\alpha = \mu$ in a handle’s mode specify the mode $\mu$ by which the source can access the target’s $\alpha$-objects. The recursive, associative definition of $\circ$ ensured that the mode $(\mu_t \circ \mu) \circ \alpha <>$ by which the sender $s$ can access returned object $o$’s $\alpha$-results through returned handle $s \rightarrow^{\mu_r \circ \mu} o$ is the same as the mode $\mu_r \circ (\mu \circ \alpha <>)$ of a (theoretical) indirect access via the receiver $r$ through call-link $s \rightarrow r$ and its result handle $r \rightarrow o$ (cf. paragraph 9). But also the access to the returned object’s $\alpha$-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 $\alpha$-parameters if the returned handle’s mode $\mu_r \circ \mu$ is a mode $m < \ldots, \alpha = \mu', \ldots >$ where $\mu'$ 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 $\mu$ is a co- or association mode then $s \rightarrow r \rightarrow o$ was for $s$ the wanted potential access path of mode $\mu_r \circ \mu$ to $o$ and its $\alpha$-parameters. Otherwise, $\mu$ was a mode free $< \ldots, \alpha = \mu'', \ldots >$ with $\mu_r \circ \mu'' = \mu' \neq \text{read} < >$, so that $\mu''$ can neither be read nor rep. In the case that $\mu''$ is a free mode, the sender was theoretically able to pass its (also free) $\mu_r \circ \mu''$ handles to the receiver as free $\mu''$-parameters, which could pass them to $o$ as $\alpha$-parameters. But in the case that $\mu''$ is an association mode $\beta <>$, the sender could pass its $\mu_r \circ \mu''$ handles as the receiver’s $\mu''$-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 $\hat{\mu}$ (with correlations $\gamma = \hat{\mu}'$) as conditions over mappings $\hat{\mu}(\hat{\gamma})$ (and $\hat{\mu}(\hat{\gamma}, \gamma)$).

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: $g_0 \models UH, UO$, and in each possible reduction step from object graph $g$ to $g'$, $g \models UH, UO$ implies $g' \models UH, UO$. 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 $g'$, a much stronger property will be needed in $g$, 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 $\mu$ are not just a $\mu$-edge followed by co-edges any more: In base-JaM, whenever a new co-edge $o \xrightarrow{co} o'$ appeared in the extended object graph $g^*$ (by whatever operation), the only new ownership paths this entailed were from $o$’s old owner to $o'$ and all its co-objects $\omega$. In full JaM, a new co-edge $o \xrightarrow{co*} o'$ may entail new ownership paths $\pi = \pi_1 \cdot o \xrightarrow{co*} o' \cdot \pi_2 \in PAP(u, \mu, \omega)$ to all objects $\omega$ that can be
reached from \( o' \) via association path sequences \( o' \xrightarrow{\alpha} o \) from \( o' \) and its co-objects \( o'' \) (i.e., \( \pi_2 = o' \xrightarrow{\text{co-oo}^*} o'' \xrightarrow{\alpha} \omega \)). This is possible whenever any potential direct extensions of the first segment \( \pi_1 \) to \( o \) by \( \alpha \)-association path sequences were already potential access paths of mode \( \mu \).

Stating this in a more formal way is not as straightforward as it might seem: First, we are talking not about actual extensions but about potential extensions—the new ownership path \( o' \xrightarrow{\alpha} o' \) actually exists in the object graph. Second, the first segment \( \pi_1 \), whose extension we are considering, is not necessarily a potential access path. For example, in the map example, we have the path \( \pi_1 = d \xrightarrow{\text{rep-}e \text{m}} e \xrightarrow{\text{data}} n \) 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 \( n \xrightarrow{\text{co-\_path}} n' \) to another node with a data-handle \( \pi_2 = n' \xrightarrow{\text{data-oo}} e' \), then \( \pi_1 \cdot n \xrightarrow{\text{co-oo}} n' \cdot \pi_2 \) 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 \( \alpha \)-association path sequence by adding dummy association handles \( o \xrightarrow{\lambda} o' \) to new dummy objects \( o.a \) to the object graph \( g \). In such an extended graph \( g^\circ \), the potential extensions become one actual extension, so that we can then simply say: There are new ownership paths \( \pi = \pi_1 \cdot o \xrightarrow{\text{co-oo}^*} o' \xrightarrow{\text{co-oo}^*} o'' \xrightarrow{\alpha} \omega \in \text{PAP}_g(u, \mu, \omega) \) iff \( \pi' = \pi_1 \cdot o \xrightarrow{\alpha} o.a \) is a potential access path \( \pi' = \text{PAP}_g(u, \mu, o.a) \).

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 \( \pi = \pi_1 \cdot o \xrightarrow{\text{beta-oo}^*} o' \xrightarrow{\text{co-oo}^*} o'' \xrightarrow{\alpha} \omega \in \text{PAP}_g(u, \mu, \omega) \) iff \( \pi' = \pi_1 \cdot o \xrightarrow{\beta} o.a \) is a potential access path \( \pi' = \text{PAP}_g(u, \mu, o.a) \).

**Definition 12** The notion of "objects reachable from \( o \) via possible \( \alpha \)-sequence of association paths" shall be called \( o' \)'s \( \alpha \)-**region**. For each object \( o \in O \) and possible association paths with the sequence \( \alpha \in A^* \) of association roles, i.e., for each region, let the corresponding dummy object be \( o.a \in O_r \) = \( O \times A^* = O \cup O \times A \cup O \times A \cup \ldots \), and call it the **region object**. Since \( O \times A^* = (O \times (A^* \times A^*)) = (O \times A^*) \times A^* \), formally also region objects \( o = q.q \) have their region objects \( o.a \) but these are simply the region objects of \( q \)'s \( q \)-**region** of \( q \)-region objects: \( o.a = (q.q).a = q.(q \cdot a) \). The extension \( g^\circ \) of graph \( g \in \text{Graph} \) by dummy edges is \( g^\circ = g \cup \{ o.a.o.a \mid o \in O_r, \alpha \in A \} \). Extended graphs \( g^\circ \) are multisets of mode-labeled edges between region object: \( g^\circ \in \text{Graph}_r = O \times A^* \times A^* \times A^* \times A^* \times A^* \times A^* \times A^* \).

We will write simply \( q.q.a \) for indirect region objects \( q.a \). Region objects include the special case of \( o.\epsilon = o \in O \subseteq O_r \). The dummy objects are the proper region objects \( o \in O \times A^+ = O_r \), i.e., region objects \( o = o.\alpha \) with \( \alpha \neq \epsilon \). 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 \( o, \omega, q, \epsilon \) etc. for objects normally range over all of \( O_r \).

Since the edges added in \( g^\circ \) lead only to region objects and do not connect proper objects, there is a potential access path between proper objects in \( g^\circ \) if and only if
there is a potential access path in \( g \).\(^4\) The move from \( g \) to \( g^\circ \) in reasoning will not affect ownership and representations between proper objects.

2. \( \{\text{NEW}\} \). The creation of a new object \( o \) by the evaluation of object creation expression \( \text{new}\langle \cdot \rangle \ e() \) in the execution of a method called on (creator) object \( r \) adds an edge \( r \xrightarrow{} \text{fresh} \cdot \ a \) to a fresh object \( o \). "Fresh" implies that \( o \) is not the currently active object \( r \), and neither target nor source of any edge in \( g \).

**Lemma 6** Consider the addition of a free edge \( h_o = r \xrightarrow{} \text{free} \cdot \ a \) to fresh object \( o \) such that \( g' = g \oplus r \xrightarrow{} \text{free} \cdot \ a \). In \( g^\circ \), there are two kinds of new potential access paths \( \pi \in PAP_{g^\circ}(o, \mu, \omega) \):

1. **Initially new paths** go from \( o = r \) to \( o = o.\alpha. \). They are extensions \( \pi_{\alpha} = r \xrightarrow{} o \cdot \alpha. \) \( o.\alpha \) of \( h_o \) by dummy edges for every \( \alpha \in \mathbb{A}^* \) with \( \mu_o(\alpha) \neq \perp \). If it is an association path, i.e., \( \mu_o(\alpha) = \beta \), and a possibly empty sequence \( \pi_3 = o.\alpha. \xrightarrow{} o.\alpha.\alpha' \) of further dummy edges, \( \pi \) has an unchanged witness \( \sigma(\pi) = \pi_1 \cdot \sigma(\pi_{\alpha}) \cdot r.\beta. \xrightarrow{} o.\beta.\alpha' \) of the same shape.

2. **Internally new paths** are concatenations \( \pi_1 \cdot \pi_{\alpha.\alpha'} = \pi_1 \cdot \pi_{\alpha} \cdot \pi_3 \) of a non-trivial unchanged path \( \pi_1 \) of edges from \( o \) to \( r \), an initially new path \( \pi_{\alpha} \) of some association mode \( \beta<\cdot \), i.e., \( \mu_o(\alpha) = \beta \), and a possibly empty sequence \( \pi_3 = o.\alpha. \xrightarrow{} o.\alpha.\alpha' \) of further dummy edges. \( \pi \) has an unchanged witness \( \sigma(\pi) = \pi_1 \cdot \sigma(\pi_{\alpha}) \cdot r.\beta. \xrightarrow{} o.\beta.\alpha' \) of the same shape.

Since all new potential access paths in \( g^\circ \) target the fresh object \( o \) and its region objects, and since these cannot be the target of ownership paths in \( g^\circ \), an UO- or UH-conflict could exist in \( g^\circ \) only among two new ownership paths: There can be none among two initially new ownership paths \( \pi_{\alpha} \) and \( \pi_{\alpha'} \) with the same target \( o.\alpha = o.\alpha' \), since this implies that they are identical. There can be none among two internally new ownership paths \( \pi_1 \cdot \pi_{\alpha.\alpha'} \) and \( \pi_1 \cdot \pi_{\alpha.\alpha'} \) with the same target \( o.\alpha.\alpha' \), since then their witnesses have the same, uniquely determined target \( \sigma(o.\alpha.\alpha') = r.\mu_o(\alpha).\alpha' \), and thus would have been in conflict in \( g^\circ \). And there can be none between initially new ownership paths \( \pi_{\alpha} \) and internally new ownership paths \( \pi_1 \cdot \pi_{\alpha.\alpha'} \) since target \( o.\alpha \) cannot be the same as target \( o.\alpha''.\alpha' \). The initially new path implies \( \sigma(\alpha) \in \{\text{free, rep}\} \) while the internally new path implies \( \mu_o(\alpha'') \in \mathbb{A} \), so that the nesting constraints on association modes prevent \( \mu_o(\alpha'' \alpha') \in \{\text{free, rep}\} \).

**Proof of the lemma:** The only added edge \( h_o \) in \( g^\circ \) targets a fresh object \( o \). This means that in \( g^\circ \), \( o \) was only connected with other objects by dummy edges \( o \xrightarrow{} o.\alpha \) to its own region objects. In \( g^\circ \), \( o \) still has no other edges, since it is not \( r \). And the only edge targeting \( o \) in \( g^\circ \) is \( h_o \). Hence the only way to extend new handle \( h_o \) to a new potential access path is along dummy edges to \( o \)'s region objects. Since dummy edges are not co-edges, this extension depends on the correlations \( \mu_o \) in \( h_o \)'s mode: \( \text{Iff } \mu_o(\alpha) \neq \perp \), then \( \pi_{\alpha} = h_o \cdot o \cdot \alpha. \) \( o.\alpha \) is a potential access path of corresponding

\(^4\)To be precise, this presupposes that the given set \( \mathcal{O} \) of object-identifiers contains no dummy objects: \( \mathcal{O} \cap \mathcal{O} \times \mathbb{A}^* = \emptyset \).

139
mode, an initially new path. In particular, π_α's mode is β<> iff μ_0(α) = β. This includes handle h_0 itself as an initially new path with α = ε.

Nesting-constraints on modes prevent extensions of free h_0 to initially new paths from being co. An initially new path π_α that is an association path of mode β<> can extend any path π_1 to r with a correlation β=μ to an internally new path π = π_1 • π_α = π_1 • π_α,ε of mode μ. Its witness σ(π) is π_1 • r β<> r.β. If an internally new path π = π_1 • π_α,ε has a mode with a correlation β=μ, it is extended further to potential access path π • o.α.α' β<> o.α.α'/β = π_1 • π_α,ε. Since it has a witness σ(π) • r.μ_0(α).α' β<> r.μ_0(α).α'/β = π_1 • r.μ_0(α).α'/β, 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 π' • π = (π' • π_1) • π_α. Since it has a witness π' • σ(π) = (π' • π_1) • r.μ_0(α).r.μ_0(α), it is an internally new path again. Other combinations are not possible, since all initially new and internally new paths target o and region objects of it, and thus cannot target the source 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 (c, µ_0, o) adds a new edge c β<> o iff it involves a real conversion of the assigned handle from mode µ to another mode µ ≠ µ_0. We can ignore the destruction g_0 = g ⊗ c β<> o' of (the edge corresponding to) the variable's old value and focus on replacing in the resulting graph g_0 the right-hand handle c β<> o by the variable's new value c β<> o. Note that the typeability of the redex e following from Theorem 6 and |= σ following from Theorem 6 guarantees µ' ≤_m µ. Hence the step from g_0 to g' can be handled by induction on the number k of elementary conversions from µ_0 to µ, i.e., µ_0 ≤_m µ_1 ≤_m µ_2 ... ≤_m µ_k = µ, with corresponding intermediate graphs g_i+1 = g_i ⊗ c β<> o ⊗ c β<> o, o up to g_k = g'.

Lemma 7 Consider elementary mode conversion g_i+1 = g_i ⊗ c β<> o ⊗ c β<> o, i.e., the substitution of an edge h_i+1 = c β<> o for an edge h_i = c β<> o with µ_i ≤_m µ_i+1. All potential access paths π ∈ PAP_{g_i+1} (c, µ, ω) in g_i+1 have a precursor π' = π[h_i/h_i+1] ∈ PAP_{g_i} (c, µ', ω) in g_i with the old edge instead of the new one. Two kinds of new potential access paths can be distinguished:

1. Initially new paths π ∈ PAP_{g_i+1} (c, µ, ω) start with the converted edge c β<> o and have a shape c β<> o, o • • • • • • • •. Their precursor π[h_i/h_i+1] has the same mode µ or a directly compatible mode µ' ≤_m µ, and has, save for the initial edge, the same shape c β<> o, o • • • • • • • •.

2. Internally new paths π ∈ PAP_{g_i+1} (c, µ, ω) have a precursor π[h_i/h_i+1] ∈ PAP_{g_i} (c, µ, ω) of the same mode and shape.
Note that the constraints on mode compatibility and on the nesting of modes ensure that \( \bar{\mu}_{i+1} \geq_m \bar{\mu}_i \) cannot be free. Hence, first, the nesting constraints on modes exclude that the extensions \( \pi = c \cdot \bar{\mu}_{i+1} \cdot \alpha \cdot \ldots \) of the converted edge, i.e., the initially new paths, have a free mode. Second, the only (initial) edge whose multiplicity is increased in \( g_{i+1}^\oplus \) 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 \( \pi \in PAP_{g_{i+1}^\oplus}^\circ (c, \mu, \omega) \) with directly compatible precursor \( \pi' \in PAP_{g_i}^\circ (c, \mu', \omega) \) with \( \mu' \leq_m \mu \) can be an ownership path only in case of \( \mu' = \text{free} <\delta> \leq_m \text{rep}<\delta> = \mu \). But then the precursor is already an ownership path, so that \( \pi \) does not change ownerships and does not affect \( UO \). And since the precursor is free, it guarantees that all old ownership paths \( \pi' \) to \( \omega \) had the initial edge \( c \cdot \bar{\mu}_{i+1} \cdot \alpha \), whose multiplicity was one. Since this multiplicity is reduced by one in \( g_{i+1}^\oplus \), no ownership path with unchanged initial edge (unchanged or internally new path) can have the target \( \omega \). \( \text{rep}<\delta> = \mu \). Since all initially new ownership paths \( \pi' \) had an ownership path \( \pi \) as precursor (of the same or compatible mode), \( \pi \)'s shape \( c \cdot \bar{\mu}_{i+1} \cdot \alpha \cdot \ldots \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \·
4. \{RET\}. The return of a handle \(\langle r, \mu_o, o \rangle\) at the end of the execution of a method called through call-link \(s \stackrel{\mu_r}{\leftarrow} r\) adds the sender's edge \(s \stackrel{\mu_r}{\leftarrow} r\) to the object graph (imported edge) and subtracts the receiver's edge \(r \stackrel{\mu_o}{\leftarrow} o\) (exported edge) and the call-link: \(g'' = g \odot r \stackrel{\mu_o}{\leftarrow} o \odot s \stackrel{\mu_r}{\leftarrow} r \odot s \stackrel{\mu_r}{\leftarrow} o\). We can ignore the second substep, the removal \(g' = g'' \ominus s(\text{im}(\eta'))\) 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 \(g''\) are \(s \stackrel{\mu_r}{\leftarrow} r\) and, if \(\mu_r \circ \mu_o = \text{co}<\), its inverse \(o \stackrel{\text{co}}>\). There are two cases how \(\mu_r \circ \mu_o\) can be free: If \(\mu_o\) is a free mode then assumption \(g \models \text{UH}\) ensures for the exported handle \(r \stackrel{\text{co}}>\). \(o\) that in \(g\) there can already be an edge \(s \stackrel{\text{co}>}{\leftarrow} o\) 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 \(\mu_o = \text{co}<\) then \(\mu_r\) is free, so that \(s \stackrel{\mu_r}{\leftarrow} r \stackrel{\text{co}>}{\leftarrow} o\) was a free path in \(g\). Hence \(g \models \text{UH}\) ensures that there can be \(s \stackrel{\text{co}>}{\leftarrow} r\) only if it equals \(s \stackrel{\mu_r}{\leftarrow} r\) and that the multiplicity of \(s \stackrel{\mu_r}{\leftarrow} r\) is one. But since the call-link's multiplicity is decreased while that of \(s \stackrel{\text{co}>}{\leftarrow} r\) is increased, the multiplicity remains one. The case with \(\mu_o = \alpha<\) and a correlation \(\alpha=\text{free}<...>\) is excluded by the nesting constraint on modes.

**Lemma 8** Consider result return \(g'' = g \odot r \stackrel{\mu_o}{\leftarrow} o \odot s \stackrel{\mu_r}{\leftarrow} r \odot s \stackrel{\mu_r}{\leftarrow} o\), i.e., the substitution of the imported edge \(h\) for the exported edge \(h\) and the call-link \(hr = s \odot r\). In \(g''\), there are three kinds of new potential access paths \(\pi \in PAP_{g''}(o, \mu, \omega)\):

1. **Co-closure paths** \(\pi \in PAP_{g''}(o, \text{co}<\), \(\omega)\) are co-paths that exist if \(\mu_r \circ \mu_o = \text{co}<\). They have a precursor \(\pi' \in PAP_{g}(o, \mu, \omega)\) with the imported handle \(s \stackrel{\text{co}>}{\leftarrow} o\) replaced by the call-link/exported handle combination \(s \circ \text{co}>\). \(r \circ \text{co}>\). \(o\), and its inverse \(s \circ \text{co}>\) replaced by the combination \(s \circ \text{co}>\). \(r \circ \text{co}>\). \(o\) of inverse exported handle and inverse call-link: \(\pi' = \pi[h_r \odot h_o/h'_o, h_o'^{-1} \odot h_r'^{-1}/h_o'^{-1}]\).

2. **Internally new paths** \(\pi \in PAP_{g''}(o, \mu, \omega)\) have a precursor \(\pi' \in PAP_{g}(o, \mu, \omega)\) with the same shape and \(\pi' = \pi[h_r \odot h_o/h'_o, h_o'^{-1} \odot h_r'^{-1}/h_o'^{-1}]\) in case of \(\mu_r \circ \mu_o = \text{co}<\), and \(\pi' = \pi[h_r \odot h_o/h'_o]\) otherwise.

3. **Initially new paths** are non-co paths \(\pi = h_o \odot \pi' \in PAP_{g''}(s, \mu, \omega)\) that exist if \(\mu_r \circ \mu_o \neq \text{co}<\). They start with \(h'_o\) have a shape \(s \circ \text{co}>\). \(r \circ \text{co}>\). \(o\), and a counterpart \(\xi \circ \pi' = h_o \odot \tilde{\pi}' \in PAP_{g}(r, \mu', \omega)\) with a mode \(\mu'\) such that \(\mu = \mu_r \circ \mu'\). The counterpart starts with \(h_o\) instead of \(h'_o\) and has shape \(r \circ \mu_o\), \(o \circ \text{co}>\). \(r \circ \text{co}>\). \(o\). The two paths' postfixes are related like internally new paths: \(\tilde{\pi}' = \tilde{\pi}[h_r \odot h_o/h'_o]\), and similarly related are \(\pi\) and the combination \(h_r \odot \xi \circ \pi'\) of the call-link and the counterpart: \(h_r \odot \xi \circ \pi' = \pi[h_r \odot h_o/h'_o]\). It is a potential access path in \(PAP_{g}(s, \mu, \omega)\) if \(\mu_o(\alpha) \in \{\text{co}\} \cup A\), and its shape is \(s \odot \mu_r \odot r \circ \text{co}>\). \(o\) if \(\mu_o = \text{co}<\) or \(s \odot \mu_r \odot r \circ \text{co}>\). \(o\) if \(\mu_o = \beta<\).
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 \( \pi \) of the kind with a precursor \( \pi' \) and another ownership path \( \bar{\pi} \) with the same target \( \omega \) that is unchanged, internally new or initially new with a precursor \( \bar{\pi}' \). Then \( g^\circ \models \text{UO} \) guarantees that \( \pi' \) and \( \bar{\pi}' \) have the same source, and thus so have \( \pi \) and \( \bar{\pi} \). There is no UO-conflict. And if \( \pi \) or \( \bar{\pi} \) were free, then \( g^\circ \models \text{UH} \) would guarantee that precursor \( \bar{\pi}' \) has precursor \( \pi' \)’s shape \( s \cdot \bar{d} \cdot r \cdot s \cdot e \cdot d \cdot \bullet \) with \( \mu_o(\bar{d}) \notin \{ \text{co} \} \cup A \). It can only be an ownership path if it is free and if it has a free counterpart \( \text{epo}(\pi) \), i.e., \( \mu_o(\bar{d}) = \text{free} \). But then \( g^\circ \models \text{UH} \) guarantees for all ownership paths \( \bar{\pi}' \) in \( g^\circ \) sharing \( \pi' \)’s target that they have \( \text{epo}(\pi) \)’s initial edge, i.e., the exported handle \( h_o \), and that \( h_o \)’s multiplicity is one. Since its multiplicity is reduced by one in \( g^n \), no unchanged or internally new path \( \bar{\pi} \) can have \( \pi' \)’s target. If an initially new ownership path \( \bar{\pi} \) with precursor \( \bar{\pi}' \) has \( \pi' \)’s target, then ownership path \( \bar{\pi}' \)’s initial edge \( s \cdot \bar{d} \cdot r \) must be the same as \( \text{epo}(\pi) \)’s initial edge \( r \cdot s \). But then \( \bar{\pi} \) and \( \pi \) start with the same edge. If an initially new path \( \bar{\pi} \) without precursor has \( \pi' \)’s target, then its counterpart \( \text{epo}(\bar{\pi}) \) is an ownership path. Hence it must have \( \text{epo}(\pi) \)’s initial edge \( r \cdot s \cdot e \cdot d \cdot \bullet \), so that \( \bar{\pi} \) and \( \pi \) start with the same edge. There is no UO- and no UH-conflict with initially new \( \pi \).

**Proof of the lemma:** For uniformity, let \( \sigma \) be the substitution \( [h_r \cdot h_o/h_o', h_o^{-1} \cdot h_r^{-1}/h_o] \) in case of \( \mu_r \circ \mu_o = \text{co}<> \) and \( [h_r \cdot h_o/h_o'] \) otherwise. In the base case of potential access paths \( \pi \) in \( g^n \), \( \pi \) is a single edge. It is new only if it is the imported handle \( \pi_o = s \cdot \bar{d} \cdot r \cdot s \cdot e \cdot d \cdot \bullet \), or its inverse \( \pi_o^{-1} = o \cdot s \cdot e \cdot d \cdot s \cdot \bar{d} \cdot r \cdot o \). It should be \( \mu_r \circ \mu_o = \text{co}<> \). If \( \mu_r \circ \mu_o = \text{co}<> \) then \( \mu_r = \mu_o = \text{co}<> \). Hence in \( g^\circ \), call-link and exported handle combine to the co-path \( s \cdot \bar{d} \cdot r \cdot s \cdot e \cdot d \cdot \bullet \), and they have inverses that combine to the co-path \( o \cdot s \cdot e \cdot d \cdot s \cdot \bar{d} \cdot r \cdot o \). These are the necessary precursors \( \sigma(\pi_o) \) and \( \sigma(\pi_o^{-1}) \) that make \( \pi_o \) and \( \pi_o^{-1} \) co-closure paths. If \( \mu_r \circ \mu_o = \text{co}<> \), \( \pi_o \) is an initially new path \( \pi_o \in PAP_{g^\circ}(s, \mu_r \circ \mu_o, o) \) and with exported handle \( \bar{d} \cdot r \cdot s \cdot e \cdot d \cdot \bullet \cdot o \) as counterpart \( \text{epo}(\pi_o) \in PAP_{g^\circ}(r, \mu_o, o) \) such that \( h_r \cdot \text{epo}(\pi_o) = \sigma(\pi) \). If \( \mu_o = \text{co}<> \) then \( \mu_r \circ \mu_o = \mu_r \), and call-link and exported handle combined to \( \pi_o \)’s precursor \( s \cdot \bar{d} \cdot r \cdot s \cdot e \cdot d \cdot \bullet \cdot o \). If \( \mu_o = \alpha<> \), then \( \mu_r \circ \mu_o \) presupposes a correlation \( \alpha = \mu \) in \( \mu_r \). But then call-link and exported handle already combined in \( g^\circ \) to the potential access path \( s \cdot \bar{d} \cdot r \cdot s \cdot e \cdot d \cdot \bullet \cdot o \) of mode \( \mu_r \circ \alpha <> = \mu_r \circ \mu_o = \mu \).

Larger potential access paths \( \pi \) in \( g^n \) are the result of extending paths \( \pi_1 \in PAP_{g^\circ}(o, \mu_1, q) \) by a co- or \( \alpha \)-path \( \pi_2 \in PAP_{g^\circ}(q, \mu_2, \omega) \), which always has a precursor \( \sigma(\pi_2) \).

- If \( \pi_1 \) is unchanged of internally new then its shape-equivalent precursor \( \sigma(\pi_1) \) is extended by precursor \( \sigma(\pi_2) \) to a precursor \( \sigma(\pi) = \sigma(\pi_1) \cdot \sigma(\pi_2) \) of \( \pi = \pi_1 \cdot \pi_2 \) with
the same mode and shape. If \(\pi_1\) and \(\pi_2\) are unchanged, \(\pi\) is unchanged. Otherwise, \(\pi\) is internally new.

- If \(\pi_1\) is a co-closure-path then its mode \(co<>\) has no correlations and thus allows only for extension by a co-path \(\pi_2\). Hence \(\pi\)'s mode is \(\pi_1\)'s mode, and the precursor's combination \(\sigma(\pi_1) \cdot \sigma(\pi_2)\) is possible and a precursor \(\sigma(\pi)\) for \(\pi\). \(\pi\) is again a co-closure path.

- If \(\pi_1\) is initially new then its counterpart \(\text{epo}(\pi_1)\) is extended by \(\sigma(\pi_2)\) to a counterpart \(\text{epo}(\pi) = \text{epo}(\pi_1) \cdot \sigma(\pi_2)\) with \(hr \cdot \text{epo}(\pi) = hr \cdot \text{epo}(\pi_1) \cdot \sigma(\pi_2) = \sigma(\pi) \cdot \sigma(\pi_2) = \sigma(\pi)\). If \(\pi_1\)'s shape is \(s \cdot \mu_{r}o_{hr} \cdot o \cdot co<> \cdot -\sigma -\alpha -\omega\) and \(\text{epo}(\pi_1)\)'s shape is \(r \cdot hr \cdot o \cdot co<> \cdot -\sigma -\alpha -\omega\) then their extensions by \(\pi_2\) and \(\sigma(\pi_2)\) to \(\pi\) and \(\text{epo}(\pi)\) have the following shapes by Lemma 10: \(s \cdot \mu_{r}o_{hr} \cdot o \cdot co<> \cdot -\sigma -\alpha -\omega\) and \(r \cdot hr \cdot o \cdot co<> \cdot -\sigma -\alpha -\omega\) in case of \(\mu_2 = co<>\), and \(s \cdot \mu_{r}o_{hr} \cdot o \cdot co<> \cdot -\sigma -\alpha -\omega\) and \(r \cdot hr \cdot o \cdot co<> \cdot -\sigma -\alpha -\omega\) in case of \(\mu_2 = co<>\). Moreover, if \(\mu_0(\alpha, \beta) = \gamma\), then \(\text{epo}(\pi)\) has mode \(\gamma<>\) and \(\mu_0o_{hr}o_\mu\) implies \(\mu_0\gamma<>\), then \(\text{epo}(\pi)\) has mode \(\gamma<>\). Hence \(\mu_0\gamma<>\) implies \(\mu_0\gamma<>\).

5. \{CALL\}, THE PROBLEM. The invocation of a method through call-link \(s \cdot \Delta \cdot \Gamma \cdot \omega\) adds the receiver's self-edge \(r \cdot \triangle \cdot \omega\), and adds one edge \(r \cdot \Delta \cdot o_j\) in the receiver for each non-nil-handle argument \(s, \mu_j', o_j\) 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 \(\mu_1, \ldots, \mu_k\), then the modes \(\mu_1', \ldots, \mu_k\) of the \(k\) handle arguments are compatible to \(\mu_\Gammao_{\Delta}o_{\mu_1}, \ldots, \mu_\Gammao_{\Delta}o_{\mu_k}\), i.e., \(\mu_j \leq \mu_\Gammao_{\Delta}o_{\mu_j}\). From the case of \{upd\}, we know that the edges \(s \cdot \Delta \cdot o_j\) in \(g\) can be replaced, in an intermediate step, to compatible edges \(s \cdot \Delta \cdot o_j\) without danger for the structure of object ownership. But does the replacement of each converted handle argument, the sent handle \(s \cdot \Delta \cdot o_j\), by the corresponding received handle \(r \cdot \Delta \cdot o_j\) preserve the structure of object ownership?

Normally, for each received handle, one expects first the initially new paths \(\pi_\Delta = r \cdot \Delta \cdot o_j \cdot \hat{e} -\omega\) that extend the handle to objects \(\omega\) reachable via \(o_j\) and that have a counterpart \(\text{sent}(\pi_\Delta) = s \cdot \Delta \cdot o_j \cdot \hat{e} -\omega\) in the graph before the supply. Furthermore, based on initially new co- or association path \(\pi_\Delta\), there should be internally new paths \(\pi = \pi' \cdot \pi_\Delta \cdot q \cdot \hat{e} -\omega\) that target objects reachable before the supply from \(s\) by a path \(s \cdot \Delta \cdot o_j \cdot \hat{e} -\omega\), and that have a witness \(\text{wit}(\pi) = \pi' \cdot s \cdot \hat{e} -\omega\). \((\pi_\Delta\) is a co-path) or \(\text{wit}(\pi) = \pi' \cdot s \cdot \hat{e} -\omega\). \(\pi_\Delta\) (if \(\pi_\Delta\) 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 \( o_j \). 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 \( r \) is reachable from \( o_j \)).

At the lowest level of the object graph, every new potential access path \( \pi \in PAP(o, \mu, \omega) \) from \( o \to w \) of course has a “precursor” \( \pi' = \pi\left[ r \xrightarrow{\mu r} s \xrightarrow{\mu^o_{o_j}} o_j \xrightarrow{\mu_j} r \right. \) in reverse and the sent handle \( s \xrightarrow{\mu r} o_j \). (We ignore the inverse received handle in case of \( \mu_j = co<> \).)

\[
\begin{array}{cccccc}
& o & \xrightarrow{\text{new}} & r & \xrightarrow{\mu_j} & o_j & \xrightarrow{\text{new}} & \omega \\
\end{array}
\]

But \( \pi' \) 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 \( \pi \)’s mode \( \mu \) how \( \pi \)’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 \( r \) and \( o_j \) that can only be bridged by following the call-link \( r \xrightarrow{\mu r} s \) against its referencing direction before continuing in forward direction with the sent handle \( s \xrightarrow{\mu r} o_j \). For reasoning about new ownership paths \( \pi \) based on such a lose kind of connection between \( o \) and \( \omega \) 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 \( \mu \)-ownership on \( \omega \), a reservation which the supply realizes by collapsing the connection to the potential access path \( \pi \). The assumption about reserved ownership has to be formulated strongly enough to ensure its own preservation when \( s \)’s reservation for a \( J-L \)-edge to \( o_j \) 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 \( \omega \), or more generally, for a \( \mu \)-path to \( \omega \)?: When there is a potential access path \( \pi \in PAP(o, \mu, q, \varphi) \) to a region object \( q, \varphi \) and from \( q \to \omega \) there is a corresponding \( \varphi \)-sequence of association paths modulo region-coupling, written, \( \varphi = q \xrightarrow{\mu} \omega \). That is, the pair \( (\pi, \varphi) \) formalizes the \( \mu \)-reservation. The “modulo region-coupling”-qualification is the crucial part that takes the gaps \( u \xrightarrow{\mu} v \xrightarrow{\mu_{o_j}^o} w \) into account that could be closed to \( u \xrightarrow{\mu} v \xrightarrow{\mu_{o_j}^o} w \) by the supply of handle \( v \xrightarrow{\mu_{o_j}^o} w \) to \( u \) through call-link \( u \xrightarrow{\mu} v \).

The region-coupling relation \( = \), 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 \( o \xrightarrow{\varepsilon} o \) entails \( o, \varepsilon \)-\( \varphi \) entails \( o, \varepsilon, \varphi \) = \( o, \varepsilon \).
Through ownership paths $\pi \in PAP(o, \mu, \omega)$ with correlations $\beta = \alpha<\gamma$, o’s $\alpha$-handles $\langle o, \alpha<\gamma, q \rangle$ and $\omega$’s $\beta$-handles $\langle \omega, \beta<\gamma, q \rangle$ can be exchanged in both directions. That is, members $q$ of o’s $\alpha$-region (cf. Definition 12) can become members of $\omega$’s $\beta$-region, and vice versa. Hence we will say that through $\pi$, o’s $\alpha$-region and $\omega$’s $\beta$-regions are coupled, in symbols $o.\alpha = \omega.\beta$.

If objects $q$ enter or leave the $\gamma$-region of an object $\omega$ in o’s $\alpha$-region ($o \dashv \gamma \omega$) then they also, respectively, enter or leave o’s $\alpha.\gamma$-region. Hence we will say that the two regions are coupled: $o.\alpha.\gamma = \omega.\gamma$. To facilitate later proofs, this is generalized from objects $\omega$ currently in o’s $\alpha$-region to objects $\omega$ that might become a member through handle exchange since it is reachable from o via an $\alpha$-sequence of association paths modulo region-coupling ($o \dashv \alpha\gamma \omega$).

For formal reason, if an ownership path $\pi \in PAP(o, \mu, \omega)$ does not correlate an association role $\beta$ ($\mu(\beta) = \perp$), then we say that the target’s $\beta$-region is coupled with the source’s “undefined region,” in symbols $o.\perp = \omega.\beta$.

One “undefined region” is as good as another; they are all mutually coupled: $o.\perp = \omega.\perp$.

It is also defined formally in figure 6.11 what it means to be an $\alpha$-sequence of association paths modulo region-coupling from o to $\omega$, or a $\Rightarrow$-path $o \dashv \alpha \Rightarrow \omega$ for short: If there is an association path $\pi \in PAP(o, \beta<\gamma, \omega)$ then there is a $\Rightarrow$-path $o \dashv \beta \Rightarrow \omega$. If there is a $\Rightarrow$-path $q \dashv \gamma \omega$, there is also the $\Rightarrow$-path $o \dashv \alpha \gamma \omega$. And intrinsic to any notion of path, two consecutive $\Rightarrow$-paths can be concatenated to another $\Rightarrow$-path, and there are empty $\Rightarrow$-paths (which are neutral elements in path-concatenation).

For reasoning about coherence in §6.5, judgments about $\Rightarrow$-relationships and $\Rightarrow$-paths are derived with an annotation “via $\Pi$” that records the set of potential access.
paths on which they are based, the path-base. Equivalence \( \Pi \equiv \Pi' \) and inequivalence \( \Pi \not\equiv \Pi' \) between two path-bases means that the set \( H \) of the edges in \( \Pi \)’s paths is, respectively, the same, or a subset of, the set \( H \) of edges in \( \Pi' \)’s paths.

Note that all above association role sequences \( \alpha, \beta, \gamma \) may also be empty, and that all above objects \( o, \omega, q \) can be region objects in the extended graph \( g^o \). No distinction between objects’ regions and region objects (or region objects’ regions) needs to be made: A region-coupling \( o.\alpha \rightleftharpoons q.\beta.\gamma \) with the \( \gamma \)-region of a region object \( \omega = q.\beta \) is equivalent to the region-coupling \( o.\alpha \rightleftharpoons q.\beta.\gamma \). This follows formally from dummy edges \( q.\beta = q.\beta = \omega \), via \( q \rightarrow q.\beta = \omega \) and \( q.\beta.\gamma = \omega.\gamma \).

**Definition 13** The reserved ownership assumption means for any two ownership reservations \( \langle \tilde{\tau}, \phi \rangle \) and \( \langle \tilde{\tau}', \phi' \rangle \) on the same object \( v \), i.e., for ownership paths \( \tilde{\tau} \in PAP(w, \mu, u.\beta) \) and \( \tilde{\tau}' \in PAP(w', \mu', u'.\beta') \), and \( \Rightarrow \)-paths \( \phi = u \rightarrow u.\beta = v \) and \( \phi' = u' \rightarrow u'.\beta' = v \), that

a) \( w' = w \) and \( \mu' = \mu \), and
b) if \( \mu \) or \( \mu' \) is free then \( \tilde{\tau} \) and \( \tilde{\tau}' \) 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 \( \beta' = \beta = \epsilon \), so that \( u.\beta = u = v = u' = u'.\beta' \)).

7. \( \{ \text{CALL} \} \), THE SOLUTION. Return now to the \( j \)-th supply substep \( g_j = g_{j-1} \oplus r \mu_o. o \otimes s \mu_r. o \), i.e., the substitution of a received handle \( h'_o = r \mu_o. o \) for a sent handle \( h_o = s \mu_r. o \) in the presence of a call-link \( h_r = s \mu_r. r \). We can now better describe how source and target of the aforementioned initially new paths \( \pi_{\alpha} = r \mu_o. o \rightarrow o.\beta = \omega \) and internally new paths \( \pi = \pi' \cdot \pi_{\alpha} \cdot q \rightarrow \omega \) were connected before the supply of handle parameters extensible to association paths, i.e., with \( \mu_o(\bar{\alpha}_i) \in A \) for some \( \bar{\alpha}_i \): First, there was, respectively, a witness \( \text{wit}(\pi_{\alpha}) = s \mu_r. o \rightarrow o.\beta = \omega \) or \( \text{wit}(\pi) = \pi' \cdot r \mu_o(\alpha) \rightarrow r.\mu_o(\bar{\alpha}).\gamma \). Second, there was an initial \( \Rightarrow \)-path \( o \rightarrow o.\beta = \omega \) or \( \Rightarrow \)-path \( o \rightarrow o.\beta = \omega \), respectively, followed by a series \( o \mu_r. o \rightarrow o.\beta = \omega \) of \( \Rightarrow \)-paths with \( \mu_o(\bar{\alpha}_i) \in A \). Third, these paths were linked by extensions \( s \mu_r. o \mu_r(\bar{\alpha}_2).\gamma_2 \), \( \ldots \), \( o \mu_r. o \mu_r(\bar{\alpha}_i).\gamma_i \) of \( \Rightarrow \)-paths with \( \mu_o(\bar{\alpha}_i) \in A \). Third, these paths were linked by extensions \( s \mu_r. o \mu_r(\bar{\alpha}_2).\gamma_2 \), \( \ldots \), \( o \mu_r. o \mu_r(\bar{\alpha}_i).\gamma_i \) of the call-link, and \( s \mu_r. o \mu_r(\bar{\alpha}_i).\gamma_i \) of the sent handle to pairs of ownership paths \( r.\mu_o(\bar{\alpha}_i-1).\gamma_{i-1} \rightarrow s. \mu_o(\bar{\alpha}_i).\gamma_i \) 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:

```
    s \mu_r. o \rightarrow r o \mu_r. \rightarrow \mu_r. o \mu_o(\bar{\alpha}_i).\gamma_i \rightarrow \mu_r. o \mu_o(\bar{\alpha}_i+1).\gamma_{i+1} \rightarrow \ldots \rightarrow \mu_r. o \mu_o(\bar{\alpha}_h).\gamma_h \rightarrow \mu_r. o \mu_o(\bar{\alpha}_h-1).\gamma_{h-1} \rightarrow \ldots \rightarrow \mu_r. o \mu_o(\bar{\alpha}_1).\gamma_1 \rightarrow \omega
```

Each forward ownership path \( s \mu_r. o \mu_o(\bar{\alpha}_i).\gamma_i \) and following \( \Rightarrow \)-path together constitute a \( \mu_i \)-ownership reservation of \( s \) on, respectively, \( r.\mu_o(\bar{\alpha}_i+1).\gamma_{i+1} \) (for \( i < h \)) or \( \omega \).
(for $i = h$). The connection can be described more generally as one initial potential access path to an object $q_0$’s region object $q_0, \bar{a}_0, \bar{\gamma}_0$ followed by a “$\bar{a}_0, \bar{\gamma}_0$-bridge” from $q_0$ to $\omega$ defined as follows:

**Definition 14** An $\bar{a}$-bridge from $o$ to $\omega$ in a supply substep is an initial $\leftarrow$-path $o \leftarrow \bar{a}_i = \omega_0$ via $\Pi_0$ followed by a series of triples $(\pi_i', \pi_i, \varphi_i)$ of two ownership paths $\pi_i' = \omega_i \leftarrow \bar{a}_i \rightarrow s$ and $\pi_i = s \rightarrow q_i, \bar{a}_i, \gamma_i$, and a $\rightarrow$-path $\varphi_i = q_i \rightarrow \omega_i$ via $\Pi_i$ from $i = 1$ to some $n \geq 0$, such that $\omega_{n+1} = \omega$. In case of parameter mode $\mu_o = \text{co}<>$, each $\bar{a}_i$ is $e$, and ownership paths $\pi_i' = h_i' \circ \bar{\pi}_i'$ and $\pi_i = h_i \circ \bar{\pi}_i$ each start with $h_i', h_i \in \{h_r, h_o\}$ and have matching shape $s \leftarrow \bar{a}_i \rightarrow o \leftarrow \bar{\pi}_i'. \bar{\pi}_i \rightarrow \omega_i$ or $s \leftarrow \bar{a}_i \rightarrow r \leftarrow \bar{\pi}_i'. \bar{\pi}_i \rightarrow \omega_i$. In case of parameter mode $\mu_o$ with $\mu_o(\bar{a}) \in A$ for some $\bar{a}$, each $\pi_i = h_i \circ \bar{\pi}_i$ starts with $h_i = h_o$ and has shape $s \rightarrow \bar{a}_i \rightarrow o \rightarrow \bar{\pi}_i \rightarrow \omega_i$ with $\mu_o(\bar{a}_i) \in A$ while $\pi_i' = h_i' \circ \bar{\pi}_i'$ starts with $h_i' = h_r$ and has shape $s \rightarrow \bar{a}_i \rightarrow r \rightarrow \bar{\pi}_i \rightarrow \omega_i$. The bridge’s path-base is $\Pi_0 \cup \bigcup_{i=1}^n \{\pi_i, \pi_i'\} \cup \Pi_i$.

**Lemma 9** In $g_j^\oplus$, there are three kinds of new potential access paths $\pi \in \text{PAP}_{g_j^\oplus}(o, \mu, \omega)$:

1. **Co-closure paths** $\pi \in \text{PAP}_{g_j^\oplus}(o, \text{co}<> \rightarrow \omega)$ are co-paths that exist if $\mu_o$ is co$<\!<\!>$ They connect old co-objects of $r$ and $o$. That is, there is a (possibly empty) path of co-edges $o \rightarrow r$ or $o \rightarrow o \rightarrow r$ and $r \rightarrow o \rightarrow o$ or $r \rightarrow r$. If $\mu_o = \mu_r = \text{co}<>$ then $\pi$ has a precursor $\pi' \in \text{PAP}_{g_j^\oplus}^0(o, \mu, \omega)$ where $\pi' = \pi[h_r^{-1} \cdot h_o / h'_r, h_o^{-1} \cdot h_r / h'_o^{-1}]$.

2. **Internally new paths** $\pi \in \text{PAP}_{g_j^\oplus}(o, \mu, \omega)$ are non-co potential access paths that exist if $\mu_o(\bar{a}) \in \{\text{co}\} \cup A$ for some $\bar{a}$. They start with an unchanged edge $h$ and have a shape $o \leftarrow \bar{a} \rightarrow q \rightarrow o \rightarrow r \rightarrow \bar{a}_i \rightarrow h_o$ or $o \rightarrow r \rightarrow \bar{a}_i \rightarrow h_o$. In case of $\mu_o = \mu_r = \text{co}<>$, $\pi$ has a precursor $\pi' \in \text{PAP}_{g_j^\oplus}^0(o, \mu, \omega)$ with the same shape and $\pi' = \pi[h_r^{-1} \cdot h_o / h'_r, h_o^{-1} \cdot h_r / h'_o^{-1}]$ ($\pi$ is an “internally only new” path). Otherwise there was a witness $\text{init}(\pi) \in \text{PAP}_{g_j^\oplus}(o, \mu, o_0, \bar{a}_0, \gamma_0)$ and a non-trivial $\bar{a}_0, \gamma_0$-bridge from $q_0$ to $\omega$ ($\pi$ is an “internally really new” path). $\pi$ and $\text{init}(\pi)$ have the same shape and have a common, non-trivial prefix $\pi_1$: $\pi = \pi_1 \cdot \pi_2$, $\text{init}(\pi) = \pi_1 \cdot q_0 \cdot \bar{a}_0, \gamma_0$. If $\mu_o \neq \text{co}<>$ then $q_0 = r$ and $\bar{a}_0, \gamma_0 \neq e$; if $\mu_o = \text{co}<>$ then $q_0 \in \{r, o\}$ $\bar{a}_0, \gamma_0$ may be empty. The bridge’s path-base is a set $\Pi$ of paths which together with $\Pi_0 = \{h'_o\}$ or, in case of $\mu_o = \text{co}<>$, with non-empty $\Pi_o \subseteq \{h'_r, h_o^{-1}\}$ contains all edges of $\pi$’s second half $\bar{\pi}_2$, edges $\Pi_A = \{h_r, h_o\}$ and some dummy edges $\Pi_d: \{\pi_2\} \cup \Pi_A \cup \Pi_d \equiv \Pi \cup \Pi_o$.

3. **Initially new paths** $\pi \in \text{PAP}_{g_j^\oplus}^0(r, \mu, \omega)$ are non-co potential access paths $\pi = h_o \cdot \pi_1 \cdot \pi_2$ that exist if $\mu_o \neq \text{co}<>$. They start with the received handle and have some shape $r \rightarrow o \leftarrow \bar{a}_i \rightarrow h_o \rightarrow \bar{\pi}_i \rightarrow \omega_i$. In $g_j^\oplus$, they have a witness $\text{init}(\pi) = h_o \cdot \pi_1 \cdot \pi_2 \in \text{PAP}_{g_j^\oplus}^0(s, \mu_o, o, q_0, \bar{a}_0, \gamma_0)$ which starts with sent handle $h_o$ followed by edges $\pi_1$ shared with $\pi$ to $q_0$ and then dummy edges $\pi_2$ to $q_0, \bar{a}_0, \gamma_0$, and which has shape $s \rightarrow \bar{a}_i \rightarrow o \rightarrow \bar{\pi}_i \rightarrow \omega_i$. It is followed by a $\bar{a}_0, \gamma_0$-bridge from $q_0 = r$ to $\omega$. 

148
via II. In the simple case, the bridge is trivial, \( q_0, \tilde{a}_0, \tilde{a}_0, q_0 = q_0 = \omega, \) and \( \pi_2 = \pi'_2 = \epsilon. \) Otherwise, \( \pi_1 \) goes to \( q_0 = r, \) \( \pi_2 \) starts with \( h'_o \) followed by edges from II and more \( h'_o \)-edges, while II contains all edges of \( \pi_2 \) other than \( h_r \) and additionally the edges \( \Pi = \{ h_r, h_o \} \) and some dummy edges \( \Pi \); \( \{ \pi_2 \} \cup \Pi = \Pi \cup \{ \Pi \}. \)

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 \( \pi \) and \( \pi' \) with the same target, namely between

- \( \mathit{wit}(\pi) = o \xrightarrow{\mu} q_0, \tilde{a}_0, \tilde{a}_0 \) or \( \mathit{wit}(\pi) = s \xrightarrow{\mu_{\tilde{a}_n}} q_0, \tilde{a}_0, \tilde{a}_0, \) respectively, and \( \omega \) connected by \( q_0, \tilde{a}_0, \tilde{a}_0, \omega \) (and trivial \( \omega \) and \( \omega \));
- every two ownership paths \( s \xrightarrow{\mu} q_i, \tilde{a}_i, \tilde{a}_i \) and \( s \xrightarrow{\mu_{\tilde{a}_i}} s \) connected by \( q_i, \tilde{a}_i, \tilde{a}_i, \omega_i \); and\n- the ownership path \( s \xrightarrow{\mu} q_j, \tilde{a}_j, \tilde{a}_j \) on \( \pi \)'s side and \( q'_j, \tilde{a}_j, \tilde{a}_j, s \) on \( \pi' \)'s side connected by \( q_j, \tilde{a}_j, \tilde{a}_j, \omega_j \) on \( \pi \)'s side and \( s \) on \( \pi' \); and\n- every two ownership paths \( s \xrightarrow{\mu_{\tilde{a}_i}} q'_i, \tilde{a}_i, \tilde{a}_i \) and \( q'_i, \tilde{a}_i, \tilde{a}_i, s \) connected by \( q'_i, \tilde{a}_i, \tilde{a}_i, \omega_i \) on \( \pi \)'s side and \( \pi' \)'s side; and\n- and between \( s \xrightarrow{\mu_{\tilde{a}_i}} q'_i, \tilde{a}_i, \tilde{a}_i \) and \( \mathit{wit}(\pi') = q'_0, \tilde{a}_0, \tilde{a}_0, t_{\mu'} \) or \( \mathit{wit}(\pi') = q'_0, \tilde{a}_0, \tilde{a}_0, \mu_{\tilde{a}_i} \) connected by \( \omega_i, \tilde{a}_i, \tilde{a}_i, \omega_i \).

Consequently, \( \pi \)'s and \( \pi' \)'s source must coincide, and if one of them is free, they have the same shape and initial 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., \( r = s \) and \( \mu_j = \mu \). Then nothing changed at all:

\[ g_{\omega}^\oplus = g_{\omega}^\emptyset. \]

And in case of \( \mu_o = \mu_r = \sigma \), the only new edges are the \( j \)-th received handle \( h'_o = r \xrightarrow{\mu} o \) and its inverse \( h''_o = r \xrightarrow{\sigma} o \). Call-link \( h_r = r \xrightarrow{\sigma} s \), sent handle \( h'_o = s \xrightarrow{\sigma} o \) of mode \( \mu \), received handle \( o \xrightarrow{\sigma} o \) and its inverse \( o \xrightarrow{\sigma} o \). Then whenever a judgment \( g_{\omega}^\oplus \vdash \pi \in \text{PAP}(o, \mu, \omega) \) can be derived in \( g_{\omega}^\emptyset \), was possible to derive \( g_{\omega}^\emptyset \vdash \sigma(\pi) \in \text{PAP}(o, \mu, \omega) \) based on replacing derivations of \( g_{\omega}^\emptyset \vdash h'_o \in \text{PAP}(o, \mu, \omega) \) from \( h'_o \in g_{\omega}^\emptyset \) and of \( g_{\omega}^\emptyset \vdash h''_o \in \text{PAP}(o, \mu, \omega) \) from \( h''_o \in g_{\omega}^\emptyset \) to derivations of \( g_{\omega}^\emptyset \vdash \sigma(h'_o) \in \text{PAP}(o, \mu, \omega) \) and of \( g_{\omega}^\emptyset \vdash \sigma(h''_o) \in \text{PAP}(o, \mu, \omega) \). Consequently, all paths \( \pi \in \text{PAP}(o, \mu, \omega) \) have a precursor \( \sigma(\pi) \in \text{PAP}(o, \mu, \omega) \). If \( \pi \) 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 \( \pi \)' as witness \( \mathit{wit}(\pi) \in \text{PAP}(o, \mu, \omega) \), and a corresponding trivial connection \( \omega \xrightarrow{\mu} \omega \). Note that the witness \( \mathit{wit}(\pi) \) of an initially new ownership path \( \pi \) is an ownership path, namely a free one, since rep-modes in parameter mode \( \mu \) have been excluded, so that \( \pi \)'s mode \( \mu \) can only be free, and consequently \( \mathit{wit}(\pi) \)'s mode \( \mu \) is free too.
mode $\mu$ is $\text{co}<\text{co}$, it must be a sequence of co-paths containing $h'_o$ or $h'^{-1}_o$. But then precursor $\sigma(\pi)$ ensures that $o$ and $\omega$ are old co-objects of $r$ and $o$ (and of each other). $\pi$ is a co-closure path. If $\pi$ is not $\text{co}$, it is an internally-only new path.

For the remaining cases, proceed by induction on the derivation of judgments $g' \vdash \pi \in \text{PAP}(o, \mu, \omega)$. In the base case, $\pi$ is only one edge. It is new in $g' \vdash$ compared to $g_0 \vdash$ only if it is the $j^{th}$ received handle $h'_o = r \xrightarrow{h_0} o$ or its inverse $h'^{-1}_o = o \xrightarrow{\text{co}o\text{co}} r$ in case of $\mu_o = \text{co}<\text{co}$. If $\mu_o = \text{co}<\text{co}$ then $h'_o$ and $h'^{-1}_o$ are obviously co-closure paths.

If $\mu_o \neq \text{co}<\text{co}$ then $h'_o$ is initially new with shape $\bullet \rightarrow \text{co} \bullet \rightarrow \text{co} \rightarrow \bullet$, with the sent handle as witness $\text{wit}(h'_o) = h_o = s \xrightarrow{\text{co}o\text{coo}} o.e$ of the same shape, and with the trivial $o.e \rightarrow^e \omega_o$ as corresponding $\epsilon$-bridge.

In the induction step, $\pi$ is the extension $\pi_1 \cdot \pi_2$ of a potential access path $\pi_1 \in \text{PAP}_{g' \vdash}(o, \mu_1, q)$ by a co- or $\alpha$-path $\pi_2 \in \text{PAP}_{g' \vdash}(q, \mu_2, \omega)$.

First, consider the case where $\pi_1$ is unchanged:

(a) If $\pi_2$ is also unchanged then $\pi$ is unchanged again.

(b) If $\pi_2$ is a co-closure path then $\mu_2 = \text{co}<\text{co}$, so that $\mu = \mu_1$. If $\mu_1 = \mu = \text{co}<\text{co}$ then $\pi$ is a co-closure path: Unchanged $\pi_1$ means that $o$ is an old co-object of $q$, which by co-closure path $\pi_2$ is an old co-object of $r$ or $o$.

If $\mu_1 \neq \text{co}<\text{co}$ then $\pi$ is an internally really new path (because the case of $\mu_r = \mu_o = \text{co}<\text{co}$ was already covered before the induction). Since $\pi_2$'s source $q'$ is an old co-object of $q' = r$ or $o$, there must be a possibly empty prefix $\pi_q = q \xrightarrow{\text{co}o\text{coo}} q'$ of unchanged co-edges in $\pi_2 = \pi_q \cdot \pi_2$. Then extension $\pi_1 \cdot \pi_q \cdot \pi_2$. is the necessary witness $\text{wit}(\pi) \in \text{PAP}_{g' \vdash}(o, \mu, q', e)$ with $q' \in \{r, o\}$ containing all of $\pi$'s edges from $\pi_1$. A corresponding $\epsilon$-bridge to $\omega$ that contains all edges of $\pi_2$'s postfix $\pi_2$ other than $h'_o$ and $h'^{-1}_o$ consists of the trivial $q' \epsilon \rightarrow q'$ and three triples $\langle h_{q'}, h_r \cdot \pi_r, \varphi_r \rangle, \langle h_r, h_o \cdot \pi_o, \varphi_o \rangle, \langle h_o, h_\omega \cdot \pi_\omega, \varphi_\omega \rangle$: $h_{q'}$ is $h_r$ if $q' = r$ and $h_\omega$ if $q' = o$; $h_o$ is $h_r$ or $h_o$ which a path $\pi_\omega$ of unchanged co-edges can extend to $\omega$ (since $\omega$ was $r$'s or $o$'s co-object); the $\varphi_x$ are trivial $\equiv$-paths $x \epsilon \rightarrow x$; and $\pi_q$ and $\pi_2$ are paths $r \xrightarrow{\text{co}o\text{coo}} r$ and $o \xrightarrow{\text{co}o\text{coo}} o$ of co-edges that include all of $\pi_2$'s co-edges on, respectively, $r$'s and $o$'s side and then lead back to their starting point. All the ownership paths in this bridge obviously have the same shape shape $s \xrightarrow{\text{co}o\text{coo}} o \xrightarrow{\text{co}o\text{coo}} o \epsilon \rightarrow o$. The bridge’s path-base is $\Pi = \{h_{q'}, h_r \cdot \pi_r, h_r \cdot \pi_o, h_o \cdot \pi_o, h_o \cdot \pi_\omega \} \equiv \{h_r, h_o, \pi_r, \pi_o, \pi_\omega \}$ such that it contains, besides $h_r$ and $h_o$, all edges of $\pi_2$'s postfix $\pi_2$ not common with $\text{wit}(\pi)$, save $h'_o$ and/or $h'^{-1}_o$: $\{\pi_2\} \cup \{h_r, h_o\} \equiv \Pi \cup \Pi_o$ with non-empty $\Pi_o \subseteq \{h'_o, h'^{-1}_o\}$.

(c) If $\pi_2$ is initially new then $\pi_2 = h'_o \cdot \pi_3 \cdot \pi_4, q = r, \mu_2 = \beta<\text{co}$, and $\pi_2$ has a witness $\text{wit}(\pi_2) = h_o \cdot \pi_3 \cdot \pi_4$ and a $\alpha_0, \gamma_0$-bridge from $q_0$ to $\omega$. $\pi$ is an internally really new path whose witness $\text{wit}(\pi)$ is the extension $\pi_1 \cdot r \xrightarrow{\beta<\text{co}} r, \beta$ of $\pi_1$. For the corresponding $\beta$-bridge from $r$ to $\omega$, consider the two alternatives guaranteed by Lemma 13:

- There may be a $\equiv$-path $r \xrightarrow{\beta<\text{coo}} \omega_0$ via $\Pi_{2,0} \cup \{h_r, \text{wit}(\pi_2)\}$. Substituting it for the first $\equiv$-path $q_0 \xrightarrow{\beta<\text{coo}} \omega_0$ via $\Pi_{2,0}$ in $\pi_2$'s $\alpha_0, \gamma_0$-bridge via $\Pi_2$ produces a $\beta$-bridge to $\omega$ via $\Pi = \Pi_2 \cup \{h_r, \text{wit}(\pi_2)\} \equiv \Pi_2 \cup \{h_r, h_o, \pi_3, \pi_4\}$.  

150
• Otherwise there is an ownership path-pair \( \langle h_r \cdot h_\beta, \text{wit}(\pi_2) \rangle = r.\beta \leftarrow s.\beta \rightarrow q_0.\alpha_0.\gamma_0 \) where \( h_\beta = r.\beta . r.\beta \). The ownership paths have the right initial edges to connect the dummy edge \( h_\beta \) with \( \pi_2 \)’s \( \delta_0.\gamma_0 \)-bridge from \( q_0 \) to \( \omega \) via \( \Pi_2 \) to a \( \beta \)-bridge to \( \omega \) with path-base \( \Pi \equiv \Pi_2 \cup \{ h_r, h_\beta, h_\alpha, \pi_3 \cdot \pi_4 \} \).

By definition of initially new \( \pi_2 = h'_0 \cdot \pi_3 \cdot \pi_4 \), either \( \pi_4 = \epsilon \) and \( \Pi_2 = \emptyset \), or \( \{ \pi_4 \} \cup \Pi_\Lambda \cup \Pi_\Theta \equiv \Pi_2 \cup \{ h'_0 \} \). Hence in the \( \equiv \)-path case, we have \( \{ \pi_2 \} \cup \{ h_r, h_\alpha \} \cup (\Pi_\Theta \cup \{ \pi'_4 \} \equiv \Pi_2 \cup \{ h'_0 \} \). And in the ownership path-pair case, \( \{ \pi_2 \} \cup \{ h_r, h_\alpha \} \cup (\Pi_\Theta \cup \{ \pi'_4 \}, h_\beta \} \equiv \Pi \cup \{ h'_0 \} \).

(d) If \( \pi_2 \) internally really new then \( \pi \) is an \textit{internally really new path} again: \( \pi_2 \)’s old witness \( \text{wit}(\pi_2) \in PAP=q_0(q, \mu_2, q_0.\alpha_0.\gamma_0) \) is extended by precursor \( \pi_1 \) to a witness \( \text{wit}(\pi) = \pi_1 \cdot \text{wit}(\pi_2) \in PAP=q_0(q, \mu, q_0.\alpha_0.\gamma_0) \) for \( \pi \). The corresponding \( \delta_0.\gamma_0 \)-bridge from \( q_0 \) to \( \omega \) via \( \Pi = \Pi_2 \) is guaranteed by \( \pi_2 \). If \( \pi_2 = \pi_3 \cdot \pi_4 \) and \( \text{wit}(\pi_2) = \pi_3 \cdot \pi'_4 \) with maximal common prefix \( \pi_3 \), then \( \{ \pi_4 \} \cup \Pi_\Lambda \cup \Pi_\Theta \equiv \Pi_2 \cup \Pi_\Theta \) is guaranteed. Since \( \pi_3 \) is also the postfix of \( \pi = \pi_1 \cdot \pi_2 = (\pi_1 \cdot \pi_3) \cdot \pi_4 \) not common with \( \text{wit}(\pi) = \pi_1 \cdot \text{wit}(\pi_2) = (\pi_1 \cdot \pi_3) \cdot \pi'_4 \), and since \( \Pi = \Pi_2 = \Pi \), we have \( \{ \pi_4 \} \cup \Pi_\Lambda \cup \Pi_\Theta \equiv \Pi \cup \Pi_\Theta \).

Second, \textit{co-closure paths} \( \pi_1 \) extend always to another \textit{co-closure path}: Since their mode \( \mu_1 = \text{co}<> \), they can only be extended by \textit{co-paths}, so that no new witness is necessary and initially new and internally new paths \( \pi_2 \) are excluded. \( \pi \)'s source \( o \) is an old co-object of \( r \) or \( o \) since it is the source of co-closure path \( \pi_1 \). \( \pi \)'s target \( \omega \) is by definition also an old co-object of \( r \) or \( o \) if \( \pi_2 \) is a co-closure path. And if \( \pi_2 \) is unchanged then \( \pi_2 \)'s precursor \( \pi'_2 \) of mode \( \text{co}<> \) means that \( \omega \) is an old co-object of \( q \), which is an old co-object of \( r \) or \( o \) since it is target of co-closure path \( \pi_1 \).

Third, if \( \pi_1 \) is \textit{initially new} or \textit{internally really new}, and \( \pi_2 \) is a \textit{co-path}, then \( \pi_2 \) can neither be initially new nor internally really new, and \( \pi \) is the same kind of new potential access path: Its witness is the same as that of \( \pi_1 \): \( \text{wit}(\pi) = \text{wit}(\pi_1) \), except in one case where the bridge is empty (see below). The corresponding \( \delta_0.\gamma_0 \)-bridge from \( q_0 \) to \( \omega \) follows from \( \pi_1 \)'s \( \delta_0.\gamma_0 \)-bridge from \( q_0 \) to \( q \):

(a) If \( \pi_2 \) is unchanged, consider the last \( \equiv \)-path \( q_j \xrightarrow{\delta_0.\gamma_0} \omega \) via \( \Pi_{1,j} \) in \( \pi_1 \)'s \( \delta_0.\gamma_0 \)-bridge to \( q \):

• If it is not empty, i.e., \( \delta_0.\gamma_0 \neq \epsilon \) or \( q_j \neq \omega_j \), then \( \pi_2 \) can extend it to \( q_j \xrightarrow{\delta_0.\gamma_0} \omega \) via \( \Pi' = \Pi_{1,j} \cup \{ \pi \} \) for some \( \pi \in \Pi_{1,j} \) (Lemma 11). By substituting it for \( q_j \xrightarrow{\delta_0.\gamma_0} \omega_j \) via \( \Pi_{1,j} \) in \( \pi_1 \)'s \( \delta_0.\gamma_0 \)-bridge to \( q \) is redirected to \( \omega \) via \( \Pi = \Pi_{1,j} \cup \{ \pi_2 \} \).

• If \( \delta_0.\gamma_0 \neq \epsilon \) and \( q_j = \omega_j \), and \( j > 0 \), then the last \( \equiv \)-path is the trivial \( q_j \xrightarrow{\delta_0.\gamma_0} q \) via \( \emptyset \) and the last ownership path is \( \pi = s.\beta \xrightarrow{\delta_0.\gamma_0} q_j \). Extension \( \pi \cdot \pi_2 \) is the ownership path \( s.\beta \xrightarrow{\delta_0.\gamma_0} q \) via \( \emptyset \) which redirects \( \pi_1 \)'s \( \delta_0.\gamma_0 \)-bridge to \( q \) via \( \Pi = \Pi_{1,j} \cup \{ \pi_2 \} \).

• If \( \delta_0.\gamma_0 \neq \epsilon \) and \( q_j = \omega_j \), and \( j = 0 \), then the last ownership path is \( \pi = s.\beta \xrightarrow{\delta_0.\gamma_0} q \) via \( \emptyset \) which redirects \( \pi_1 \)'s \( \delta_0.\gamma_0 \)-bridge to \( q \) via \( \Pi = \Pi_{1,j} \cup \{ \pi_2 \} \).

This applies only to the case of initially new \( \pi_1 \). Its witness \( \text{wit}(\pi_1) \)
is $s \overset{\ell}{\to} q_j, \bar{\gamma}_j, \bar{\gamma}'_j = \omega'_j = q$. Then $\text{wit}(\pi) = \text{wit}(\pi_1) \cdot \pi_2 = o \overset{\ell}{\to} \omega = \omega, e$. The corresponding $e$-bridge is the trivial initial $\Rightarrow$-path via $\Pi = \emptyset$ $\omega \overset{\Rightarrow}{\to} \omega$ followed by the empty triple series.

(b) If $\pi_2$ is a co-closure path then $\mu_o = \text{co}<$, so that the case of initially new $\pi_1$ does not apply. Co-closure path $\pi_2$ means that $q$ as well as $\omega$ are old co-objects of $r$ or $o$, i.e., there are co-paths $\pi_q = r \overset{\ell \cdot o}{\to} q$ or $o \overset{\ell \cdot o}{\to} q$, and $\pi_\omega = r \overset{\ell \cdot o}{\to} \omega$ or $o \overset{\ell \cdot o}{\to} \omega$. Since $\mu_o = \text{co}<, \mu_r \circ \mu_o = \mu_r$. Hence $h_r$ or $h_o$ combined with $\pi_q$ or $\pi_\omega$ to potential access paths $\hat{\pi}' \in \mathit{PAP}_{\emptyset} (s, \mu_r, q)$ and $\hat{\pi} \in \mathit{PAP}_{\emptyset} (s, \mu_r, \omega)$ of shape $s \overset{\ell \cdot r}{\to} r \overset{\ell \cdot o}{\to} o \overset{-}{\to} \cdot$. Since the case of $\mu_r = \mu_o = \text{co}<$ was already handled before the induction, $\mu_r$ must be a free or rep mode: $\mu_o = \text{co}<$ means that $\Gamma_n, \kappa_n \vdash \hat{\pi}$ excludes read and association modes for $\mu_r$. Hence $(\hat{\pi}', \hat{\pi})$ is an ownership-path-pair $q \overset{\ell \cdot r}{\to} s \overset{\ell \cdot o}{\to} \omega = \omega, e$ with call-link or sent handle as initial edge and with the same shape. It, and the trivial $\Rightarrow$-path $\omega \overset{\Rightarrow}{\to} \omega$ via $\Omega$ extend $\pi_1$'s $\bar{\alpha}_0, \bar{\gamma}_0$-bridge from $\omega_0$ to $q$ via $\Pi_1$ to $\pi_1$'s $\bar{\alpha}_0, \bar{\gamma}_0$-bridge from $\omega_0$ to $\omega$ via $\Pi = \Pi_1 \cup \{\pi_q, \pi_\omega\} \cup \emptyset$. In order to ensure coverage by $\Pi$ for all edges from $\pi_2$ (see below), simply chose $\pi_q$ so that it includes $\pi_2$'s co-edges on $q$'s side and chose $\pi_\omega$ so that it includes $\pi_2$'s co-edges on $\omega$'s side.

Initially new $\pi_1 = h'_o \cdot \pi_3 \cdot \pi_4$ means either $\pi_3 = \emptyset$ and $\Pi_1 = \emptyset$, or $\{\pi_4\} \cup \Pi_\Lambda \cup \Pi_\Theta = \Pi_1 \cup \{h'_o\}$. And internally really new $\pi_1 = \pi_3 \cdot \pi_4$ means $\{\pi_4\} \cup \Pi_\Lambda \cup \Pi_\Theta = \Pi_1 \cup \Pi_\Omega$. In internally really new $\pi$, $\pi_3$ is still the maximal common prefix, and $\pi_4 \cdot \pi_2$ is the rest: $\pi = \pi_1 \cdot \pi_2 = \pi_3 \cdot (\pi_4 \cdot \pi_2)$.

In (a), the case of $\Pi = \Pi_1 = \emptyset$ means for initially new $\pi_1$ that $\pi_4 = \pi'_4 = \emptyset$ and $\Pi_1 = \emptyset$. Hence the postfix of $\pi$ and $\text{wit}(\pi)$ are the same: $\pi = h'_o \cdot \hat{\pi}$ and $\text{wit}(\pi) = h_o \cdot \hat{\pi}$ with $\Pi = \emptyset$. The case of $\Pi = \Pi_1 \cup \{\pi_2\}$ means for initially new $\pi_1$ that $\{\pi_4 \cdot \pi_2\} \cup \Pi_\Lambda \cup \Pi_\Theta = \Pi \cup \{h'_o\}$. For internally really new $\pi$, it means $\{\pi_4 \cdot \pi_2\} \cup \Pi_\Lambda \cup \Pi_\Theta = \Pi \cup \Pi'_o$.

In (b), a co-closure path $\pi_2$'s edges are $h'_o$ or $h'^{-1}_o$ or both, as well as old co-edges on $r$'s and on $o$'s side, which are contained in $\pi_q$ and $\pi_\omega$, or vice versa. Hence $\Pi = \Pi_1 \cup \{\pi_q, \pi_\omega\}$ for internally really new $\pi_1$ that $\{\pi_4 \cdot \pi_2\} \cup \Pi_\Lambda \cup \Pi_\Theta = \Pi \cup \Pi'_o$ where $\Pi'_o$ is that non-empty subset of $\{h'_o, h'^{-1}_o\}$ contained in $\pi_2$. The case of initially new $\pi_1$ does not apply.

Fourth, if $\pi_1$ is initially new or internally really new, and $\pi_2$ is an association path, then $\pi_2$ cannot be a co-closure path, and $\pi$ is the same kind of path as $\pi_1$. The witness $\text{wit}(\pi_1) = s \overset{\ell \cdot o}{\to} o' \overset{\ell \cdot o}{\to} q_0, \bar{\alpha}_0, \bar{\gamma}_0 \text{ or } o \overset{\ell \cdot o}{\to} o' \overset{\ell \cdot o}{\to} q_0, \bar{\alpha}_0, \bar{\gamma}_0$, respectively, is extended by dummy edge $\pi'_\beta = q_0, \bar{\alpha}_0, \bar{\gamma}_0 \overset{\beta \cdot o}{\to} q_0, \bar{\alpha}_0, \bar{\gamma}_0$, 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 $\bar{\alpha}_0, \bar{\gamma}_0, \beta$-bridge from $q_0$ to $\omega$ follows from $\pi_1$'s $\bar{\alpha}_0, \bar{\gamma}_0$-bridge from $q_0$ to $q$.

(a) If $\pi_2$ is unchanged then $\pi_1$'s $\bar{\alpha}_0, \bar{\gamma}_0$-bridge via $\Pi_1$ can be extended along $\pi_2 = q \overset{\beta \cdot o}{\to} \omega$ to a $\bar{\alpha}_0, \bar{\gamma}_0, \beta$-bridge to $\omega$ via $\Pi = \Pi_1 \cup \{\pi_2\} \cup \Pi_\beta$ (Lemma 14). In case
of initially new \( \pi_1 = h'_0 \cdot \pi_3 \cdot \pi_4 \) while \( \pi_4 = \pi'_4 = \epsilon \) and \( \Pi_1 = \emptyset \), witness \( \text{wit}(\pi_1) = h'_0 \cdot \pi_3 \cdot \pi'_4 \) targets \( q \) and can be extend along \( \pi_2 \) to a witness \( \text{wit}(\pi) = \text{wit}(\pi_1) \cdot \pi_2 \) targeting \( \omega \).

(b) If \( \pi_2 \) is a initially new path \( \pi_2 = h'_0 \cdot \pi_5 \cdot \pi_6 \) then \( q = r \). For \( \pi_2 \), Lemma 13 guarantees two cases:

- There may be a \( \equiv \)-path \( r \rightarrow q'_0 \rightarrow \omega'_0 \) via \( \Pi_{2,0} \cup \{h_r, \text{wit}(\pi_2)\} \equiv \Pi_{2,0} \cup \{h_r, h_o, \pi_5 \cdot \pi_6\} \). Then \( \pi_1 \)'s \( \alpha_0, \gamma_0 \)-bridge from \( q_0 \) to \( q = r \) via \( \Pi_1 \) can be extended along this \( \equiv \)-path to a \( \alpha_0, \gamma_0 \)-bridge to \( \omega'_0 \) via \( \Pi'_1 \equiv \Pi_1 \cup \{h_r, \text{wit}(\pi_2)\} \cup \Pi_2 \) (Lemma 14). It is extended by the rest of \( \pi_2 \)'s bridge to a \( \alpha_0, \gamma_0 \)-bridge to \( \omega \) via \( \Pi \equiv \Pi_1 \cup \Pi_2 \cup \{h_r, \text{wit}(\pi_2)\} \cup \Pi_3 \equiv \Pi' \equiv \Pi_1 \cup \Pi_2 \cup \{h_r, \pi_5 \cdot \pi_6\} \cup \Pi_4 \cup \Pi_5 \) where \( \Pi_4 = \{h_r, h_o\} \) and \( \Pi_5 = \{h_r, h_o, \pi_5 \cdot \pi_6\} \).

- Otherwise there is an ownership path-pair \( \langle h_r \cdot h_\beta, \text{wit}(\pi_2) \rangle = r \beta \rightarrow q'_0 \rightarrow q'_0 \rightarrow \omega'_0 \rightarrow \omega'_0 \) via \( \Pi_{2,0} \) in \( \pi_2 \)'s \( \alpha_0, \gamma_0 \)-bridge via \( \Pi_1 \) mean \( q = r \rightarrow \beta \rightarrow \omega_0 \) via \( \Pi_{2,0} \cup \{\text{wit}(\pi_2)\} \) (Lemma 12). Along this \( \equiv \)-path, \( \pi_1 \)'s \( \alpha_0, \gamma_0 \)-bridge from \( q_0 \) to \( q = r \) via \( \Pi_1 \) can be extended along dummy edge \( h_\beta = r \rightarrow \beta \rightarrow \omega_0 \) via \( \Pi_1 \cup \{\text{wit}(\pi_2)\} \cup \Pi_2 \) (Lemma 14). It is extended by the rest of \( \pi_2 \)'s bridge to a \( \alpha_0, \gamma_0 \)-bridge to \( \omega \) via \( \Pi = \Pi_1 \cup \Pi_2 \cup \{\text{wit}(\pi_2)\} \cup \Pi_3 \).

(c) If \( \pi_2 \) is internally really new then \( \pi_2 \)'s witness \( \text{wit}(\pi_2) = q \equiv 2 \rightarrow q'_0 \rightarrow h'_0 \) and the first \( \equiv \)-path \( q'_0 \rightarrow h'_0 \rightarrow \omega'_0 \) via \( \Pi_{2,0} \) in \( \pi_2 \)'s \( \alpha_0, \gamma_0 \)-bridge via \( \Pi_2 \) mean \( q = \beta \rightarrow \omega_0 \) via \( \Pi_{2,0} \cup \{\text{wit}(\pi_2)\} \) (Lemma 12). Along this \( \equiv \)-path, \( \pi_1 \)'s \( \alpha_0, \gamma_0 \)-bridge from \( q_0 \) to \( q = r \) via \( \Pi_1 \) can be extended along dummy edge \( h_\beta = r \rightarrow \beta \rightarrow \omega_0 \) via \( \Pi_1 \cup \{\text{wit}(\pi_2)\} \cup \Pi_2 \) (Lemma 14). It is extended by the rest of \( \pi_2 \)'s bridge to a \( \alpha_0, \gamma_0 \)-bridge to \( \omega \) via \( \Pi = \Pi_1 \cup \Pi_2 \cup \{\text{wit}(\pi_2)\} \cup \Pi_3 \).

Initially new \( \pi_1 = h'_0 \cdot \pi_3 \cdot \pi_4 \) means either \( \pi_4 = \epsilon \) and \( \Pi_1 = \emptyset \), or \( \{\pi_4\} \cup \Pi_4 = \Pi_1 \cup \{h'_0\} \). And internally really new \( \pi_1 = \pi_3 \cdot \pi_4 \) means \( \{\pi_4\} \cup \Pi_4 = \Pi_1 \cup \Pi_3 \). In internally really new \( \pi, \pi_3 \) is still the maximal common prefix, and \( \pi_4 \cdot \pi_2 \) is the rest: \( \pi = \pi_1 \cdot \pi_2 = \pi_3 \cdot (\pi_4 \cdot \pi_2) \).

In (a), \( \Pi = \Pi_1 \cup \{\pi_2\} \cup \Pi_3 \). For initially new \( \pi \), we get \( \{\pi_4 \cdot \pi_2\} \cup \Pi_4 \cup \{\Pi_\eta \cup \Pi_\beta \cup \{h'_0\}\} \equiv \Pi \cup \{h'_0\} \). For internally really new \( \pi \), we get \( \{\pi_4 \cdot \pi_2\} \cup \Pi_4 \cup \{\Pi_\eta \cup \Pi_\beta \} \equiv \Pi \cup \Pi'_\eta \).

In (b), initially new \( \pi_2 = \pi_5 \cdot \pi_6 \) means \( \{\pi_6\} \cup \Pi_4 \cup \Pi'_\eta \equiv \{h'_0\} \cup \Pi_2 \); and we have \( \Pi = \Pi_1 \cup \Pi_2 \cup \{\pi_5\} \cup \Pi_3 \cup \Pi_4 \). This means for initially new \( \pi_1 \) that \( \{\pi_4 \cdot \pi_2\} \cup \Pi_4 \cup \{\Pi_\eta \cup \Pi_\beta \} \equiv \Pi \cup \{h'_0\} \). And for internally really new \( \pi_1 \), it means \( \{\pi_4 \cdot \pi_2\} \cup \Pi_4 \cup \{\Pi_\eta \cup \Pi'_\eta \cup \Pi'_\beta \} \equiv \Pi \cup \Pi'_\eta \).

In (c), internally really new \( \pi_2 = \pi_5 \cdot \pi_6 \) with witness \( \text{wit}(\pi_2) = \pi_5 \cdot \pi_6 \) means \( \{\pi_6\} \cup \Pi_4 \cup \{\Pi_\eta \cup \Pi'_\eta \} \equiv \Pi_2 \cup \Pi'_\eta \). Hence \( \Pi = \Pi_1 \cup \Pi_2 \cup \{\text{wit}(\pi_2)\} \cup \Pi_3 \) means for initially new \( \pi_1 \) that \( \{\pi_4 \cdot \pi_2\} \cup \Pi_4 \cup \{\Pi_\eta \cup \Pi'_\eta \cup \Pi_\beta \} \equiv \Pi \cup \{h'_0\} \), and means for internally really new \( \pi_1 \) that \( \{\pi_4 \cdot \pi_2\} \cup \Pi_4 \cup \{\Pi_\eta \cup \Pi'_\eta \cup \Pi_\beta \cup \{\pi_6\}\} \equiv \Pi \cup \Pi'_\eta \cup \Pi'_\beta \).
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 \( \Leftarrow \)-paths under co-paths, and the closure of \( \Leftarrow \)-paths under the \( \Leftarrow \)-regions corresponding to region objects.

**Lemma 10 (Shape extension)** If potential access path \( \pi_1 \in PAP(o, \mu_1, q) \) has shape \( o \Leftarrow u \Leftarrow \alpha \Leftarrow \bullet \) and \( \pi_2 \in PAP(q, \mu_2, \omega) \) is a potential access path of mode \( \mu_2 = co\alpha \) or \( \alpha\omega \) then \( \pi_1 \bullet \pi_2 \) has shape \( o \Leftarrow u \Leftarrow \alpha \Leftarrow \bullet \) or shape \( o \Leftarrow u \Leftarrow \alpha \Leftarrow \bullet \) and \( \pi_2 \) as respectively.

**Proof:** Shape \( o \Leftarrow u \Leftarrow \alpha \Leftarrow \bullet \), with \( \alpha = \alpha_1 \ldots \alpha_n \) means that \( \pi_1 \) is a path \( o \Leftarrow u \Leftarrow \alpha \Leftarrow \bullet \) and \( \pi_2 \) has shape \( \pi_2 = q \leftarrow w \). Hence if \( \mu_2 = \alpha\omega \), then its extension by \( \pi_2 = o \Leftarrow u \Leftarrow \alpha \Leftarrow \bullet \) and \( \alpha = \epsilon \), \( \pi_1 \) is \( o \Leftarrow u \Leftarrow \alpha \Leftarrow \bullet \) and \( \pi_2 \) is a path \( o \Leftarrow u \Leftarrow \alpha \Leftarrow \bullet \) with shape \( o \Leftarrow u \Leftarrow \alpha \Leftarrow \bullet \). In case of \( \mu_2 = co\alpha \) and \( \alpha \neq \epsilon \), there is a last association path \( u_1 \Leftarrow \alpha \) in \( \pi_1 \). It is extended by \( \pi_2 = q \leftarrow w \) to a path \( o \Leftarrow u \Leftarrow \alpha \Leftarrow \bullet \). Hence \( \pi_1 \bullet \pi_2 \) is \( o \Leftarrow u \Leftarrow \alpha \Leftarrow \bullet \), with the obvious shape \( o \Leftarrow u \Leftarrow \alpha \Leftarrow \bullet \).

**Lemma 11** If \( o \Leftarrow q \leftarrow \varepsilon \gamma \) via \( \Pi \) with \( \gamma \neq \varepsilon \) or \( \gamma \neq q \), and \( \pi \in PAP(q, co\epsilon, \omega) \) then \( o \Leftarrow q \leftarrow \varepsilon \gamma \) via \( \Pi' = \Pi \cup \{\pi \} \) for some \( \pi' \in \Pi \).

**Proof by induction on the definition of** \( o \Leftarrow q \leftarrow \varepsilon \gamma \): Note that the condition excludes trivial \( \Leftarrow \)-paths \( o \Leftarrow q \leftarrow \varepsilon \gamma \). Hence the only base case is a \( \Leftarrow \)-path via \( \{\pi' \} \) with \( \alpha = \beta \) based on potential access path \( \pi' \in PAP(o, \beta<\epsilon, q) \). Then also \( \pi' \bullet \pi \in PAP(o, \beta<\epsilon, q) \) and thus \( o \Leftarrow q \leftarrow \varepsilon \gamma \) via \( \{\pi' \} \). In the induction step, if \( o \Leftarrow q \leftarrow \varepsilon \gamma \) because \( o\alpha = q' \varepsilon \gamma \) via \( \Pi_1 \) and \( q' \varepsilon \gamma \) via \( \Pi_2 \), then \( q' \varepsilon \gamma \) via \( \Pi' = \Pi_1 \cup \Pi_2 \) by induction hypothesis, and thus \( o \Leftarrow q \leftarrow \varepsilon \gamma \) via \( \Pi_1 \cup \Pi_2 \) by \( o\alpha = q' \varepsilon \gamma \). If \( o \Leftarrow q \leftarrow \varepsilon \gamma \) because \( o \Leftarrow q \leftarrow \varepsilon \gamma \) via \( \Pi_1 \) and \( q \varepsilon \gamma \) via \( \Pi_2 \) with \( \alpha_1 \bullet \alpha_2 = \alpha \) then in case of \( \alpha_2 \neq \epsilon \) or \( \gamma \neq q' \), \( q' \varepsilon \gamma \) via \( \Pi_2 \) by induction hypothesis, and thus \( o \Leftarrow q \leftarrow \varepsilon \gamma \) via \( \Pi_1 \leftarrow \Pi_2 \). And in case of \( \alpha_2 = \epsilon \) and \( \gamma \neq q' \), \( q' \varepsilon \gamma \) via \( \Pi_2 \) by induction hypothesis. Hence \( o \Leftarrow q \leftarrow \varepsilon \gamma \) via \( \Pi_1 \leftarrow \Pi_2 \) by induction hypothesis. That is \( o \Leftarrow q \leftarrow \varepsilon \gamma \) via \( \Pi_1 \leftarrow \Pi_2 \) since \( \alpha_1 \bullet \epsilon = \alpha_1 \bullet \alpha_2 = \alpha \).

**Lemma 12** If \( o \Leftarrow q \Leftarrow \varepsilon \gamma \) via \( \Pi \) and \( q \varepsilon \gamma \) via \( \Pi' \) then \( o \Leftarrow q \leftarrow \varepsilon \gamma \) via \( \Pi \cup \Pi' \).

**Proof:** Axiom \( q, o \Leftarrow \varepsilon \gamma \) via \( \emptyset \) implies \( (q, o) \varepsilon \gamma \Leftarrow \emptyset \) via \( \emptyset \). With \( \bar{\beta} = \epsilon \), this makes \( q \varepsilon \gamma \) via \( \emptyset \) mean \( q \Leftarrow o \Leftarrow \varepsilon \gamma \) via \( \Pi \cup \emptyset \). It extends \( o \Leftarrow \varepsilon \gamma \) via \( \Pi \cup \Pi' \).
The next three lemmas are in the context of parameter supply substeps during a 
legal \( \text{call} \)-reduction step, i.e., with a typing \( \Gamma_n, \kappa_n \vdash \ell : \hat{\tau} \) for the operation call expression \( \text{red} \). They concern the graph \( \mathit{g}' \) after addition of a new received handle
\( h_o = r \cdot \mu_o \cdot o \) and removal of old sent handle \( h'_o = s \cdot \mu_o \cdot o \) from the previous graph \( \mathit{g} \), i.e., \( \mathit{g}' = \mathit{g} \oplus r \cdot \mu_o \cdot o \oplus s \cdot \mu_o \cdot o \).

**Lemma 13** Consider an initially new association path \( \pi \in \text{PAP}_g(r, \beta\leftrightarrow, \omega) \) whose \( \alpha_0, \gamma_0 \)-bridge from \( q_0 \) to \( \omega \) in \( \mathit{g} \) starts with \( {\Leftarrow\Rightarrow}-\text{path} q_0 \cdot \alpha_0, \gamma_0 \cdot \omega_0 \) via \( \Pi_0 \). In \( \mathit{g} \), there was
- there was a \( {\Leftarrow\Leftarrow}\)-path \( r \cdot \beta\leftrightarrow, \omega_0 \) via \( \{h_r, \\text{mit}(\pi)\} \cup \Pi_0 \), or
- \( h_r \cdot r \cdot \beta\leftrightarrow, \omega_0 \) and \( \text{mit}(\pi) \) constitute an ownership path-pair \( r, \beta \leftarrow \mu \cdot s \rightarrow q_0, \alpha_0, \gamma_0 \)
of shape \( s \cdot \mu_o \cdot r \cdot \beta\leftrightarrow, \omega_0 \), and \( s \cdot \mu_o \cdot o \cdot \beta\leftrightarrow, \omega_0 \), respectively.

**Proof:** First, an initially new \( \pi \) has some shape \( r \cdot \mu_o \cdot o \cdot \beta\leftrightarrow, \omega_0 \) with \( \mu_o(\alpha_0, \gamma_0) = \beta \) since \( \pi \)'s mode is \( \beta\leftrightarrow \). But if \( \mu_o \) contains \( \beta \), then the existence \( h'_o = s \cdot \mu_o \cdot o \) presupposes for some \( \mu \) a corresponding correlation \( \beta = \mu \) in the call-link’s mode \( \mu_r \). It specifies that \( \mu_r \cdot \beta\leftrightarrow = \mu \), meaning that \( \pi \)'s witness \( \text{mit}(\pi) \) has mode \( \mu_r \cdot o = \mu_r \cdot \beta\leftrightarrow = \mu \).

Second, as a mode with a correlation \( \beta = \mu \), \( \mu_r \) cannot be a \( \text{co} \)- or association mode; and \( \Gamma_n, \kappa_n \vdash \ell : \hat{\tau} \) excludes that \( \mu_r \) is \( \text{read} \)-- \( \mu_r \) can only be \( \text{free} \), \( \text{rep} \), or an association mode:
- In case of an association mode \( \mu = \alpha \leftrightarrow \), the call-link \( h_r = s \cdot \mu \cdot r \) established region-coupling \( r, \beta \leftarrow s, \alpha \) via \( \{h_r\} \) because \( \mu_r = \text{free} \), \( \beta = \alpha \leftrightarrow \), or \( \text{rep} \). But then witness \( \text{mit}(\pi) \in \text{PAP}_g(s, \alpha \leftrightarrow, q_0, \alpha_0, \gamma_0) \) means the \( {\Leftarrow\Rightarrow}-\text{path} q_0 \cdot \beta\leftrightarrow, \omega_0 \) via \( \{h_r, \text{mit}(\pi)\} \). It and the first \( {\Leftarrow\Rightarrow}-\text{path} q_0 \cdot \alpha_0, \gamma_0 \cdot \omega_0 \) of \( \pi \)'s \( \alpha_0, \gamma_0 \)-bridge entail \( r \cdot \beta\leftrightarrow, \omega_0 \) via \( \{h_r, \text{mit}(\pi)\} \cup \Pi_0 \) (Lemma 12).
- In case of a \( \text{free} \) or \( \text{rep} \) mode \( \mu \), consider on one side the extension of the call-link with correlation \( \beta = \mu \) to the \( \mu \)-path \( \hat{\pi} = s \cdot \mu \cdot r \cdot \beta\leftrightarrow, \omega_0 \) with shape \( s \cdot \mu \cdot r \cdot \beta\leftrightarrow, \omega_0 \)
- \( \cdot \beta \rightarrow, \omega_0 \). On the other side is \( \pi \)'s witness \( \text{mit}(\pi) \in \text{PAP}_g(s, \mu, q_0, \alpha_0, \gamma_0) \) with shape \( s \cdot \mu \cdot o \cdot \beta\leftrightarrow, \omega_0 \).

**Lemma 14 (Bridge extension)** Consider an initially new or internally really new path’s \( \alpha_0 \)-bridge from \( q_0 \) to \( \omega \) via \( \Pi \). For any \( {\Leftarrow\Leftarrow}\)-path \( q \cdot \beta\leftrightarrow, \omega \) via \( \Pi' \), there is a \( \alpha_0, \beta \)-bridge from \( q_0 \) to \( \omega \). Its path-base \( \Pi' \equiv \Pi \cup \Pi' \cup \Pi_\beta \) contains beside \( \Pi \) and \( \Pi' \) a certain set \( \Pi_\beta \) of dummy edges \( u \cdot \beta\leftrightarrow, u, \beta \).

155
Proof: Each \(\equiv\)-path \(q_i - \delta_0 - \gamma_{in} - \omega_{i+1}\) via \(\Pi_i\) in the bridge can be extended to \(q_i - \delta_\omega - \gamma_{in} - \omega_{i+1},\beta\) via \(\Pi_i \cup \{\omega_{i+1} - \beta_{in} - \omega_{i+1}\}\), and the final \(q_n - \delta_0 - \gamma_{in} - \omega_{n+1} = q\) via \(\Pi_n\) can be extended to \(q_n - \delta_\omega - \gamma_{in} - \omega\) via \(\Pi_n \cup \Pi'\). But a corresponding extension of the bridge’s ownership path-pairs \(\langle \pi_i', \pi_i \rangle = \omega_i \delta_\omega \cdot s \cdot q_i \alpha_i \cdot \beta\) for \(\alpha_i = \beta \in \mathbb{A}\) for some decomposition \(\alpha_i = \alpha_i \beta\). This means that the received handle's extension along a \(\alpha\)-sequence of association paths is a path of mode \(\beta\), which cannot be extended further to a read path.

- There can never be correlations \(\beta = \alpha\) to \(\alpha\).
- There can be no correlation to read in the mode \(\mu\) of extensions \(\pi_i'\) and \(\pi_i\) of \(h_i\) or \(h_i'\): \(\Gamma_n, \kappa_n \vdash \beta : \tau\) ensures that the base-mode \(\mu_o \mu_0 (\beta)\) of extensions of the sent handle are read only if corresponding extension of the received handle have base-mode \(\mu_0 (\beta) = \text{read}\) too. This excludes the case of mode \(\mu_0 = \alpha\) from which no read mode can be extracted. In case of \(\mu_0 (\beta) \in A\) for some \(\beta\), \(\pi_i\) has shape \((s, \mu_0, a, o) \cdot \text{read} \cdot \bullet \cdot \delta_a \rightarrow \bullet \cdot \mu_0 (\alpha) = \beta \in A\) for some decomposition \(\alpha_i = \alpha_i \beta\). This means that the received handle’s extension along a \(\alpha\)-sequence of association paths is a path of mode \(\beta\), which cannot be extended further to a read path.

- If \(\mu_i\) has a correlation \(\beta = \alpha_i\) to a free or rep mode, then \(\pi_i'\) and \(\pi_i\) are extended by dummy edges \(h_i = \omega_i \beta_{in} - \omega_{i+1}\) and \(h_i' = q_i \alpha_i \beta_{in} - q_i \alpha_i \beta\) to the ownership path-pair \(\langle \pi_i', \pi_i \rangle = \omega_i \beta \cdot s \cdot q_i \alpha_i \beta\) for the ownership path-pair \(\langle \pi_i', \pi_i \rangle = \omega_i \beta \cdot s \cdot q_i \alpha_i \beta\). Additionally, \(\equiv\)-path \(q_i - \delta_0 - \gamma_{in} - \omega_{i+1}\) via \(\Pi_i\) implies \(q_i \alpha_i \beta = \omega_{i+1},\beta\) via \(\Pi_i\). Hence we have \(\omega_i, \beta = \omega_{i+1}, \beta\) via \(\Pi_i \cup \{\pi_i', \pi_i\}\).

Observe that the bridge’s path-base \(\Pi\) contained \(\pi_i', \pi_i\) and \(\Pi_i\).

If there are consecutive modes \(\mu_i \ldots \mu_j\) with \(\mu_k (\beta) \in A_i\) they together imply \(\omega_i, \beta = \omega_j, \beta\) via \(\Pi_i \cup \{\pi_i', \pi_i\} \ldots \cup \Pi_j \cup \{\pi_j', \pi_j\}\). It, and dummy edge \(h_{j+1} = \omega_{j+1} - \beta_{in} - \omega_{j+1}, \beta\) if \(j < n\) or extension \(\equiv\)-path \(q_i = \omega_i \beta_{in} - \omega_{i+1}\) if \(j = n\) imply \(\omega_{j+1} = \omega_j, \beta = \omega_j, \beta\) if \(j < n\) or extension \(\equiv\)-path \(q_i = \omega_i \beta_{in} - \omega_{i+1}\) if \(j = n\) imply \(\omega_j, \beta = \omega_j, \beta\) or \(\omega\). It extends the \(\equiv\)-path \(q_j = \delta_{\alpha_j} - \gamma_{in} - \omega_j\) via \(\Pi_j\) to \(q_j - \delta_{\alpha_j} - \gamma_{in} - \omega_{j+1}, \beta\) or \(\omega\), 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 \(\omega\) (if \(j = n\)) or the following extended ownership path-pair (if \(j < n\)) to the right in the \(\alpha_0, \beta\)-bridge from \(q_0\) to \(\omega\). Its path-base is \(\Pi_i \cup \Pi_i \cup \{\pi_i', \pi_i\} \ldots \cup \Pi_j \cup \{\pi_j', \pi_j\} \cup \Pi_{j+1} \) with \(\Pi_{j+1} = \{h_{j+1}\}\) or \(\Pi_i\) respectively.

All in all, we have a \(\alpha_0, \beta\)-bridge from \(q_0\) to \(\omega\) whose path-base \(\Pi'\) contains besides \(\Pi\) some dummy edges \(h_i\) and \(h_i'\), all paths from \(\Pi\) either directly or in extended form: the \(\Pi_i\) of the \(\alpha_0\)-bridge’s \(\equiv\)-paths, the ownership paths \(\pi_i'\) and \(\pi_i\) with \(\mu_i, \alpha \notin \{\text{free, rep}\}\) as well as those with \(\mu_i, \alpha \in \{\text{free, rep}\}\). \(\blacksquare\)

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 \( \rightsquigarrow \)-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 \( \rightsquigarrow \)-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 \( h_o = r \cdot h_o \cdot o \) to a fresh object: \( g' = g \oplus r \cdot h_o \cdot o \).

Define the substitution \( \sigma(q) = \begin{cases} r \cdot \mu_o(a) \cdot a' & \text{if } q = o.\alpha.\alpha' \text{ with } \mu_o(a) \in A \_ \\ q & \text{otherwise} \end{cases} \)

a) In \( g'^{\circ} \), there are three kinds of non-trivial \( \rightsquigarrow \)-paths \( \varphi = o \rightsquigarrow \gamma \rightarrow \omega \) via \( \Pi \):

- \( \varphi \) has a counterpart \( \sigma(o) - \gamma \rightarrow \omega \) via \( \Pi' \) in \( g^{\circ} \) and neither \( o \) nor \( \omega \) are region objects \( o.\beta_1 \ldots \beta_k \) of \( o \) such that for all \( i \leq k \), \( \mu_o(\beta_1 \ldots \beta_i) \notin A \_ \).
- \( \varphi \) is a \( \rightsquigarrow \)-path \( o = o.\beta \rightarrow \gamma \rightarrow \omega \) via dummy edges from one region object \( o \) to another such that for all prefixes \( \hat{\beta} \) of \( \beta \) we have \( \mu_o(\hat{\beta}) \notin A \_ \). That is, \( \Pi \subseteq \Pi' \), which is defined below.
- \( \varphi \) is the combination \( o.\beta \rightarrow \gamma \rightarrow \omega \) with \( \gamma = \gamma_1.\gamma_2 = \gamma_1 \), of an (unchangeable) \( \rightsquigarrow \)-path of the second kind and a \( \rightsquigarrow \)-path of the first kind, with \( \mu_o(\beta, \gamma_1) \in A \_ \).

b) In \( g'^{\circ} \), all new region-couplings \( \omega.\gamma = o.\alpha \) via \( \Pi \) have a counterpart \( \sigma(\omega, \gamma) = \sigma(o.\alpha) \) via \( \Pi' \) in \( g^{\circ} \). If \( \omega.\gamma = o.\beta_1 \ldots \beta_n \) or \( o.\alpha = o.\beta_1 \ldots \beta_n \) then there is an \( i \leq n \) such that \( \mu_o(\beta_1 \ldots \beta_i) \in A \_ \).

The path-base \( \Pi \) of first-kind \( \rightsquigarrow \)-paths and of new region-couplings contains, apart from initially new ownership paths \( \pi \in \Pi_N \) and apart from dummy edges \( h' \in \Pi_G \) of \( o \) and of its region objects \( o.\alpha \) on the way to \( \mu_o(\alpha.\alpha.\alpha') \in A \_ \) only edges with counterparts in their counterpart's path-base \( \Pi' \), and contains them all: \( \sigma(\Pi \setminus \Pi_N) \subseteq \Pi' \), where \( \Pi_N = \{ r \cdot h_o \cdot o.\alpha \mid \mu_o(\alpha) \in \{\text{free, rep}\} \wedge \exists \alpha', \mu_o(\alpha.\alpha') \in A \_ \} \) and \( \Pi_G = \{ o.\alpha \mapsto o.\alpha.\alpha \mid \mu_o(\alpha) \notin A \_ \wedge \exists \alpha', \mu_o(a.\alpha.\alpha') \in A \_ \} \).

**Proof by simultaneous induction on the definition of \( \rightsquigarrow \)-paths and region-couplings:** Lemma 6 guarantees that all potential access paths in \( g'^{\circ} \) are unchanged, initially new, or internally new. Notice that the witnesses \( \sigma(\pi) \) of initially new paths from \( o \) to \( \omega \) go from \( \sigma(o) \) to \( \sigma(\omega) \). The base case, a non-trivial \( \rightsquigarrow \)-path \( o \rightsquigarrow \gamma \rightarrow \omega \) via \( \{\pi\} \) is based on a path \( \pi \in \text{PAP}_{g^{\circ}}(o, \beta<, \omega) \), and a new region-coupling \( \omega.\beta \Rightarrow o.\beta(\beta) \) via \( \{\pi\} \) is established by an ownership path \( \pi \in \text{PAP}_{g^{\circ}}(o, \mu(\beta)) \) with \( \mu(\beta) \in A \_ \).

- If \( \pi \) is initially new or internally new, then \( o \neq o.\beta \) and \( \omega = o.\alpha.\alpha' \) with \( \mu_o(\alpha) \in A \). Hence \( \sigma(o) = o \) and \( \sigma(\omega) = \sigma(o.\alpha.\alpha') = r.\mu_o(\alpha).\alpha' \). If \( \pi \) has a witness \( \sigma(\pi) \), it connects \( \sigma(o) \) and \( \sigma(\omega) \). Hence in case of \( \rightsquigarrow \)-paths, \( \sigma(\pi) \) is the necessary counterpart \( \sigma(o) - \gamma \rightarrow \omega \) via \( \{\sigma(\pi)\} \) for a first-kind \( \rightsquigarrow \)-path. And in case of new
region-coupling and internally new \( \pi, \sigma(\pi) \) is an ownership path that established
\( \mathbf{r} \mu_\alpha(\vec{\alpha}) . \vec{\alpha}' . \beta = \sigma(\omega) . \beta = \sigma(\omega . \beta) \Rightarrow o . \mu(\beta) = \sigma(o) . \mu(\beta) = \sigma(o . \mu(\beta)) \) via \( \{ \sigma(\pi) \} \). If \( \pi \) is an initially new ownership path \( \pi_\beta \) then \( o = r, \omega = o . \vec{\beta}, \) and \( \mu(\beta) = \mu_\alpha(\vec{\beta} . \beta) \).

Hence trivial \( \sigma(o \vec{\beta} . \beta) = \mathbf{r} \mu_\alpha(\vec{\beta} . \beta) = \mathbf{r} . \mu(\beta) \) via \( \Pi' = \emptyset \) is the witness for \( o . \vec{\beta} . \beta \Rightarrow \mathbf{r} . \mu(\beta) \) via \( \Pi = \{ \pi_\beta \} \). Since \( \pi_\beta \in \Pi_N, \sigma(\Pi \setminus \Pi_N) = \emptyset \equiv \Pi' = \emptyset \).

- If \( \pi \) is unchanged with source \( o \neq o . \vec{\beta} \) then it cannot target any object \( o . \vec{\beta} \). Hence \( \sigma(o) = o . \sigma(\omega) = o . \omega = \sigma(\omega . \gamma) = o . \vec{\alpha} \) and \( \sigma(\omega . \vec{\alpha}) = o . \vec{\alpha} \) via \( \{ \sigma(\pi) \} \) for a

first-kind \( \equiv \)-path \( \Pi = \{ \pi \} \). And in case of new region-coupling, \( \sigma(\pi) = \pi \) established the counterpart \( \sigma(\omega . \beta) = \omega . \beta = o . \mu(\beta) = \sigma(o . \mu(\beta)) \) via \( \Pi' = \{ \sigma(\pi) \} \).

- If \( \pi \) is unchanged with source \( o = o . \vec{\beta} \), then it can only be a dummy edge \( o . \vec{\beta} . \beta \).

\( o . \vec{\beta} . \beta \). Hence the region-coupling case does not apply. In case of \( \vec{\beta} = \vec{\alpha} . \vec{\alpha}' \) with \( \mu_\alpha(\vec{\alpha}) \in A_\perp \), the dummy edge \( \sigma(\pi) = \mathbf{r} . \mu_\alpha(\vec{\alpha}) . \vec{\alpha}' \vec{\beta} . \beta \) is the necessary counterpart \( \sigma(o) \vec{\beta} . \beta \sigma(\omega) \) via \( \Pi' = \{ \sigma(\pi) \} \) for a first-kind \( \equiv \)-path \( \Pi = \{ \pi \} \). If there is no \( \mu_\alpha(\vec{\alpha}) \in A_\perp \) then \( \varphi \) is a second-kind, unchanged \( \equiv \)-path.

In the induction step for part (a), consider first the case of \( o \vec{\alpha} . \varphi \omega \) via \( \Pi = \Pi_1 \cup \Pi_2 \) because of \( \vec{\gamma} = \vec{\gamma}_1 \cdot \vec{\gamma}_2 \) and two \( \equiv \)-paths \( o \vec{\alpha} . \varphi \omega \) via \( \Pi_1 \) and \( \Pi_2 \), respectively. If the second \( \equiv \)-path is of the second or third kind, then \( q = o . \vec{\alpha}_1 \ldots \vec{\alpha}_k \) and the first \( \equiv \)-path can only be of the second kind. Consequently, \( o \vec{\alpha} . \varphi \omega \) is of, respectively, second or third kind. If \( q \vec{\alpha} . \varphi \omega \) is of the first kind and \( o \vec{\alpha} . \varphi \omega \) too, then the combination \( \sigma(o) \vec{\alpha} . \varphi \omega \sigma(q) = \vec{\alpha} . \varphi \omega \) of their counterparts via \( \Pi_1 \) and \( \Pi_2 \), respectively, is a counterpart \( o \vec{\alpha} . \varphi \omega \) via \( \Pi' = \Pi'_1 \cup \Pi'_2 \). Since \( \sigma(\Pi_1 \setminus \Pi_N \setminus \Pi_\Theta) \equiv \Pi'_1 \) and \( \sigma(\Pi_2 \setminus \Pi_N \setminus \Pi_\Theta) \equiv \Pi'_2 \), also \( \sigma(\Pi \setminus \Pi_N \setminus \Pi_\Theta) = \sigma(\Pi_1 \cup \Pi_2 \setminus \Pi_N \setminus \Pi_\Theta) \equiv \Pi'_1 \cup \Pi'_2 = \Pi' \). If \( o \vec{\alpha} . \varphi \omega \) \( q \) is of the second kind, then the combination \( o \vec{\alpha} . \varphi \omega \sigma(q) = \vec{\alpha} . \varphi \omega \) to \( o \vec{\alpha} . \varphi \omega \) is obviously of the third kind. If \( o \vec{\alpha} . \varphi \omega \) \( q \) is of the third kind, then it decomposes into a second-kind \( o \vec{\alpha} . \varphi \omega \), \( q' \) and a first-kind \( q' \vec{\alpha} . \varphi \omega \) \( q \). Hence \( o \vec{\alpha} . \varphi \omega \) \( q \) can be decomposed into a second-kind \( o \vec{\alpha} . \varphi \omega \), \( q' \) and a first-kind \( q' \vec{\alpha} . \varphi \omega \) \( q \). It is a third-kind \( \equiv \)-path again.

If \( o \vec{\alpha} . \varphi \omega \) \( \omega \) via \( \Pi = \Pi_1 \cup \Pi_2 \) because of \( \vec{\gamma}_1 \equiv q . \vec{\gamma}_1 \) via \( \Pi_1 \) and \( \varphi_2 = q . \vec{\alpha} . \varphi \omega \) \( \omega \) via \( \Pi_2 \) then the induction hypothesis guarantees a counterpart \( \sigma(o . \vec{\gamma}) \equiv \sigma(q . \vec{\gamma}_1) \) via \( \Pi'_1 \) and that \( q . \vec{\alpha} \varphi \omega \) \( \omega \) belongs to one of three cases:

- If \( \varphi_2 \) has a counterpart \( \sigma(q) \vec{\alpha} . \varphi \omega \sigma(\omega) \) via \( \Pi'_2 \) then either \( \sigma(q) = q \), and thus \( \sigma(q . \vec{\gamma}) = q . \vec{\gamma}_1 \), or \( q = o . \vec{\alpha} \vec{\alpha}' \), and thus \( \sigma(q) = o . \mu_\alpha(\vec{\alpha}) . \omega = \sigma(o . \mu_\alpha(\vec{\alpha})) \vec{\alpha} . \vec{\alpha}' \vec{\gamma}_1 \). In both cases, counterparts \( \sigma(q) \vec{\alpha} . \varphi \omega \sigma(\omega) \) and \( \sigma(q . \vec{\gamma}) \equiv \sigma(q . \vec{\gamma}_1) \) combine to a first-kind \( \equiv \)-path’s counterpart \( \sigma(o) \vec{\alpha} . \varphi \omega \sigma(\omega) \) via \( \Pi' = \Pi'_1 \cup \Pi'_2 \) with \( \sigma(\Pi \setminus \Pi_N \setminus \Pi_\Theta) \equiv \Pi' \).

- \( \varphi_2 \) can be a \( \equiv \)-path \( q = o . \vec{\beta} \vec{\phi} . \varphi \omega \) \( o . \vec{\beta} \vec{\phi} \varphi \omega \) \( \omega \) via \( \Pi_2 \subseteq \Pi_\Theta \). On one hand, \( \mu_\alpha(\vec{\alpha}) \notin A_\perp \) for all proper prefixes \( \vec{\beta} \) of \( \vec{\beta} \). On the other hand, for some prefix
\[ \text{In the induction step of part (b), the symmetry case of new region-couplings is trivial.}\]

\[ \text{Therefore } \sigma(q,q') = \sigma(w) = r \mu_0(\beta'). \text{ Hence the region-coupling's counterpart } \sigma(o,q') = \sigma(q,w') = r \mu_0(\beta) \text{ and trivial } r \mu_0(\beta) - \gamma_\omega = r \mu_0(\beta) \text{ via } \emptyset \text{ combine to a first-kind } \rightsquigarrow \text{-path's counterpart } \sigma(o) - \gamma_\omega \text{ via } \Pi' = \Pi'_1 \cup \emptyset. \]

\[ \text{Since } \Pi_2 \subseteq \Pi_\emptyset, \Pi \setminus \Pi_N \setminus \Pi_\emptyset = \Pi_1, \text{ so that } \sigma(\Pi_1 \setminus \Pi_N \setminus \Pi_\emptyset) \equiv \Pi'_1 \text{ means } \sigma(\Pi \setminus \Pi_N \setminus \Pi_\emptyset) \equiv \Pi'. \]

\[ \varphi_2 \text{ can be the combination of a second-kind } \rightsquigarrow \text{-path } \varphi_3 = o, \beta' - \gamma_\omega \ominus o, \beta, q' \text{ via } \Pi_3 \subseteq \Pi_\emptyset \text{ and a first-kind } \rightsquigarrow \text{-path } \varphi_4 = o, \beta' - \gamma_\omega - \gamma_\omega \omega \text{ with } q = o, \beta \text{ such that } \gamma = \gamma', \gamma'' \text{ and } \mu_0(\beta, \gamma') \in A, \text{ and with } \Pi_2 = \Pi_3 \cup \Pi_4. \]

\[ \text{Hence the region-coupling's counterpart } \sigma(o,q') = \sigma(q,w') = r \mu_0(\beta) \text{ and } \varphi_3 \text{ 's counterpart } \sigma(q,w') = \gamma_\omega \sigma(w) \text{ via } \Pi'_3 \text{ combine to a first-kind } \rightsquigarrow \text{-path's counterpart } \sigma(o) - \gamma_\omega \sigma(w) \text{ via } \Pi'_1 \cup \Pi'_3. \]

\[ \text{Since } \Pi_3 \subseteq \Pi_\emptyset, \Pi \setminus \Pi_N \setminus \Pi_\emptyset = \Pi_1 \cup \Pi_4, \text{ so that } \sigma(\Pi_1 \setminus \Pi_N \setminus \Pi_\emptyset) \equiv \Pi'_1 \text{ and } \sigma(\Pi_1 \setminus \Pi_N \setminus \Pi_\emptyset) \equiv \Pi'_4 \text{ means } \sigma(\Pi \setminus \Pi_N \setminus \Pi_\emptyset) \equiv \Pi'. \]

In the induction step of part (b), the symmetry case of new region-couplings is trivial.

In another case, \( o, \alpha \vdash \omega, q' \text{ via } \Pi = \Pi_1 \cup \Pi_2 \text{ because } o, \alpha \vdash \omega, q' \text{ via } \Pi_1 \text{ and } q, q' \equiv \omega, q' \text{ via } \Pi_2. \text{ If both are new, then the induction hypothesis guarantees counterparts } \sigma(o, \alpha) \equiv \sigma(q, q') \text{ via } \Pi'_1, \text{ and } \sigma(q, q') \equiv \sigma(\omega, q') \text{ via } \Pi'_2, \text{ which combine to counterpart } \sigma(o, \alpha) \equiv \sigma(\omega, q') \text{ via } \Pi'_3 = \Pi'_1 \cup \Pi'_2 \text{ with } \sigma(\Pi_1 \setminus \Pi_N \setminus \Pi_\emptyset) \equiv \Pi'. \]

\[ \text{The same works with an unchanged first region-coupling if } \sigma(o, \alpha) = o, \alpha \text{ and } \sigma(q, q') = q, q', \text{ and works with an unchanged second region-coupling if } \sigma(q, q') = q, q' \text{ and } \sigma(\omega, q') = \omega, q. \text{ Consider the case that the first region-coupling is new and the second is old with } q, q' = o, \beta \text{ or } \omega, q' = o, \beta \text{ (the reverse case follows by symmetry). Since there are no old ownership paths to fresh } o \text{ and its region objects, an old region-coupling connecting their regions can only be a trivial region-coupling } q, q' = (o, \alpha_1 \ldots \alpha_n, \alpha_{n+1} \ldots \alpha_n) \equiv (o, \alpha_1 \ldots \alpha_2, \alpha_{2+1} \ldots \alpha_n = \omega, q' \text{ via } \Pi_2 \text{ obtained from dummy association paths } o, \alpha \equiv \gamma_\omega \alpha, \gamma_1, o, \alpha, \gamma_1 \equiv \alpha, o, \alpha, \gamma_1, \ldots \}. \]

\[ \text{But then } \sigma(q, q') = \sigma(\omega, q') \text{ and } \omega \text{ new, then the new region-coupling's counterpart guaranteed by the induction hypothesis is the desired } \sigma(o, \alpha) \equiv \sigma(q, q') = \sigma(\omega, q') \text{ via } \Pi'_3 = \Pi'_1. \text{ Since } \Pi_2 \subseteq \Pi_\emptyset, \sigma(\Pi \setminus \Pi_N \setminus \Pi_\emptyset) = \sigma(\Pi_1 \setminus \Pi_N \setminus \Pi_\emptyset) \equiv \Pi'_1 \equiv \Pi'. \]

\[ \text{If } o, \alpha, \gamma \vdash \omega, q' \text{ via } \Pi, \text{ follows from } \varphi = o - \gamma_\omega \omega \text{ via } \Pi \text{ then the induction hypothesis guarantees that } \varphi \text{ belongs to one of three cases:} \]

\[ \text{• If } \varphi \text{ has a counterpart } \sigma(o) - \gamma_\omega \sigma(w) \text{ via } \Pi'. \text{ It entails the necessary counterpart } \sigma(o, \alpha, \gamma) \equiv \sigma(o, \alpha, q') \equiv \sigma(\omega, q') \text{ via } \Pi'. \]

\[ \text{• } \varphi \text{ can be a } \rightsquigarrow \text{-path } o = o, \beta - \gamma_\omega \ominus o, \beta, q' = \omega \text{ based on dummy edges in } \Pi_\emptyset. \]

\[ \text{Then it, and consequently } o, \alpha, \gamma \vdash \omega, q' \text{ is not new.} \]

\[ \text{• } o - \gamma_\omega \omega \text{ can be combined from a second-kind } \rightsquigarrow \text{-path } \varphi_2 = o, \alpha_1 - \alpha_2 \gamma_\omega \omega \text{ via } \Pi_2 \text{ with } \alpha = \alpha_1, \alpha_2, \text{ and } o = o, \beta \text{ such that } \mu_0(\beta, \alpha_1) \in A. \text{ In this case, } \varphi_2 \text{ 's counterpart } \sigma(o, \alpha_1) - \gamma_\omega \omega \sigma(w) \text{ established the right region-coupling } \sigma(o, \alpha_1, \alpha_2, \gamma) = \sigma(o, \alpha_1, \alpha_2, \gamma) = \sigma(o, \alpha, \gamma) \equiv \sigma(\omega, q') \text{ via } \Pi' = \Pi'_2 \text{ for new } o, \alpha, \gamma \vdash \omega, q' \text{ via } \Pi. \text{ Since } \Pi_1 \subseteq \Pi_\emptyset, \sigma(\Pi_1 \setminus \Pi_N \setminus \Pi_\emptyset) = \sigma(\Pi_2 \setminus \Pi_N \setminus \Pi_\emptyset) \equiv \Pi'_2 = \Pi'.} \]

159
Lemma 16 Consider elementary mode conversion $g' = g \oplus c \mu_o$, $o \ominus c \mu'_o$; i.e., the substitution of an edge $h'_o = c \mu'_o$, $o$ for an edge $h_o = c \mu_o$, $o$ with $\mu \leq \mu'$. In $g'^\otimes$,

- all $\Rightarrow$-paths $o \rightarrow \gamma \omega$ via $\Pi$ have a precursor $o \rightarrow \gamma \omega$ via $\Pi[h_o/h'_o]$, and
- all region-couplings $\omega, \gamma \Rightarrow o, \alpha$ via $\Pi$ have a precursor $\omega, \gamma \Rightarrow o, \alpha$ via $\Pi[h_o/h'_o]$.

Proof: Lemma 7 guarantees that all potential access paths $\pi \in PAP_{g^\otimes}(o, \mu, \omega)$ in $g'^\otimes$ are unchanged, internally new, or initially new. They all have a precursor $\pi' = \pi[h_o/h'_o] \in PAP_{g^\otimes}(o, \mu', \omega)$ of the same mode $\mu' = \mu$ or of a directly compatible mode $\mu' \leq \mu$. By the definition of $\leq \mu$, a mode $\mu \geq \mu'$ cannot be an association mode. Hence all association paths $\pi$ have a precursor $\pi'$ of the same mode. The base case of new $\Rightarrow$-paths $\varphi$ differs at most in the path-base: $\varphi$ is now via $\{\pi\}$ instead of via $\{\pi'\}$. Also by the definition of $\leq \mu$, a mode $\mu \geq \mu'$ can be a free or rep mode only in case of $\mu' = \text{free} \delta < \mu \leq \mu$. Hence all ownership paths $\pi$ have a precursor $\pi'$ that was already an ownership path and had the same correlations $\delta$. Hence the base case of new $\Rightarrow$-path-couplings differs at most in the path-base: $o, \gamma \Rightarrow \omega, \gamma$ is via $\pi$ now instead of via $\pi'$. By induction therefore all $\Rightarrow$-paths and region-coupling via $\pi$ have a precursor via $\Pi[h_o/h'_o]$.

Lemma 17 Consider result return $g' = g \ominus r \mu_o \ominus s \mu_r \ominus s \mu_{o,r} \ominus o$, i.e., the substitution of the imported edge $h'_o = s \mu_{o,r} \ominus o$ for the exported edge $h_o = r \mu_o \ominus o$ and the call-link $h_r = s \mu_r \ominus r$. In $g'^\otimes$,

- all $\Rightarrow$-paths $o \rightarrow \gamma \omega$ via $\Pi$ have a precursor $o \rightarrow \gamma \omega$ via $\sigma(\Pi)$, and
- all region-couplings $\omega, \gamma \Rightarrow o, \alpha$ via $\Pi$ have a precursor $\omega, \gamma \Rightarrow o, \alpha$ via $\sigma(\Pi)$,

where substitution $\sigma$ is $[h_r \cdot h_o/h'_o, h_o^{-1} \cdot h_r^{-1}/h_o^{-1}]$ in case of $\mu_r \cdot \mu_o = \text{co}<>$, and $[h_r \cdot h_o/h'_o]$ otherwise.

Proof: Lemma 8 guarantees that all potential access paths in $g'^\otimes$ are unchanged, internally new or initially new. If these are association paths $\pi \in PAP_{g^\otimes}(o, \beta<> \omega)$, they have a precursor $\pi' = \sigma(\pi) \in PAP_{g^\otimes}(o, \beta<> \omega)$. Hence the base case of new $\Rightarrow$-paths $\varphi$ differs at most in the path-base: $\varphi$ is now via $\{\pi\}$ instead of via $\{\sigma(\pi)\}$. Similarly most ownership paths $\pi$ that establish a region-coupling via $\{\pi\}$ have a precursor $\sigma(\pi)$ that established it already in $g^\otimes$ via $\{\sigma(\pi)\}$. The base case of a really new region-coupling could only come from an initially new ownership path $\pi \in PAP_{g^\otimes}(o, \mu, \omega)$ without precursor $\sigma(\pi)$, i.e., whose counterpart $\text{cp}(\pi)$'s mode $\mu'$ with $\mu = \mu_r \cdot \mu_o$ is not a co- or association mode, but a free mode. Then $\pi$ is a free path which establishes the region-coupling $\omega, \gamma \Rightarrow s, \mu(\gamma)$ in $g'^\otimes$ in two cases:

- $\mu(\gamma) = \alpha$, i.e., $\mu$ has a correlation $\gamma = \alpha<>$. Since $\pi$'s mode $\mu$ is the adaption $\mu_r \cdot \mu_o'$ of its counterpart's non-co and non-association mode $\mu'$, it can contain the correlation $\gamma = \alpha<>$ only because $\text{cp}(\pi)$'s mode $\mu'$ contains $\gamma = \beta<>$ and $\mu_r$ contains $\beta = \alpha<$. Correlations in $\mu'$ mean that it cannot be a co- or association mode, and thus $\mu_r \cdot \mu_o'$ can be a free or rep mode only if $\mu'$ is a free or rep mode. That
is, \( \mu' = \text{free} \ldots, \gamma = \beta \ldots \rightarrow \) or \( \text{rep} \ldots, \gamma = \beta \ldots \rightarrow \). Hence, on one hand, \( \text{free} \) counterpart \( \text{epr}(\pi) \) established \( \omega, \gamma \Rightarrow r, \beta \) via \( \{\text{epr}(\pi)\} \) in \( g^\circ \). On the other hand, a legal return step means that in this case \( \mu_r \) is a \( \text{free} \) or \( \text{rep} \) mode. That is, \( \mu_r = \text{free} \ldots, \beta = \alpha \ldots \rightarrow \) or \( \text{rep} \ldots, \beta = \alpha \ldots \rightarrow \). Hence \( h_r \) established \( r, \beta \Rightarrow s, \alpha \) via \( \{h_r\} \) in \( g^\circ \). In combination, we have \( \omega, \gamma \Rightarrow s, \mu(\gamma) \Rightarrow s, \alpha \) via \( \Pi' = \{h_r, \text{epr}(\pi)\} \).

\( \bullet \) \( \mu(\gamma) = \bot \), i.e., \( \mu \) has no correlation \( \gamma = \mu'' \). Since \( \pi \)'s mode \( \mu \) is the adaption \( \mu_r \circ \mu' \) of its counterpart's non-co and non-association mode \( \mu' \), it can contain no correlation \( \gamma = \mu'' \) only if \( \text{epr}(\pi)' \)'s mode \( \mu' \) contains no correlation \( \gamma = \mu'' \). Consequently, \( \text{free} \) counterpart \( \text{epr}(\pi) \) established \( \omega, \gamma \Rightarrow r, \bot \) via \( \{\text{epr}(\pi)\} \) in \( g^\circ \). But since also \( r, \bot \Rightarrow s, \bot \) via \( \emptyset \), this means by transitivity \( \omega, \gamma \Rightarrow s, \bot \Rightarrow s, \mu(\gamma) \) via \( \Pi' = \{\text{epr}(\pi)\} \).

Hence there is a precursor for \( \omega, \gamma \Rightarrow s, \bot \Rightarrow s, \mu(\gamma) \) via \( \Pi' = \{\pi\} \) such that \( \Pi' \in \sigma(\Pi) \) since \( h_r \cdot \text{epr}(\pi) = \sigma(\pi) \).

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 \( \S 6.3.1 \), \( \alpha \)-bridges were used for the description of the new potential access paths. They consisted of an initial \( \Leftarrow \)-path \( o \xrightarrow{-\alpha} -\gamma \omega_0 \) followed by a series of triples \( \langle \pi_i, \pi'_i, \varphi'_i \rangle \) made of a pair of ownership paths \( \pi_i \) and \( \pi'_i \) and another \( \Rightarrow \)-path \( \varphi'_i \). The description of the new \( \Leftarrow \)-paths will require a generalization to bridges where an initial \( \Leftarrow \)-path \( o \xrightarrow{-\alpha} -\gamma \omega_0 \) is followed by a series of quadruples \( \langle \varphi'_i, \pi'_i, \pi_i, \varphi_i \rangle \), which will be called a "\( \alpha \)-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 \( \omega \) 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 \( \pi'_i \Rightarrow s, -\beta \Rightarrow q_i, \alpha_i, \) and \( \varphi'_i = q'_i \xrightarrow{-\alpha} -\gamma \omega_i \), where \( \pi_i = s \xrightarrow{\beta} q_i, \alpha_i \), and \( \varphi_i = q_i \xrightarrow{-\gamma} -\gamma \omega_i \omega_{i+1} \) such that \( \omega_1 = o \) and \( \omega_{n+1} = \omega \). The qbridge is via \( \bigcup_{i=1}^{n+1} \Pi_i \cup \Pi'_i \cup \{\pi_i, \pi'_i\} \) if the \( \varphi_i \) is via \( \Pi_i \) and the \( \varphi'_i \) is via \( \Pi'_i \).

An \( \alpha \)-qbridge from \( o \) to \( \omega \) is an initial \( \Leftarrow \)-path \( o \xrightarrow{-\alpha} -\gamma \omega_1 \) via \( \Pi_0 \) with \( \omega_1 = o \) or a reservation bridge from \( \omega_1 \) to \( \omega \) via \( \Pi' \). from \( i = 1 \) to \( n \geq 0 \), such that \( \omega_{n+1} = \omega \). The qbridge is, respectively, via \( \Pi_0 \) or via \( \Pi_0 \cup \Pi' \).

In case of \( \mu_o = \text{co} < \), for each quadruple in both kinds of qbridges there is an \( \alpha \) such that each of the two ownership paths \( \pi_i \) and \( \pi'_i \) in it has shape \( s \xrightarrow{\mu o \mu_o} o \xrightarrow{co} \bullet \xrightarrow{-\alpha} \bullet \) or \( s \xrightarrow{\mu r \mu_o} r \xrightarrow{co} \bullet \xrightarrow{-\alpha} \bullet \). If \( \mu_o \neq \text{co} < \), one of the two ownership paths \( \pi_i \) and \( \pi'_i \) in each quadruple in the qbridges has shape \( s \xrightarrow{\mu o \mu_o} o \xrightarrow{co} \bullet \xrightarrow{-\alpha} \bullet \) for some \( \alpha \) and \( \alpha' \) with \( \mu_o(\alpha) \in A \), while the other has shape \( s \xrightarrow{\mu_r \mu_o} r \xrightarrow{co} \bullet \xrightarrow{\mu o (\alpha') \gamma } \bullet \).

**Lemma 18** Assume the reserved ownership assumption in \( g^\circ \) and \( q = \text{ogr}(e, \eta, s) \). Consider parameter supply \( g' = g \odot r \xrightarrow{\mu o} o \odot s \xrightarrow{\mu o \mu_o} o \), i.e., the substitution of a received
handle \( h_0 = r \). For a sent handle \( h'_0 = s \) in the presence of a call-link \( h_r = s \).

a) For each non-trivial \( \implies \)-path \( o \rightarrow \gamma \rightarrow \omega \) via \( \Pi \) in \( g^\circ \), there was a \( \gamma \)-qbridge from \( o \) to \( \omega \) in \( g^\circ \).

b) For each new region-coupling \( o.\alpha \implies \omega.\gamma \) via \( \Pi \) in \( g^\circ \), there was a reservation qbridge from \( o.\alpha \) to \( \omega.\gamma \) in \( g^\circ \).

Both qbridges are via \( \Pi' \) such that \( \Pi' \cup \Pi_0 \equiv \Pi \cup \Pi_\Lambda \cup \Pi_\circ \) where \( \Pi_\circ \) is some set of dummy edges, where \( \Pi_0 \subseteq \{ h_0, h_0^{-1} \} \), and where \( \Pi_\Lambda = \{ h_r, h'_0 \} \) if \( \mu_0(\alpha) \in \Lambda \) for some \( \alpha \), and \( \Pi_\Lambda = \{ h_r, h'_0 \} \) or \( \{ h_r, h'_0, h_r^{-1} \} \) or \( \{ h_r, h_r^{-1}, h'_0, h_r^{-1} \} \) if \( \mu_0 = \mu_r = \text{co}<> \), and \( \Pi_\Lambda = \emptyset \) otherwise.

Proof by simultaneous induction on the definition of \( \implies \)-paths and region-couplings: Lemma 9 guarantees that all potential access paths in \( g^\circ \) are unchanged, internally new, initially new, or co-closure paths. In the base case of part (a), an association path \( \pi \in \text{PAP}^g(\alpha, \beta<>\omega) \) established the \( \implies \)-path \( o \rightarrow \gamma \rightarrow \omega \) via \( \Pi = \{ \pi \} \) with \( \gamma = \beta \). And in the base case of part (b), an ownership path \( \pi \in \text{PAP}^g(o, \mu, \omega) \) with \( \mu(\beta) \in \Lambda \) established the new region-coupling \( o.\mu(\beta) \implies \omega.\beta \) via \( \Pi' = \{ \pi' \} \).

(1) If \( \pi \) is an unchanged or internally-only new path then it has a precursor \( \pi' = \pi \) or \( \pi' = [h_r^{-1} \cdot h'_0 / h_0, h'_0 / h_0^{-1} \cdot h_r / h_0^{-1}] \). In part (a), \( \pi' \) is a \( \implies \)-path \( o \rightarrow \gamma \rightarrow \omega \) via \( \Pi = \{ \pi' \} \). In part (b), \( \pi' \) established the region-coupling \( \omega.\beta \implies o.\mu(\beta) \) via \( \Pi' = \{ \pi' \} \). Obviously, \( \Pi' \cup \Pi_0 \equiv \Pi \cup \Pi_\Lambda \) for a subset \( \Pi_\Lambda \) of \( \{ h_r, h_r^{-1}, h'_0, h_0^{-1} \} \) and a subset \( \Pi_0 \) of \( \{ h_0, h_0^{-1} \} \).

(2) If \( \pi \) is internally really new then \( \pi = \pi_1 \cdot \pi_2 \), there is a witness \( \text{wit}(\pi) = \pi_1 \cdot \pi_2' \in \text{PAP}^g(\mu, h_0, \alpha_0) \) and a \( \alpha_0 \)-bridge to \( \omega \) via \( \Pi' \). In part (a), \( \text{wit}(\pi) = q \cdot \beta<>\omega, q_0, \alpha_0 \) and the initial \( \implies \)-path \( q_0 \rightarrow q_0 \cdot \beta<>\omega_0 \) via \( \Pi'_0 \) of the \( \alpha_0 \)-bridge implied the \( \implies \)-path \( o \rightarrow \gamma \rightarrow \omega_0 \) via \( \Pi_0 \cup \{ \text{wit}(\pi) \} \) (Lemma 12). And the triples \( \{ \pi_1', \pi_1, \varphi_1 \} \) of the \( \alpha_0 \)-bridge can be transformed into the quadruples \( \{ \varphi_1', \pi_1', \varphi_1, \varphi_1 \} \) of a reservation qbridge by adding \( \varphi_1' = \omega_{1-\epsilon} \cdot \omega_{1-\epsilon} \cdot \epsilon \) via \( \Omega \) for each \( \pi_1' = q \cdot \beta<>h_{\epsilon-\omega_{1-\epsilon}} \cdot \omega_{1-\epsilon} \). Together we are the desired \( \gamma = \beta \)-qbridge from \( o \) to \( \omega \) via \( \Pi'' = \Pi' \cup \{ \text{wit}(\pi) \} \equiv \Pi' \cup \{ \pi_1, \pi_2 \} \). Since internally really new \( \pi \) guarantees \( \Pi' \cup \Pi_0 \equiv \{ \pi_2 \} \cup \Pi_\Lambda \cup \Pi_\circ \cup \{ \pi_2 \} \) and since \( \pi = \pi_1 \cdot \pi_2 \), we have \( \Pi'' \cup \Pi_0 \equiv \{ \pi \} \cup \Pi_\Lambda \cup \Pi_\circ \cup \{ \pi_2 \} \). That is, \( \Pi'' \cup \Pi_0 \equiv \Pi \cup \Pi_\Lambda \cup (\Pi_\circ \cup \{ \pi_2 \}) \).

In part (b), \( \mu(\beta) = \gamma \in \Lambda \). The \( \alpha_0 \)-bridge is extended along dummy edge \( h_0 = \omega.\beta \). To a \( \alpha_0.\beta \)-bridge from \( q_0 \) to \( \omega.\beta \) via \( \Pi'' \equiv \Pi' \cup \{ h_0 \} \) (Lemma 14). Its initial \( \implies \)-path \( q_0 \rightarrow q_0.\beta<>\omega_0 \) via \( \Pi'_0 \) implies \( (q_0, \alpha_0, \beta).\epsilon = \omega_0.\epsilon \) via \( \Pi'_0 \). The ownership path \( \text{wit}(\pi) = q \cdot \mu<>q_0, \alpha_0 \) implies the matching \( o.\gamma \implies (q_0, \alpha_0, \beta) \) via \( \Pi'_0 \). In the case that the \( \alpha_0.\beta \)-bridge contains no triple series, i.e., \( \omega_0 = \omega.\beta \), these couplings mean \( o.\gamma \equiv (\omega.\beta, \epsilon) \) via \( \Pi'' \cup \{ \text{wit}(\pi) \} = \Pi'' \), \( \{ \text{wit}(\pi) \} \). Otherwise, dummy edge \( h_0 = o \cdot \beta<>o.\gamma \) modulo these couplings is \( \epsilon' = o.\gamma + \epsilon.\gamma \) via \( \Pi'_0 \cup \{ \text{wit}(\pi), h_0 \} \). It extends the first triple \( \{ \pi_1, \pi_1, \varphi_1 \} \) in the \( \alpha_0.\beta \)-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 \( \Pi'' = \Pi'' \cup \{ \text{wit}(\pi), h_0 \} \equiv \Pi'' \cup \{ h_0 \} \).
Since internally really new $\pi$ guarantees $\Pi' \cup \Pi_o \equiv \{n_2\} \cup \Pi_A \cup \Pi_\otimes$ and since $\text{wit}(\pi) = \pi_1 \cdot \pi_2$ and $\pi = \pi_1 \cdot \pi_2$, we have $\Pi'' \cup \Pi_o \equiv \{n_2\} \cup \Pi_A \cup \Pi_\otimes \cup \{h, h_0\} \cup \{\text{wit}(\pi), h_0\} = \{\pi_2\} \cup \Pi_A \cup (\Pi_\otimes \cup \{h_\beta, \bar{h}_\beta\} \cup \{\pi_2\}) \equiv \{\pi_2\} \cup \Pi_A \cup (\Pi_\otimes \cup \{h_\beta, \bar{h}_\beta, \bar{h}_\beta\} \cup \Pi_\beta).

(3) If $\pi$ is initially new, then $o = r$ and there is a witness $\text{wit}(\pi) = h'_\alpha \cdot \pi_1 \cdot \pi_2 = s \in w_0 \cdot \alpha_0$ and a $\alpha_0$-bridge to $\omega$ via $\Pi'$. In part (a), Lemma 13 allows two cases for initially new association path $\pi$:

- There is a $\leftarrow\rightarrow$-path $r \leftarrow h_\beta$ via $\Pi_0 \cup \{h_r, \text{wit}(\pi)\}$ to the object $\omega_0$ from which the series of triples in $\pi$'s $\alpha_0$-bridge leads to $\omega$. This $\leftarrow\rightarrow$-path, and the triple series transformed to quadruples as above, constitute a $\beta$-qbridge from $o = r$ to $\omega$ via $\Pi'' = \Pi' \cup \{h_r, \text{wit}(\pi)\} \equiv \Pi' \cup \{h_r, h'_\alpha \cdot \pi_1 \cdot \pi_2\} \equiv \Pi' \cup \{\pi\} \cup \Pi_A \cup \Pi_\otimes$ with $\Pi_\alpha = \{h_r, h'_\alpha\}$ and $\Pi'_\alpha = \{\pi_2\}$.

- There is an ownership path-pair $(h_r \cdot h_\beta, \text{wit}(\pi)) = r \cdot s \cdot \beta \cdot s \cdot g_0 \cdot \alpha_0$ with $h_\beta = r \cdot \beta$. In front of $\pi$'s $\alpha_0$-bridge to $\omega_0$ to $\omega$. Add the initial $\leftarrow\rightarrow$-path $\varphi_0 = g_0 \cdot \overleftarrow{\alpha_0} \cdot \alpha_0 \cdot \omega_0 \text{ via } \Pi_0$ of the $\alpha_0$-bridge to these two in order to get another triple $(h_r \cdot h_\beta, \text{wit}(\pi), \varphi_0)$, and transform this triple and the $\alpha_0$-bridge's triples to quadruples as above. This gives us a reservation qbridge from $r \cdot \beta$ to $\omega$ via $\Pi'' = \Pi' \cup \{h_r, h_\beta, \text{wit}(\pi)\}$. Add $h_\beta$ as initial $\leftarrow\rightarrow$-path, and we get the desired $\beta$-qbridge from $o = r$ to $\omega$ via $\Pi'' = \Pi' \cup \{h_r, h_\beta, \text{wit}(\pi)\}$.

Since initially new $\pi$ guarantees $\Pi' \cup \{h_o\} \equiv \{n_2\} \cup \Pi_A \cup \Pi_\otimes$, we have $\Pi'' \cup \{h_o\} \equiv \Pi' \cup \{\pi\} \cup \Pi_A \cup \Pi_\otimes$ and $\Pi'_\alpha = \{\pi_2\} \cup \Pi_\otimes$.

In part (b), on one hand, $\pi$'s $\alpha_0$-bridge can be extended along dummy edge $h_\beta = \omega \cdot \beta \cdot \omega$ to a $\alpha_0, \beta$-bridge from $g_0$ 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 $\pi$ with $\mu(\beta) \in A_\perp$:

- Region-coupling $\omega, \mu(\beta) \leftarrow \omega, \beta$ already existed in $g_\otimes$ via $\Pi'' = \Pi' \cup \{\text{wit}(\pi)\} \cup \Pi'_\alpha \equiv \Pi' \cup \{h_\beta, \text{wit}(\pi)\} \cup \Pi_\beta \cup \Pi'_\alpha = \Pi' \cup \{h_\beta, \bar{\pi}'\} \cup \Pi'_\alpha \cup \Pi''$ with $\Pi'' = \{\pi\} \cup \Pi_A \cup \Pi_\otimes$ with $\Pi'_\alpha = \emptyset$ or $\{h_r\}$, with $\Pi''_\alpha = \Pi_\beta \cup \{h_\beta\}$ and with $\Pi'_\alpha = \{h_\beta\}$. And there was an ownership path-pair $(h_r \cdot h_\beta, \text{wit}(\pi) \cdot h'_\beta) = o \cdot \mu(\beta) \leftarrow \cdot \beta \cdot \cdot s \cdot \cdot g_0 \cdot \alpha_0, \beta$. These two are completed to a quadraple by trivial $\varphi' = o \cdot \mu(\beta) \leftarrow \cdot \beta \cdot \cdot s \cdot \cdot \overleftarrow{\alpha_0} \cdot \cdot \cdot \omega_0 \text{ via } \Pi_0$ to the right. It and the $\alpha_0, \beta$-bridge's triples transformed to quadruples as above, constitute the desired reservation qbridge from $o \cdot \mu(\beta)$ to $\omega$, $\beta$ via $\Pi'' = \Pi' \cup \{h_r \cdot h_\beta, \text{wit}(\pi) \cdot h'_\beta\} \equiv \Pi' \cup \{h_\beta, h_r, h'_\beta, h_\beta', \bar{\pi}'\} \cup \Pi_\beta = \Pi' \cup \{\bar{\pi}'\} \cup \Pi'_\alpha \cup \Pi''$ with $\Pi'_\beta = \Pi_\beta \cup \{h_\beta, h'_\beta, h_\beta'\}$ and $\Pi''_\alpha = \{h_r, h'_\beta\}$.

Since initially new $\pi$ guarantees $\Pi' \cup \{\bar{\pi}'\} \equiv \{\pi\} \cup \Pi_A \cup \Pi_\otimes$, we have $\Pi'' \cup \{\bar{\pi}'\} \equiv \Pi' \cup \{\bar{\pi}'\} \cup \Pi'_\alpha \cup \Pi''_\alpha \cup \{h_o\} \equiv \{\pi\} \cup \Pi_A \cup \Pi_\otimes \cup \Pi''_\alpha$.

In the induction step, we can ignore trivial $\leftarrow\rightarrow$-paths $o \leftarrow \omega$ since they produce nothing new.

For part (a), consider first the case of $o \leftarrow \bar{\pi}' \omega \Pi$ because of $\bar{\pi}' = \bar{\pi}_1 \cdot \bar{\pi}_2$ and two $\leftarrow\rightarrow$-paths $o \leftarrow \bar{\pi}_1 \cdot \omega q \leftarrow \bar{\pi}_2 \omega$ via $\Pi_1$ and $\Pi_2$. respectively. The induction
hypothesis guarantees a $\gamma_1$-qbridge from $o$ to $q$ \textit{via} $\Pi_1$, and a $\gamma_2$-qbridge from $q$ to $\omega$ \textit{via} $\Pi_2$. These two qbridges combine to a $\gamma_1, \gamma_2$-qbridge from $o$ to $\omega$ \textit{via} $\Pi' \equiv \Pi'_1 \cup \Pi'_2 \cup \Pi_{\gamma}$ (Lemma 22). This is the desired $\gamma$-qbridge with $\Pi' \cup \Pi_o \equiv \Pi'_1 \cup \Pi'_2 \cup \Pi_{\gamma_2} \cup \Pi_o \equiv (\Pi_1 \cup \Pi_A \cup \Pi'_0) \cup (\Pi_2 \cup \Pi_A \cup \Pi''_0) \cup \Pi_{\gamma} = \Pi \cup (\Pi_A) \cup (\Pi'_0 \cup \Pi''_0 \cup \Pi_{\gamma_2})$.

If $o \rightarrow \omega$ because $o.\alpha = q.\gamma$ \textit{via} $\Pi_1$ and $q \rightarrow \omega$ \textit{via} $\Pi_2$ then the induction hypothesis guarantees a $\gamma$-qbridge from $q$ to $\omega$ \textit{via} $\Pi'_2$, and two cases for the region-coupling:

- Either $o.\alpha = q.\gamma$ is unchanged. Then it and the $\gamma$-qbridge's initial \textit{\Rightarrow}-path $q \rightarrow \omega$ \textit{via} $\Pi'_{2,0}$ imply the \textit{\Rightarrow}-path $o \rightarrow \omega$ \textit{via} $\Pi_1 \cup \Pi'_{2,0}$ (Lemma 12). It, together with the $\gamma$-qbridge's reservation qbridge from $\omega_1$ to $\omega$ constitutes the desired $\alpha$-qbridge from $o$ to $\omega$ \textit{via} $\Pi' = \Pi_1 \cup \Pi_2$.

- Or there is a reservation qbridge from $o.\alpha$ to $q.\gamma$ \textit{via} $\Pi'_{1,0}$. Its final \textit{\Rightarrow}-path $o_n \rightarrow q.\gamma$ \textit{via} $\Pi'_{1,n}$ and the $\gamma$-qbridge's initial \textit{\Rightarrow}-path $q \rightarrow \omega$ \textit{via} $\Pi'_{2,0}$ imply the \textit{\Rightarrow}-path $o_n \rightarrow q.\gamma$ \textit{via} $\Pi'_{1,n} \cup \Pi'_{2,0}$ (Lemma 12). Substituting it for the reservation qbridge's final \textit{\Rightarrow}-path links it with the $\gamma$-qbridge's reservation qbridge to a reservation qbridge from $o.\alpha$ \textit{via} $\Pi'_{1} \cup \Pi_2$. Prefixing it with $\pi_o = o.\alpha$ as a basic \textit{\Rightarrow}-path produces the desired $\alpha$-qbridge from $o$ to $\omega$ \textit{via} $\Pi' = \Pi'_1 \cup \Pi_2 \cup \{\pi_o\}$.

Since $\Pi'_2 \cup \Pi''_0 \equiv \Pi_2 \cup \Pi_A \cup \Pi''_0$, in the former case $\Pi' \cup \Pi''_0 \equiv \Pi_1 \cup (\Pi'_2 \cup \Pi''_0) \equiv \Pi_1 \cup (\Pi_2 \cup \Pi_A \cup \Pi''_0) = \Pi \cup \Pi_{\gamma}$, and in the latter case $\Pi'_1 \cup \Pi''_0 \equiv \Pi_1 \cup (\Pi'_1 \cup \Pi''_0) \equiv \Pi_1 \cup (\Pi_1 \cup (\Pi_2 \cup \Pi_A \cup \Pi''_0) \cup \{\pi_o\}) = \Pi_1 \cup (\Pi_1 \cup \Pi_2 \cup \Pi_A \cup \Pi''_0) \cup \{\pi_o\}$.

In one case of the induction step of part (b), $o.\alpha = \omega.\gamma$ \textit{via} $\Pi$ is new because a reservation qbridge from $\omega.\gamma$ \textit{via} $o.\alpha$ \textit{via} $\Pi'$. Note that reservation qbridges are symmetric in structure: If read in reverse, this reservation qbridge is a reservation qbridge from $o.\alpha$ \textit{via} $\omega.\gamma$.

In another case, $o.\alpha = \omega.\gamma$ because $o.\alpha = q.\tilde{\beta}$ \textit{via} $\Pi_1$ and $q.\tilde{\beta} = \omega.\gamma$ \textit{via} $\Pi_2$. If none of the two region-couplings is new, they combined to $o.\alpha = \omega.\gamma$ \textit{via} $\Pi_1 \cup \Pi_2 = \Pi$ already in $g^\circ$. If both are new, then the induction hypothesis's reservation qbridges from $o.\alpha$ to $q.\beta$, and from $q.\tilde{\beta}$ to $\omega.\gamma$ concatenate to one reservation qbridge from $o.\alpha$ to $\omega.\gamma$ \textit{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 $o.\alpha$ to $q.\beta$ \textit{via} $\Pi'_1$ in $g^\circ$ has a last \textit{\Rightarrow}-path $o_n \rightarrow q.\tilde{\beta} \rightarrow \omega.\gamma$ \textit{via} $\Pi'_{1,n}$. In conjunction with old $q.\tilde{\beta} = \omega.\gamma$ in $g^\circ$, this entails $o_n \rightarrow \omega.\gamma$ \textit{via} $\Pi'_{1,n} \cup \Pi_2$ (Lemma 22). It, together with the rest of the reservation qbridge from $o.\alpha$ to $q.\tilde{\beta}$ is a reservation qbridge from $o.\alpha$ to $\omega.\gamma$ \textit{via} $\Pi' = \Pi'_1 \cup \Pi_2$. The case of unchanged first and new second region-coupling follows by symmetry.

If new $o.\alpha \cdot \gamma = \omega.\gamma$ is implied by new $o \rightarrow \omega$ \textit{via} $\Pi'$ in $g^\circ$. It can be extended
by $\omega \overset{\delta}{\rightarrow} \omega' \overset{\gamma}{\rightarrow}$ (as a $\Rightarrow$-path via $\Pi_\omega = \{ \omega \overset{\mu_1}{\rightarrow} \omega', \ldots, \omega \overset{\mu_{n-1}}{\rightarrow} \omega' \}$) to a $\overline{\alpha} \overset{\delta}{\rightarrow} \gamma$-qbridge from $o$ to $\omega' \overset{\gamma}{\rightarrow}$ via $\Pi'_\omega = \Pi' \cup \Pi_\omega \cup \Pi_\gamma$ (Lemma 22). That qbridge's initial $\Rightarrow$-path $o \overset{\delta}{\rightarrow}^* \omega'_1 \overset{\gamma}{\rightarrow} c$ via $\Pi'_\omega$ means $o \overline{\alpha} \overset{\delta}{\rightarrow} \gamma \overset{\gamma}{\rightarrow} e \rightleftarrows \omega'_1$. Hence dummy edge path $o \overset{\delta}{\rightarrow} \omega'_1 \overset{\gamma}{\rightarrow} \alpha \overset{\delta}{\rightarrow} \gamma$ (as a $\Rightarrow$-path via $\Pi'_{\overline{\alpha} \overset{\delta}{\rightarrow} \gamma} = \{ o \overset{\delta}{\rightarrow} \omega'_1, \ldots, \omega'_1 \}$) implies the $\Rightarrow$-path $o \overset{\gamma}{\rightarrow} \omega'_1 \overset{\gamma}{\rightarrow} \alpha \overset{\delta}{\rightarrow} \gamma$ via $\Pi'_\omega \cup \Pi_{\overline{\alpha} \overset{\delta}{\rightarrow} \gamma}$. It can extend the $\Rightarrow$-path $\varphi'_1 = \delta_1 \overset{\delta}{\rightarrow} \omega'_1 \overset{\gamma}{\rightarrow} \alpha \overset{\delta}{\rightarrow} \gamma$ via $\Pi'_\omega \cup \Pi_{\overline{\alpha} \overset{\delta}{\rightarrow} \gamma} \cup \Pi''_\gamma$. By substituting $\varphi'_1$ for $\varphi'_1$, the $\overline{\alpha} \overset{\delta}{\rightarrow} \gamma$-qbridge's reservation qbridge from $\omega'_1$ to $\omega' \overset{\gamma}{\rightarrow}$ is transformed to the desired reservation qbridge from $o \overset{\gamma}{\rightarrow} \alpha \overset{\delta}{\rightarrow} \gamma$ to $\omega' \overset{\gamma}{\rightarrow}$ via $\Pi'' = \Pi'' \cup \Pi_{\overline{\alpha} \overset{\delta}{\rightarrow} \gamma}$.

By induction hypothesis, $\Pi' \cup \Pi_\omega = \Pi' \cup \Pi_\alpha \cup \Pi_\beta$. Hence $\Pi'' = \Pi'' \cup \Pi_\omega \cup \Pi_\gamma \cup \Pi_{\overline{\alpha} \overset{\delta}{\rightarrow} \gamma} = \Pi' \cup \Pi_\omega \cup \Pi_\gamma \cup \Pi_{\overline{\alpha} \overset{\delta}{\rightarrow} \gamma}$. 

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 $\Gamma_n, \kappa_n \vdash e : \tau$ for the operation call expression redex $e$. They concern the graph $g'$ after addition of a new received handle $h_0 = r \overset{\mu_{\beta}}{\rightarrow} o$ and removal of old sent handle $h_0 = s \overset{\mu_0 \overset{\beta}{\rightarrow} o}$ from the previous graph $g$, i.e., $g' = g \uplus r \overset{\mu_0 \overset{\beta}{\rightarrow} o} \uplus o \overset{\mu_0 \overset{\beta}{\rightarrow} o} \uplus s \overset{\mu_0 \overset{\beta}{\rightarrow} o} \uplus o$.

The first three lemmas are about initially new ownership paths. The final lemma is about qbridges.

**Lemma 19** Assume $g \models \text{UH}$ and $g = \text{ogr}(e, \bar{\gamma}, s)$. 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 $e$ would contain the handle $h$ twice: As value $(s, \mu_r, r)$ of the receiver expression and as value $(s, \mu_r = \mu_r, s)$ of an argument expression. Hence its multiplicity in $g$ should be more than one by $g \models \text{ogr}(e, \bar{\gamma}, s)$. On the other hand, if the sent handle is free, $g \models \text{UH}$ ensures that its multiplicity is one in $g$. A contradiction.

**Lemma 20** Assume the reserved ownership assumption in $g^{\text{eq}}$ and $g = \text{ogr}(e, \bar{\gamma}, s)$. Consider an initially new ownership path $\pi \in \text{PAP}_{\mu_{\beta}}(o, \mu, \omega)$. Its witness $\text{wit}(\pi) = s \overset{\mu_{\beta}}{\rightarrow} q_0, q_0$ is related with $\omega$ through a single $\Rightarrow$-path $q_0 = \omega \overset{\delta}{\rightarrow} q_0 = \omega$; this path is the $\overline{\alpha} \overset{\delta}{\rightarrow} \gamma$-bridge from $q_0$ to $\omega$; there is no series of triples.

**Proof:** Initially new $\pi$ is an extension of $h_0$. Hence by $\Gamma_n, \kappa_n \vdash e : \tau$, it can be an ownership path only if it is free, i.e., $\mu = \text{free} \langle \ldots \rangle$. But then $\text{wit}(\pi)$ is free too: $\mu_r \overset{\beta}{\rightarrow} \mu \overset{\beta}{\rightarrow} \text{free} \langle \ldots \rangle = \text{free} \langle \ldots \rangle$. Since it is $h_0$'s extension and the nesting constraint excludes correlations to free modes, $h_0$ must be free, so that it cannot coincide with the call-link $h_r = s \overset{\mu_r}{\rightarrow} r$. This follows from Lemma 19 since the reserved ownership assumption obviously implies $g \models \text{UH}$.
If there is a triple in \( \pi \)'s the \( \alpha_0 \)-bridge there would be a first ownership path-pair \( \omega_0 \; \overleftarrow{\mu_\pi} \; s \overrightarrow{q_1} \; \alpha_1 \). The left one starts with \( h_r \) by the definition of initially new paths (since \( \mu_\pi \neq \text{co}<> \)). The triple of witness \( \text{wit}(\pi) = s \overleftarrow{\mu_\pi} \; q_0 \; \alpha_0 \), initial \( \equiv \)-path \( q_0 \overleftarrow{\delta^0-\gamma_{\omega_0}} \; \omega_0 \), and \( \omega_0 \; \overrightarrow{\mu_\pi} \; s \) would by the reserved ownership assumption imply that \( \text{wit}(\pi) \) also starts with \( h_r \). \( \text{wit}(\pi) \), however, by definition starts with \( h'_0 \). But \text{free} \( h'_0 \) cannot coincide with \( h_r \).

**Lemma 21** Assume the reserved ownership assumption in \( g^o \) and \( g = ogr(e, \bar{\eta}, s) \). Consider an initially new ownership path \( \pi \in PAP_{g^o}(r, \mu, \omega) \) whose \( \alpha_0 \)-bridge from \( q_0 \) to \( \omega \) via \( \Pi \in g^o \) starts with \( \equiv \)-path \( q_0 \overleftarrow{\delta^0-\gamma_{\omega_0}} \; \omega_0 \) via \( \Pi_0 \). For any \( \beta \) with \( \mu(\beta) \in A_\perp \), there was

\* a region-coupling \( r.\mu(\beta) = \omega.\beta \) via \( \Pi \cup \{ \text{wit}(\pi) \} \) or \( \Pi \cup \{ \text{wit}(\pi), h_r \} \), or

\* an ownership path-pair \( \langle h_r, h'_r, \text{wit}(\pi) \rangle = r.\mu(\beta) \overleftarrow{\delta^0} k.\alpha_0 \overrightarrow{q_0} \).\( \beta \) extending \( h_r \) and \( \text{wit}(\pi) \), respectively, by dummy edges \( h'_r \) and \( \beta \) while \( \mu(\beta) \in A_\perp \).

**Proof:** Let \( \omega' = q_0 \; \alpha_0 \) be short for the target of \( \pi \)'s witness \( \text{wit}(\pi) \in PAP_{g^o}(s, \mu, q_0, \alpha_0) \).

Notice that there are no triples in initially new ownership path \( \pi \)'s \( \alpha_0 \)-bridge, i.e., \( \omega_0 = \omega \) and \( \Pi_0 = \Pi \) (Lemma 20). Initially new \( \pi \) is an extension of \( h_0 \). Hence by \( \Gamma_n, c_n, e : \tilde{\tau} \), it can be an ownership path only if it is \text{free}. But then witness \( \text{wit}(\pi) \) of mode \( \mu_\pi \alpha_\mu \) is \text{free} too and correlates the same association roles which \( \pi \) correlates.

If \( \mu(\beta) = \perp \), this means that \( \mu \) lacks a correlation for \( \beta \). Then also \( \mu_\pi \alpha_\mu \) lacks it, so that \text{free} \( \text{wit}(\pi) \in PAP_{g^o}(s, \mu, \omega', \beta) \) established \( s.\perp = \omega' \beta \) via \( \{ \text{wit}(\pi) \} \). But since \( r.\perp = s.\perp \), this entails \( r.\perp = r.\mu(\beta) = \omega', \beta = q_0 \; \alpha_0 \beta \) via \( \{ \text{wit}(\pi) \} \). And since \( q_0 \overleftarrow{\delta^0-\gamma_{\omega_0}} \; \omega_0 = \omega \) means \( q_0 \; \alpha_0 \; \beta \equiv \omega \beta \) via \( \Pi_0 = \Pi \) we have \( r.\mu(\beta) = \omega, \beta \) via \( \Pi \cup \{ \text{wit}(\pi) \} \).

If \( \mu(\beta) = \alpha \in A \), this means that \( \mu \) contains the correlation \( \beta = \alpha<> \). But then \( \mu_\pi \), which contains \( \mu \), contains \( \alpha<> \). Therefore the existence of \( h'_0 = s \; \overleftarrow{r.\mu_\pi} \; \alpha \) presupposes a corresponding correlation \( \alpha = \hat{\mu} \) to some \( \hat{\mu} \) in the call-link's mode \( \mu_\pi \). It implies that \( \mu_\pi \alpha_\mu \) contains the correlation \( \beta = \hat{\mu} \).

\* \( \hat{\mu} \) cannot be \text{co}<> since there are no correlations to \text{co}; neither can it be a \text{read} mode Since \( \mu(\beta) = \mu_\pi (\alpha, \beta) = \alpha, \Gamma_n, c_n, e : \tilde{\tau} \), ensures that \( \mu_\pi \alpha_\mu (\alpha, \beta) = \mu_\pi \alpha_\mu (\beta) = \hat{\mu}(\epsilon) \) is not \text{read}.

If \( \hat{\mu} \) is a \text{free} or \text{rep} mode then, on one side, call-link \( h_r = s \; \overleftarrow{m \cdots \alpha = \hat{\mu} \cdots} \; r \) is extended by dummy edge \( h'_r = r \; \overleftarrow{\alpha = \hat{\mu}} \; r.\alpha \) to the ownership path \( \hat{\tau}' = s \; \overrightarrow{r.\alpha} \). On the other side, \( \text{wit}(\pi) \) of mode \( \mu_\pi \alpha_\mu = \text{free}<> \), \( \beta = \hat{\mu}, \ldots \) is extended by dummy edge \( h_\beta = q_0 \; \alpha_0 \; \overrightarrow{\delta^0} \; q_0 \; \alpha_0 \beta \) to the ownership path \( \hat{\tau} = s \; \overrightarrow{q_0} \; q_0 \; \alpha_0 \beta \)

If \( \hat{\mu} \) is an association mode \( \gamma <> \) then, first, call-link \( h_r = s \; \overleftarrow{m \cdots \alpha = \gamma \cdots} \; r \) established \( s.\gamma = r.\alpha \) via \( \{ h_r \} \) since it is \text{free} or \text{rep}; As a mode with a correlation \( \alpha = \hat{\mu} \), \( \mu_\pi \) cannot be a \text{co} - or association mode; and a \text{read} mode is excluded by \( \Gamma_n, c_n, e : \tilde{\tau} \) since \( \mu(\beta) = \mu_\pi (\alpha, \beta) \in A \). Second, \( \text{wit}(\pi) \) of mode \( \mu_\pi \alpha_\mu = \text{free}<> \), \( \beta = \gamma <> \), \( \ldots \) established \( s.\gamma = (q_0 \; \alpha_0 \beta) \) via \( \{ \text{wit}(\pi) \} \). Third, first and only \( \equiv \)-path \( q_0 \; \overleftarrow{\delta^0-\gamma_{\omega_0}} \; \omega_0 = \omega \) established \( q_0 \; \alpha_0 \; \beta \equiv \omega, \beta \) via \( \Pi_0 = \Pi \). All three region-couplings together mean \( r.\mu(\beta) = \omega, \beta \) via \( \Pi \cup \{ h_r, \text{wit}(\pi) \} \).
Lemma 22 (Qbridges)

a) If there is a $\alpha$-qbridge from $o$ to $w$ via $\Pi$ and $o.\alpha \Rightarrow o'.\alpha'$ via $\Pi'$, then there is a $\alpha'$-qbridge from $o'$ to $w$ via $\Pi \cup \Pi'$.

b) A $\alpha$-qbridge from $o$ to $w$ via $\Pi$ can be extended along $\omega \sim \beta \Rightarrow \gamma$ via $\Pi'$ to a $\alpha.\beta$-qbridge from $o$ to $w'$ via $\Pi' \equiv \Pi \cup \Pi' \cup \Pi_{\beta}$.

c) A $\alpha$-qbridge from $o$ to $q$ via $\Pi_{1}$ and a $\gamma$-qbridge from $q$ to $w$ via $\Pi_{2}$ concatenate to a $\alpha.\gamma$-qbridge from $o$ to $w$ via $\Pi' \equiv \Pi_{1} \cup \Pi_{2} \cup \Pi_{\gamma}$.

Proof: Part (a). The $\alpha$-qbridge starts with a $\Rightarrow$-path $o \sim q_{0} \sim \ldots q_{1} \sim w$ via $\Pi_{0}$. It, and $o.\alpha \Rightarrow o'.\alpha'$ imply the $\Rightarrow$-path $o' \sim q_{0} \sim \ldots q_{1} \sim w$ via $\Pi_{0} \cup \Pi'$, It, together with the $\alpha$-qbridge's reservation qbridge from $\omega_{1}$ to $\omega$, constitutes a $\alpha'$-qbridge from $o'$ to $w$ via $\Pi \cup \Pi'$.

Part (b). The proof is a generalization of that for the extension of bridges with triple-series (Lemma 14): The $\Rightarrow$-paths $o \sim q_{0} \sim \ldots q_{1} \sim w$ via $\Pi_{0}$, and each intermediate $q_{i} \sim q_{i+1}$ via $\Pi_{i}$ in the quadruples, can obviously be extended to $q_{0} \sim q_{1} \sim \ldots q_{1} \sim w$, respectively. The final $\Rightarrow$-path $q_{n} \sim q_{n+1} \sim \ldots w$ via $\Pi$, extends to $q_{n} \sim q_{n+1} \sim \ldots w$ via $\Pi_{i} \cup \Pi_{\beta}$. The corresponding ownership path-pair $(q_{i}, \alpha_{i}) \Rightarrow q_{i+1}, \alpha_{i+1}$ can be obtained by extension of those quadruple's ownership path-pairs $(q_{i}, \alpha_{i}) \Rightarrow q_{i+1}, \alpha_{i+1}$ if $\mu_{i}$ has an $\beta$-correlation to a free or rep mode.

As explained in the proof of Lemma 14, ownership path pairs $(\pi_{i}, \pi_{i})$ that cannot be extended instead establish $(q_{i}, \alpha_{i}), \beta \Rightarrow \mu_{i}, (\beta) \Rightarrow (q_{i}, \alpha_{i}), \beta$ via $(\pi_{i}, \pi_{i})$. This is extended to the left by $q_{i} \sim q_{i+1}$ via $\Pi_{i}$ implying $q_{i}, \beta \Rightarrow q_{i+1}, \alpha_{i}, \beta$, and to the right by $q_{i} \sim q_{i+1}$ via $\Pi_{i}$, implying $q_{i}, \beta \Rightarrow q_{i+1}, \alpha_{i}, \beta$, to the region-coupling $\omega_{i}, \beta \Rightarrow \omega_{i+1}, \beta$ via $\Pi_{i} \cup \Pi_{i} \cup \{\pi_{i}, \pi_{i}\}$. Hence, for any number of consecutive modes $\mu_{i}, \ldots, \mu_{j}$ with $\mu_{k}(\beta) \in \mathcal{A}_{\perp}$, the gap between $q_{0}$ (if $i = 0$) or the preceding extended ownership path-pair (if $i > 0$), and $\omega$ (if $j = n$) or the following extended ownership path-pair (if $j < n$) in the $\beta.\beta$-qbridge from $o$ to $w$ is closed, as shown in Lemma 14, by a $\Rightarrow$-path $q_{i-1} \sim \ldots q_{i+1} \sim \ldots w$, respectively.

Part (c). By part (b), the first qbridge can be incrementally extended along the second qbridge’s initial $\Rightarrow$-path $q \sim q_{0} \sim \ldots q_{1} \sim w$ via $\Pi_{0}$ to a $\alpha_{1}, \alpha_{2}$-qbridge from $o$ to $w$, via $\Pi_{1} \equiv \Pi_{1} \cup \Pi_{2,0} \cup \Pi_{\beta}$. It is extended by the second qbridge’s reservation qbridge from $\omega_{1}$ to $\omega$ via $\Pi' \equiv \Pi_{1} \cup \Pi_{2} \cup \Pi_{\gamma_{1}} \ldots \cup \Pi_{\gamma_{n}}$.

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 \( e_0, \eta_0, s_0, o_0, g_0 \rightarrow^* e', \eta', s', o_0, g' \) is a reduction defined relative to a program \( p \) with \( \vdash p \) start \( e_0 \) then \( g'^{\circ} \) 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 \( N = 0 \), \( g' \) is the empty graph \( g_0 = \emptyset \). Its extension \( g'^{\circ} \) by dummy edges of non-composition modes trivially satisfies the assumption. In the induction step \( N \rightarrow N + 1 \), reduction \( e_0, \eta_0, s_0, o_0, g_0 \rightarrow^* e, \eta, s, o, g \rightarrow e', \eta', s', o', g' \). By induction hypothesis, the assumption holds in \( g^{\circ} \). A look at the context rules shows that the changes to the object graph are absolutely independent of the term context around the redex \( \partial \). Hence we can move directly to a case analysis of the rule by which redex \( \partial \) is reduced.

In case of \( \{\text{var}\}, \{\text{var}\}, \{\text{rdcp}\}, \) and \( \{\text{null}\} \), the object graph is unchanged, so that the assumption is trivially preserved. As explained in the proof for Theorem 2, the case of \( \{\text{rdcp}\} \) 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 \( g'' \), a violation of the reserved ownership assumption by any combination \( \langle \pi, \varphi, \varphi', \pi' \rangle \) made of two ownership paths \( \pi \in \text{PAP}_{g^{\circ}}(o, \mu, q, \alpha) \) and \( \pi' \in \text{PAP}_{g^{\circ}}(o', \mu', q', \alpha') \), and two \( \righteq \)-paths \( \varphi = q - \alpha - \eta_{\omega} \) \( \omega \) and \( \varphi' = q' - \alpha' - \eta_{\omega} \) \( \omega \).

\{new\} The reduction of new\( \!\!\!\langle \delta \rangle \!\!\!\rangle c(\) adds an edge \( h_o = r.\mu_n. o \) to a fresh object \( o \), where \( \mu_o = \text{free}\!\!\!\langle \delta \rangle \). The kinds of ownership paths \( \pi \) in \( g^{\circ} \) guaranteed by Lemma 6 and the kinds of \( \righteq \)-paths \( \varphi \) in \( g^{\circ} \) guaranteed by Lemma 15 combine as follows:

- If \( \pi \) is an initially new ownership path then its target \( q.\alpha \) is some \( (o, \gamma).\alpha \) with \( \mu_o(\gamma, \alpha) \in \{\text{free}, \text{rep}\} \). Since no modes can be nested to an association mode, this implies \( \mu_o(\beta) \notin A \) for all prefixes \( \beta \) of \( \gamma.\alpha \). Hence the only \( \righteq \)-path \( q - \gamma^- - \eta_{\omega} \) \( \omega \) is the dummy edge path \( o, \gamma.\alpha^- \omega = o.\gamma.\alpha = q.\alpha \) (Lemma 15).

- If ownership path \( \pi \) is unchanged then \( q.\alpha \) cannot be \( o \) or one of its region objects. Hence \( \sigma(q) = q \) and \( q - \alpha - \eta_{\omega} \) \( \omega \) must be a \( \righteq \)-path with a counterpart \( \tilde{\varphi} = \sigma(q) - \alpha - \eta_{\omega} \) \( \sigma(\omega) \) in \( g^{\circ} \) (Lemma 15). \( \pi \) and \( \tilde{\varphi} \) are half a quadruple \( \langle \pi, \tilde{\varphi}, \ldots \rangle \) since \( q = \sigma(q) \).

- If \( \pi \) is internally new then its target \( q.\alpha \) is some \( o.\tilde{\gamma}.\tilde{\alpha} \) with \( \mu_o(\tilde{\gamma}) \notin A \). That is, \( q = o.\alpha' \) for the \( \alpha' \) with \( \alpha' = \gamma'.\alpha'' \). If \( \alpha' \) is a proper prefix of \( \gamma' \) with \( \gamma' = \alpha'.\alpha'' \) then \( q - \alpha - \eta_{\omega} \) \( \omega \) can be decomposed into a dummy edge path \( o.\alpha'.\alpha'' - \tilde{q} = o.\tilde{\gamma}.\tilde{\alpha''} \) and a \( \righteq \)-path \( \tilde{\varphi} = \tilde{q} - \gamma' - \eta_{\omega} \) \( \omega \) with counterpart \( \sigma(\tilde{q}) - \gamma - \eta_{\omega} \) \( \sigma(\omega) \) (Lemma 15). Hence this case of \( \pi \in \text{PAP}_{g^{\circ}}(o, \mu, q, \alpha) \) and \( \varphi = q - \alpha - \eta_{\omega} \) \( \omega \) can be reduced to the case of \( \tilde{\pi} \in \text{PAP}_{g^{\circ}}(o, \mu, q, \gamma) \) and \( \tilde{\varphi} = \tilde{q} - \gamma' - \eta_{\omega} \) \( \omega \). In the case that \( \alpha' \) is \( \gamma' \) or an extension of it, \( q - \alpha - \eta_{\omega} \) \( \omega \) must be a \( \righteq \)-path with a counterpart \( \tilde{\varphi} = \sigma(q) - \alpha - \eta_{\omega} \) \( \sigma(\omega) \) in \( g^{\circ} \) (Lemma 15). \( \pi \)'s witness \( \sigma(\pi) \) and \( \tilde{\varphi} \) are half a quadruple \( \langle \sigma(\pi), \tilde{\varphi}, \ldots \rangle \) since \( \sigma(\pi) \)'s target is \( r.\mu_o(\gamma).\gamma' = \sigma(q) \).

This shows that if \( \omega \) is some \( o.\tilde{\gamma} \) with \( \mu_o(\tilde{\gamma}) \notin A \) for all prefixes \( \tilde{\gamma} \) of \( \gamma \) then, first, \( \pi \) and \( \pi' \) must both be initially new, and, second, \( q.\alpha = q'.\alpha' \). Hence, \( \pi \) and
\( \pi' \) are one and the same potential access path \( \pi = \pi_1 = \pi' \), and thus automatically have the same source, mode, and shape: The quadruple \( \langle \pi, \varphi, \varphi', \pi' \rangle \) in \( g^\otimes \) satisfies the assumption.

In other cases of \( \omega, \pi \) and \( \pi' \) 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 \( \equiv \)-paths’ counterparts \( \tilde{\varphi} \) and \( \tilde{\varphi}' \) to a quadruple \( \langle \tilde{\pi}, \tilde{\varphi}, \tilde{\varphi}', \tilde{\pi}' \rangle \) in \( g^\otimes \). Since \( \pi \) and \( \pi' \) have the same source, mode, and shape as \( \tilde{\pi} \) and \( \tilde{\pi}' \), respectively, quadruple \( \langle \pi, \varphi, \varphi', \pi' \rangle \) in \( g^\otimes \) cannot violate the assumption if \( \langle \tilde{\pi}, \tilde{\varphi}, \tilde{\varphi}', \tilde{\pi}' \rangle \) did not violate it in \( g^\otimes \).

**{upd}** Consider the reduction of a destructive assignment \( \hat{e} = \ell = \langle c, \mu, o \rangle \) to a location \( \ell \) containing the old handle \( \langle c, \mu', \omega \rangle \): In this case, \( g' = g \odot c \overset{\mu'}{\omega} \odot c \overset{\mu_1}{\omega} \odot c \overset{\mu_2}{\omega} \odot c \overset{\mu_{k-1}}{\omega} \odot \ldots \odot c \overset{\mu_k}{\omega} \odot o \). The typeability of the redex \( \hat{e} \) following from Theorem 6 ensures that \( \mu \leq_m \mu' \). Proceed by induction on the number \( k \) of elementary conversions from \( \mu \) to \( \mu' \), i.e., \( \mu \leq_m \mu_1 \leq_m \mu_2 \ldots \leq_m \mu_{k-1} \leq_m \mu_k = \mu' \). In the base case, \( \mu = \mu' \), so that the addition of \( c \overset{\mu}{\omega} \odot o \) is canceled out by the removal of \( c \overset{\mu_k}{\omega} \odot o \). The removal \( g' = g \odot c \overset{\mu}{\omega} \odot c \overset{\mu_1}{\omega} \odot \ldots \odot c \overset{\mu_k}{\omega} \odot o \) creates no new ownership paths, and thus no new region-couplings, and also no new association paths, and thus no new \( \equiv \)-paths. Hence it trivially preserves the assumption.

In the induction step \( k \rightarrow k + 1 \), the induction hypothesis guarantees that the assumption holds in \( g^k = g^k \odot c \overset{\mu_k}{\omega} \odot c \overset{\mu_{k-1}}{\omega} \odot \ldots \odot c \overset{\mu_1}{\omega} \odot o \). After the final step from \( g^k \) to \( g' = g^k \odot c \overset{\mu_k}{\omega} \odot c \overset{\mu_1}{\omega} \odot \ldots \odot c \overset{\mu_2}{\omega} \odot c \overset{\mu_1}{\omega} \odot o \), consider the quadruple \( \langle \pi, \varphi, \varphi', \pi' \rangle \). Lemma 16 guarantees that there are no new \( \equiv \)-paths \( \varphi \) and \( \varphi' \) in \( g^\otimes \). 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 \( \pi \) and \( \pi' \) in §6.3.1 (see there).

**{ret}** For the reduction of \( \langle \text{return } \langle r, \mu_0, o \rangle; \rangle \) in the context of top-level environment with call-link \( h_r = \langle s, \mu_r, r \rangle \), consider first the replacement \( g'' = g \odot r \overset{\mu_0}{\omega} \odot s \overset{\mu_r}{\omega} \odot r \overset{\mu_0}{\omega} \odot s \overset{\mu_0}{\omega} \odot o \) of the exported handle and the call-link by the imported handle. Lemma 17 guarantees that there are no new \( \equiv \)-paths \( \varphi \) and \( \varphi' \) in \( g^\otimes \). 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 \( \pi \) and \( \pi' \) in §6.3.1 (see there). The final step to \( g' \), which only removes handles, namely those from the environment’s locations, obviously preserves the assumption.

**{call}** For an operation call expression \( \hat{e} = \langle s, \mu_r, r \rangle \leftarrow f(\langle s, \mu'_1, o_1 \rangle, \ldots, \langle s, \mu'_k, o_k \rangle) \) 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 \( \mu'_i \) of a sent handle is compatible to the adaption \( \mu_r \circ \mu_i \) of the mode of the corresponding method’s parameter relative to the call-link \( h_r: \mu'_i \leq_m \mu_r \circ \mu_i \).
Proceed by induction on the number \( k \) of (non-nil) sent handles \( h''_o = (s, \mu'_o, o_i) \). In the base case \( k = 0 \), \( g' = g \oplus r \cdot \circ<> \cdot r \). The only potentially new edge in the new extended graph \( g'' \), is \( r \cdot \circ<> \cdot r \). (Note that it is identical to its own inverse \( r \cdot \circuit<> \cdot r \).) All new ownership paths and association paths \( \pi \) in \( g' \) must contain \( r \cdot \circ<> \cdot r \) (at least once). They all have the obvious precursor \( \pi' \) in which all occurrences of the new edge have been cut out. These precursors have the same shape as \( \pi \), so that these new ownership paths cannot cause a violation of the assumption.

In the induction step \( k - 1 \rightarrow k \), \( g' = g''_k \cdot \circ<> \cdot o_k \oplus s \cdot \mu'_k \cdot o_k \), where the assumption holds for \( g''_k \) by induction hypothesis. The intermediate step \( g''_k = g_{k-1} \cdot \circ<> \cdot o_k \oplus s \cdot \mu'_k \cdot o_k \) of converting the \( k \)th sent handle’s mode \( J-L_k \) exactly to the adaption \( J-L_k \circ<> J-L_k \) of the parameter’s mode is like the conversion before assignment. The preservation of the assumption under this change was already shown in the \{upd\}-case above. Note the preserved assumption entails in particular \( g''_k \models \text{UH} \).

For the final substep, the actual supply \( g' = g''_k \cdot \circ<> \cdot o_k \oplus s \cdot \mu'_k \cdot o_k \), observe first that the multiplicity of all free edges remains below two: The only edges whose multiplicity is increased in the extension \( g''_k \) of \( g' \) are \( h_0 = r \cdot \circ<> \cdot o_k \), and, if \( \mu_k = \circ<> \cdot r \), its inverse \( h_0^{-1} = o_k \cdot \circ<> \cdot r \). If \( \mu_k \) is a free mode, then \( \mu_r \circ<> \mu_k \) is free. Hence sent handle \( h_0 = s \cdot \mu_r \circ<> \cdot o_k \) ensures by the assumption that it is the only old free edge targeting \( o_k \) and that its multiplicity is one. But then, after decreasing its multiplicity, no old free edge remains in \( g' \). The multiplicity of new free received handle \( h_0 = r \cdot \circ<> \cdot o_k \) 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 \( g^{\circ \circ} \). The ownership paths \( \pi \) and \( \pi' \) and \( \Leftrightarrow \)-paths \( \varphi \) and \( \varphi' \) possible in \( g^{\circ \circ} \) are described in Lemmas 9 and 18. In \( g^{\circ \circ} \) the source of \( \pi \)'s and \( \pi' \)'s precursor, witness, or witness, respectively was bridged by a series of \( n \) such quadruples. On the way from \( o \) to \( w \), there were \( h \) quadruples \( \langle \pi_1, \varphi_1, \varphi'_1, \pi'_1 \rangle, \ldots, \langle \pi_{n-1}, \varphi_{n-1}, \varphi'_{n-1}, \pi'_{n-1} \rangle \) and the half-quadruple \( \langle \pi_n, \varphi_n, \pi'_n \rangle \) on the way from \( w \) to \( o' \):

- For \( \varphi \) there was a \( \delta\)-qbridge: initial \( \varphi_j \), and quadruples \( \langle \varphi'_j, \pi'_j, \varphi_j+1, \varphi_j+1 \rangle, \ldots, \langle \varphi'_n, \pi'_n, \varphi_n, \varphi_n \rangle \).
- If \( \pi \) is unchanged or internally-only new, \( j = 1 \). \( \pi_1 = \pi_j \) is \( \pi \)'s equivalent precursor and \( \varphi_j = \varphi_1 \) is \( \varphi_j \) from \( \varphi \).
- If \( \pi \) is internally really new, there was a witness \( \pi_1 = \text{init}(\pi) \), an initial \( \Leftrightarrow \)-path \( \varphi_1 \), and a triple series transformable to a series of quadruples \( \langle \varphi'_1, \pi'_1, \pi_2, \varphi_2 \rangle, \ldots, \langle \varphi'_j, \pi'_j, \pi_j, \varphi_j \rangle \) in this case, the missing \( \varphi_j \) is the \( \Leftrightarrow \)-path \( q_j \cdot \circ<> \cdot \varphi_j \) implied by \( \pi \)'s \( \delta\)-\( \circ <> \)-bridge’s last \( \Leftrightarrow \)-path \( \varphi_j' = q_j \cdot \circ<> \cdot \varphi_j \) and by \( \varphi \)'s initial \( \Leftrightarrow \)-path \( \varphi_j = q \cdot \circ<> \cdot \varphi_j \) (Lemma 12).
- If \( \pi \) is initially new, there was a witness \( \pi_1 = \text{init}(\pi) \), an initial \( \Leftrightarrow \)-path \( \varphi_1 \), and a triple series transformable to a series of quadruples \( \langle \varphi'_1, \pi'_1, \pi_2, \varphi_2 \rangle, \ldots,
In this case, the missing \( \varphi_j \) is \( q_j \). If this reservation qbridge has the length one, i.e., \( (\pi_0, \varphi_0, \pi'_0) = (\pi_n, \varphi_n, \varphi'_n, \pi'_n) \), then the assumption for \( g''^\oplus \) guarantees that \( \pi_0 \)'s source \( o_0 \) and mode \( \mu_0 \) equals \( \pi'_n \)'s source \( o_{n+1} \) and mode \( \rho_{n+1} \). And if \( \mu_0 \) or \( \rho_{n+1} \) is \text{free}, then \( \pi_0 \) and \( \pi'_n \) have the same shape. Hence, first, if neither \( \pi \) nor \( \pi' \) are initially new, this means that they have the same source and, if one is \text{free}, the same initial edge: There no conflict with the assumption. Second, if ownership path \( \pi \) (or \( \pi' \)) is initially new then its mode \( \mu \) (or \( \mu' \)) is \text{free} by \( \Gamma_n, \kappa_n \vdash \dot{e} : \tau \), and thus the mode \( \mu_0 \circ \mu = \dot{\mu}_0 \) (or \( \mu_0 \circ \mu' = \dot{\mu}_{n+1} \)) of its witness \( \pi_0 = \text{wit}(\pi) \) (or \( \pi'_n = \text{wit}(\pi') \)) is \text{free} too. But then both \( \pi_0 \) and \( \pi'_n \) by assumption start with the initial edge of initially new paths: \( h'_0 \). If, while \( \pi \) is initially new, \( \pi' \) were unchanged, internally-only new, or internally really new (or vice versa), then \( \pi' \)'s unchanged initial edge would by assumption coincide with the initial handle \( h'_0 \) of \( \pi \)'s witness \( \text{wit}(\pi) \). But as initial edge of \text{free} path \( \pi \), \( h'_0 \)'s multiplicity of one (guaranteed by the assumption) decreases to zero in \( g''^\oplus \); it cannot be unchanged. Hence \( \pi \) and \( \pi' \) must be both initially new. Since their witnesses \( \pi_0 \) and \( \pi'_n \) by assumption have the same shape \( s \). If there is a longer series of quadruples \( (\pi_i, \varphi_i, \varphi'_i, \pi'_i) \), the assumption for \( g''^\oplus \) guarantees for each that \( \pi'_i \)'s source \( o_i \) equal \( \pi'_i \)'s source \( o_{i+1} \) and mode \( \mu_{i+1} \). That is, \( o_0 = s = o_{n+1} \) and \( \mu_0 = \mu_i = \ldots = \mu_{n+1} \). And if \( \mu_0 \) or \( \mu_{i+1} \) is \text{free}, then \( \pi_i \) and \( \pi'_i \) have the same initial edge \( u \). By assumption, and thus the same shape \( u \). For extending paths by association paths to \text{free} paths. The way how \( \pi'_i \)'s and \( \pi'_{i+1} \)'s shape are related by Lemma 18, this means that only the case of \( \mu_k = \text{co}<> \) is possible every ownership path \( \pi_i \) and \( \pi'_i \) has shape \( s \). Hence, if neither \( \pi \) nor \( \pi' \) are initially new, all of this means that they have the same source. Additionally, if one of them is \text{free}, they both can only have the shape \( s \). There is no conflict with the assumption. And if ownership path \( \pi \) (or \( \pi' \)) were initially new, then \( \mu_k \neq \text{co}<> \) and \( \pi_0 = \text{wit}(\pi) \) (or \( \pi'_n = \text{wit}(\pi') \)) would be \text{free} and start with \( h'_0 \). But this would contradict the \( \mu_0 = \text{co}<> \) necessary for \text{free} ownership paths \( \pi'_i \)'s and \( \pi'_{i+1} \) 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 \( e_0, \eta_0, s_0, om_0, g_0 \rightarrow^* e', \eta', s', om', g' \) is a reduction defined relative to a program \( p \) with \( \vdash p \) start \( e_0 \) then

\[ g' \models UH, UO \]

Proof: Lemma 23 guarantees the reserved ownership assumption about quadruples \( (\pi, \phi, \phi', \pi') \) in \( g^{\circ} \) (cf. Definition 13). UO and UH are included as the special case where \( \pi \in PAP'(w, \mu, v) \) and \( \pi' \in PAP'_g(w', \mu', v) \), and where \( \phi \) and \( \phi' \) are the trivial \( \equiv \)-path \( v \Leftarrow \eta \Leftarrow v \).

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 \( e_0, \eta_0, s_0, om_0, g_0 \rightarrow^* e', \eta', s', om', g' \) is a reduction defined relative to a program \( p \) with \( \vdash p \) start \( e_0 \) then

\[ g', \eta' \models MCP, MC \]

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 \( \mu_r \in Wr(\kappa) \) 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 \( \mu_r \) is free or rep” as “if \( \mu_r \) has base-mode free or rep,” and “\( \mu_r \) can be co” as “\( \mu_r \) can be co<”.

The sanctuary and \( Wr(\kappa) \) 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 “\( PAP(o, rep, \omega) \)” and “\( PAP(o, free, \omega) \)” to “\( PAP(o, rep<\ldots>, \omega) \)” and “\( PAP(o, free<\ldots>, \omega) \),” and expands “\( \mu_r \) cannot be free,” “\( \mu_r = rep, \)” and “\( \mu_r = co \)” to “\( \mu_r \) cannot be free<\ldots>,” “\( \mu_r = rep<\ldots> \),” and “\( \mu_r = co< \),” 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 \( e_0, \eta_0, s_0, om_0, g_0 \Rightarrow e, \eta, s, om, g \Rightarrow e', \eta', s', om', g' \) is a reduction defined relative to a program \( p \) with \( \vdash p \) start \( e_0 \) then for all \( o \in \text{dom}(om) \),

\[
\text{CState}_{s,om}(o) \neq \text{CState}_{s',om'}(o) \Rightarrow \exists i \leq n. r_i = o \land \kappa_i = \text{mut}
\]

where \( \eta = \eta_{h_1}, \ldots, \eta_{h_n} \) with \( h_i = (s_i, \mu_i, r_i) \).

**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 \( e_0, \eta_0, s_0, om_0, g_0 \Rightarrow e, \eta, s, om, g \Rightarrow e', \eta', s', om', g' \) is a reduction defined relative to a program \( p \) with \( \vdash p \) \( \text{start} \) \( e_0 \) then

\[
C\text{State}_{g,om}(o) \neq C\text{State}_{g',om'}(o) \Rightarrow \exists \omega \in \{o\} \cup S\text{anc}_g(o). s|_{flds\eta}(\omega) \neq s'|_{flds\eta}(\omega)
\]

So far only the source, mode and target of a potential access path or \( \Rightarrow \)-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 \( \Rightarrow \)-path by capturing or overwriting a handle in a field. The difficult case are the new class of \text{rep} paths in JaM that are extensions like \( \text{d read<dest=rep<...>>} \), \( i \text{ dest\Omega} \), \( e \text{ of paths by association paths since these may have prefixes that are not \text{rep} paths. There is not even a guarantee that in the field subgraph any path from } o \text{ to intermediate object } i \text{ 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 \text{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 \text{rep} handles or handles extensible to \text{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 \text{rep} path since their \text{rep} sub-objects are protected in \( o \)'s sanctuary. The safety of storage in a \text{free} sub-object depends on the initial \text{free} handle: It should be captured in a field. Then, in order to call the mutator on the \text{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 \( \text{Sub}(o) \) with the set \( H \) of the initial handles of the used \text{free} paths. The subset of \( o \)'s sub-objects safe for the storage of \text{rep} paths then is \( \text{Sub}_H(o) \) for some \( H \subseteq \text{grs}_{om}(s) \).

\[
\text{Sub}_H(o) = \{q\} \cup \text{Sub}_H(q) \\
\cup \bigcup_{h \in H} \text{Sub}_H(h) \\
\cup \bigcup_{\eta \in \text{PAP}_g(o,\text{free}<\eta>,q)} \{q\} \cup \text{Sub}_H(q)
\]

Moreover, one can easily see that it is safe if \text{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 \textit{legally mutable} in JaM, \( \text{lmut}(\omega) \), requires to have a legally mutable owner \( o \), or to be reachable by a \text{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):
\[ \text{lmut}(\omega) \leftrightarrow_\text{df} \exists o. \text{Osh}_o(o, \omega) \neq \emptyset \land \text{lmut}(o) \]
\[ \lor \exists o, \delta, h, \tilde{\pi}, \tilde{\pi} \in \text{PAP}_g(o, \text{free}^{\delta}, \omega) \land h \notin \text{fgrom}(s) \]

**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:

\[ s \big| \bigcup_{\omega \in \text{StRep}_{s, \text{om}}(o)} \text{flds}_s(\omega) \neq s' \big| \bigcup_{\omega \in \text{StRep}_{s', \text{om'}}(o)} \text{flds}_s'(\omega) \]

If the domain of the restriction is unchanged then the composite state change means that the store changed somewhere in \( \bigcup_{\omega \in \text{StRep}_{s, \text{om}}(o)} \text{flds}_s(\omega) = \bigcup_{\omega \in \text{StRep}_{s', \text{om'}}(o)} \text{flds}_s'(\omega) \). It changed at some \( \ell \in \text{flds}_s(\omega) \) of some \( \omega \in \text{StRep}_{s, \text{om}}(o) : s(\ell) \neq s'(\ell) \). Hence \( s \big| \text{flds}_s(\omega) \neq s' \big| \text{flds}_s'(\omega) \) for this \( \omega \).

Next consider a change \( \bigcup_{\omega \in \text{StRep}_{s, \text{om}}(o)} \text{flds}_s(\omega) \neq \bigcup_{\omega \in \text{StRep}_{s', \text{om'}}(o)} \text{flds}_s'(\omega) \). Since the set of field locations of each "old" object \( \omega \in \text{StRep}_{s, \text{om}}(o) \) remains unchanged in any reduction step (\( \text{flds}_s(\omega) = \text{flds}_s'(\omega) \)), such a change presupposes a change in the set of state-representing implementation objects. That is, \( \text{StRep}_{s, \text{om}}(o) \neq \text{StRep}_{s', \text{om'}}(o) \). This expands to

\[ \{ o \} \cup \bigcup_{\text{PAP}_g(o, \text{rep}, q) \neq o} \text{StRep}_{s, \text{om}}(q) \neq \{ o \} \cup \bigcup_{\text{PAP}_g(o, \text{rep}, q) \neq o} \text{StRep}_{s', \text{om'}}(q) \]

There must be an object \( q \) that is reachable from \( o \) by a non-empty sequence \( o = o_0 \xrightarrow{-\text{rep}} o_1 \xrightarrow{-\text{rep}} \ldots \xrightarrow{-\text{rep}} o_n = \omega \) of \( \text{rep} \) paths in field subgraph \( \text{fgrom}(s) \) but no such sequence in \( \text{fgrom}'(s') \), or vice versa. Paths \( \pi = o_j \xrightarrow{-\text{rep}} o_{j+1} \) in the field subgraph are created by assigning one of \( \pi \)'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 \( \text{lmut} \) and \( \text{Sub} \) developed above,

\[ q_0 \xrightarrow{-\text{Lp}} q_1 \ldots q_k \xrightarrow{-\text{Lp}} q_{k+1} \in \text{PAP}_g(o, \text{rep}^{\delta}, o_{j+1}) \]
\[ \Rightarrow \forall i \in \{1, \ldots, k\}, \neg\text{lmut}(q_i) \land \exists H \subseteq \text{fgrom}(s). q_i \in \text{Sub}_H(o_j) \]

This property will be shown below in Corollary 2 to be an invariant of legal JaM program executions and thus hold in particular in \( s, om, g \). An object \( q_i \) that is not legally mutable, \( \neg\text{lmut}(q_i) \), 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 \( \text{free} \) path \( \hat{\pi} \) with initial handle \( h \) stored in a field cannot occur as call-link in the environment stack since \( h \) 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 \( q_i \xrightarrow{-\text{Lp}} q_{i+1} \) in assignment
and destructive read steps. Only they can create or destroy \textit{rep} paths $\pi = q_0 \rightarrow q_1 \rightarrow \ldots \rightarrow q_k \rightarrow q_{k+1} \in PAP_{\text{fgrom}(s)}(o_j, \text{rep} \ldots \rightarrow \text{rep} \ldots \rightarrow \text{rep})$ in the field subgraph. Consequently, the object $q_i$ whose field change $(s' [\text{rdbs}(q_i) \neq s' [\text{rdbs}(q_i)])$ creates or destroys a path $\pi = o_j \rightarrow \text{rep} \rightarrow o_{j+1}$ in sequence $o = o_0 \rightarrow \text{rep} \rightarrow o_1 \rightarrow \text{rep} \rightarrow \ldots \rightarrow \text{rep} \rightarrow o_n = \omega$ of \textit{rep} paths in the field subgraph is in $o$'s sanctuary ($q_i \in \text{Sanc}(o_j) \land o_j \in \text{Sanc}(o) \Rightarrow q_i \in \text{Sanc}(o)$).

The invariant used above needs to be refined to strengthen it for a preservation proof.

\textbf{Step 1.} Provision has to be taken for showing the preservation of the invariant's “$q_i \in \text{SubH}(o)$” case when one of the captured \textit{free} handles $h \in H \subseteq \text{fgrom}(s)$ 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 \textit{rep} path $\pi \in PAP_{\text{fgrom}(s)}(o, \text{rep} < \delta, \omega)$ that passed through $q_i$ in the field subgraph view. To this end, field-captured \textit{rep} paths $\pi$ through sub-objects $q_i$ in field-captured \textit{free} sub-objects $q$ are required to contain the corresponding \textit{free} path's initial handle. They have to pass through all the \textit{free} handles $h \in H$ on which $q_i \in \text{SubH}(o)$ was based: The condition $H \subseteq \{\pi\}$ has to be added. Actually, since all handles in $\pi$ are in the field subgraph, it can replace the condition $H \subseteq \text{fgrom}(s)$.

\textbf{Step 2.} In order to prepare for step 3 with \textit{rep} paths not only in the field subgraph and uncaptured \textit{free} handles (that can be exchanged as parameter and result), we have to be more precise about the \textit{free} sub-objects in which \textit{rep} handles and handles extensible to \textit{rep} paths can actually be stored: $o$ can store them through a \textit{free} path in its direct \textit{free} sub-objects only if the \textit{free} path’s mode $\hat{\mu}$ contains a (nested) correlation of \textit{rep} mode, i.e., $\hat{\mu}(\bar{\eta}) = \text{rep}$ for some $\bar{\eta}$. And $o$’s sub-objects $q$ can store such handles in their \textit{free} sub-objects only if the handles are association handles or can be extended to association paths and if the \textit{free} path’s mode $\hat{\mu}$ contains a correlation to the corresponding association mode, i.e., $\hat{\mu}(\bar{\eta}) = \beta \in A$.

That is, as non-immutable intermediate objects in \textit{rep} paths in the field subgraph only those objects $u \in \text{SubH}(o)$ should be accepted where all \textit{free} handles $h \in H$ on the way from $o$ to $u$ have a mode $\hat{\mu}$ with $\hat{\mu}(\bar{\eta}) \in \{\text{rep}\} \cup A$. Using a corresponding predicate $\text{repdn}(u \leftarrow v) \iff \exists \bar{\alpha} \in \hat{\mu}(\bar{\eta}) \in \{\text{rep}\} \cup A$, the invariant for all objects $q_i$ in $o$’s \textit{rep} paths $\pi$ in $\text{fgrom}(s)$ now reads

\[ \neg \text{mut}(q_i) \lor \exists H \in \{\pi\}, q_i \in \text{SubH}(o) \land \forall h \in H. \text{repdn}(h) \]

\textbf{Step 3.} Showing preservation of the invariant about \textit{rep} paths in the field subgraph under the capturing of handles in fields requires us to know already something similar about \textit{all} \textit{rep} paths $\pi \in PAP_{\mu}(o, \mu, \omega)$ in $g$. Also these are safe if they pass through only objects $u$ safe for \textit{rep} paths in the field subgraph. But we have to deal with an additional possibility specific to uncaptured \textit{rep} paths $\pi$: Naturally, an uncaptured \textit{rep} path $\pi$ should be able to pass through sub-objects $q_i$ of $o$ reachable not only through captured but also uncaptured \textit{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 \( h \) of the free paths are edges of \( \pi \): Then \( \pi \) can 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 \( \pi \) should be

\[
\neg \text{lmut}(q_i) \\
\lor \exists H \in \{\pi\}. q_i \in \text{Sub}_H(o) \land \forall h \in H. \text{repdn}(h) \\
\lor \exists \hat{q}. \neg \text{lmut}(\hat{q}) \land \exists H \in \{\pi\}. q_i \in \text{Sub}_H(\hat{q}) \land \forall h \in H. \text{repdn}(h)
\]

But this is not all.

Step 4. We also have to deal with the possibility that \( \pi \) contains not the uncaptured initial handle \( \hat{l} \) 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 \( \pi \) is safe if \( h \) is an uncaptured edge local to an observer invocation, because then \( h \) blocks \( \pi \)’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 \( \hat{l} \) are "observer bounded," \( \text{obsbd}(h, h') \), in the following sense: \( h \) and \( \hat{l} \) are uncaptured handles; \( h \) is local to call-levels \( l \) in which an observer is executing; and (the unique free handle) \( h \) 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, \( h \) must be in the terminated invocation, so that its destruction by return leaves \( q_i \) as an immutable object. The uncaptured free handle \( \hat{l} \) may be passed through another free handle \( \hat{l}' \) to an object and captured there. But then we will have \( \text{obsbd}(h, h') \) instead.

That is, the condition \( \exists H \in \{\pi\}. q_i \in \text{Sub}_H(o) \land \forall h \in H. \text{repdn}(h) \) needs to be expanded to \( \exists H. q_i \in \text{Sub}_H(o) \land \forall h \in H. \text{repdn}(h) \land (\{h\} \in \{\pi\} \lor \exists h. \{h\} \in \{\pi\} \land \text{obsbd}(h, h)) \), and the same for the \( q_i \in \text{Sub}_H(\hat{q}) \) case. To make the extended invariant easier to handle, its formulation will be restructured: For each object \( u \) in any rep path \( \pi \) of \( o \) there is an \( H \) such that

\[
\neg \text{lmut}(u) \lor u \in \text{Sub}_H(o) \lor \exists \hat{q}. \neg \text{lmut}(\hat{q}) \land u \in \text{Sub}_H(\hat{q}) \\
\land \forall h \in H. \text{repdn}(h) \land (\{h\} \in \{\pi\} \lor \exists h. \{h\} \in \{\pi\} \lor \text{obsbd}(h, h))
\]

where \( \text{obsbd}(h, h) \) is formalized as follows using the kinds \( \kappa_i \) of the methods executing at the various call-levels in the environment stack \( \tilde{\eta} = \eta_{h_1}^{\kappa_1} \ldots \eta_{h_n}^{\kappa_n} \), and using the receivers \( r_i \) and modes \( \mu_i \) of the call-links \( h_i = (s_i, \mu_i, r_i) \) in it:

\[
\text{obsbd}(o \downarrow \omega, h) \leftrightarrow \forall h. \omega. h \notin \text{grfm}(s) \land \mu \neq \text{co} < > \\
\land \forall l, l'. \text{atlevel}(o \downarrow \omega, l) \Rightarrow \kappa_l = \text{obs} \\
\land \text{atlevel}(h, l') \Rightarrow \forall i. \mu_i \leq l \Rightarrow \kappa_i = \text{obs}
\]

\[
\text{atlevel}(h, l) \equiv h \in \text{im}(s | \text{im}(\eta)) \lor \varepsilon_i = \ldots h \ldots \lor (l < n \land h = h_{i+1})
\]
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`, or as a temporary value in the runtime term at method nesting level `l`, written `h ∈ E_l`, or as the call-link `h_{l+1}` from call-level `l` to the next. The `E_l` are determined by the decomposition of current runtime term `e` using a series of reduction context `E_{l-1}, ..., E_0` and an innermost runtime term `E_0` containing no inlined method body, so that `e = E_1[...E_{n-1}E_n]`.

**Step 5.** The final step is to move to the graphs `g^®` and `fgom(s)^®` extended by region objects, and to generalize ownership paths paths `π` to ownership paths reservations `<π, φ>`. 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(ω)` of `lmut(ω)`, `RSubH(o)` of `SubH(o)`, `res(o, μ, ω)` of `PAP(o, μ, ω) ≠ 0`, and `res(h, μ, ω)` of `h • π ∈ PAP(o, μ, ω)`.

**Lemma 26** If `e_0, η_0, s_0, om_0, g_0` \(\Rightarrow^* e, \bar{η}, s, om, g\) is a reduction defined relative to a program `p` with `\vdash p` start `e_0` then

\[
\begin{align*}
h_1 \cdot \bar{π} & \in \text{PAP}_g^®(o, \text{rep}<...>, q, \bar{γ}) \land q \dashv \bar{π} - \gamma_\text{om} \omega \text{ via } \Pi \\
\Rightarrow & \forall \{v, \text{om}, w\} \in \{\bar{π}\} \cup \Pi . \forall u \in \{v, w\} . \exists \Pi . \\
\neg \text{rmut}(u) & \lor u \in \text{RSubH}(o) \lor \exists \bar{q} . \neg \text{rmut}(\bar{q}) \land u \in \text{RSubH}(\bar{q}) \\
\land & \forall h \in H . \text{repln}(h) \land \left(\{h\} \in \{π\} \lor \exists \Pi . \{h\} \in \{π\} \cup Π \land \text{obsbd}(h, h)\right)
\end{align*}
\]

The invariant as it was used in Lemma 25 is implied by this:
Corollary 2

\[ q_0 \xrightarrow{\mu_0} q_1 \ldots q_k \xrightarrow{\mu_k} q_{k+1} \in PAP_{\text{from}}(o, \text{rep}\langle \delta \rangle, \omega) \Rightarrow \forall i \in \{1, \ldots, k\}, \neg \text{imut}(q_i) \lor \exists H \subseteq \text{gr}_{\text{om}}(s), q_i \in \text{Sub}_H(o) \]

Proof: Any \text{rep} path \( \pi \in PAP_{\text{from}}(o, \text{rep}\langle \ldots \rangle, \omega) \) in the field subgraph is extended to a reservation \( \langle \pi, \varphi \rangle \) by trivial \( \equiv \)-path \( \varphi = \omega \xrightarrow{\pi} \omega \) via \( \emptyset \). Since \( \{\pi\} \cup \Pi \subseteq \text{gr}_{\text{om}}(s) \), the alternative with \( \text{obsbd}(h, h) \) never applies. All \( h \in H \) are in \( \pi \), hence \( H \subseteq \text{gr}_{\text{om}}(s) \). Obviously, the lemma’s \( \neg \text{imut}(u) \) implies \( \neg \text{imut}(u) \). \( u \in \text{Sub}_H(o) \) or \( u \in \text{Sub}_H(q) \), respectively, if there is a connection by ownership \textit{paths}. Otherwise the bridge of ownership reservations on the way from \( o \) or \( q \) to \( u \) contains a right-most reservation that is not an ownership path. Its target \( q' \) is \( \neg \text{imut}(q') \). If \( q' \) is \( u \), then \( \neg \text{imut}(u) \). And if \( u \in \text{Sub}_H(q) \) with not legally mutable \( q \) or \( q' \), then also \( u \) is not legally mutable since \( H \subseteq \text{gr}_{\text{om}}(s) \). 

The preservation of the lemma’s invariant under conversion of handles from \text{free} to \text{rep} requires a similar invariant about \text{free} reservations \( \langle \pi, \varphi \rangle \). These are harder to reason about than \text{rep} reservations because \text{free} handles can be exchanged between objects. Here it is crucial that the nesting constraint on modes excludes correlations to \text{free} modes on top of that to \text{co-}\text{}. This lets \text{free} paths be like in base-JaM: a \text{free} edge followed by \text{co-}edges. Hence the intermediate objects \( u \) in \text{free} paths starting with \( h_1 \) are all targeted by \text{free} paths starting with \( h_1 \). For \text{free} reservations, this has to be generalized to the objects \( u \in \text{RFree}(h_1) \) to which \text{free} reservations exist with initial handle \( h_1 \) (cf. figure 6.12), and to objects that are immutable:\footnote{Since no proper region object can be targeted by \text{free} paths made only of \text{free} edges and \text{co-}edges, \( \varphi \) must be an \( \equiv \)-\textit{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 \text{rep} reservations, to say that all objects in \( \pi \) and \( \Pi \) are in \text{RFree}(h_1) \) or immutable.}

\[ h_1 \cdot \pi \in PAP_{\text{from}}(o, \text{free}\langle \ldots \rangle, q_\varphi) \land q \xrightarrow{\varphi} \omega \text{ via } \Pi \Rightarrow \forall \{v \notin w\} \in \{\pi\} \cup \Pi, \forall u \in \{v, w\}, \neg \text{imut}(u) \lor u \in \text{RFree}(h_1) \]

Proof of the lemma: The lemma’s invariant on \text{rep} paths and the auxiliary invariant on \text{free} paths are handled simultaneously by induction on the number \( N \) of reduction steps from \( e_0 \) to \( e \). In the base case \( N = 0, \ g = g_0 = \emptyset \) is empty. Hence there can be no ownership paths, so that the invariant holds trivially. In the induction step \( N \to N + 1 \), reduction \( e_0, \eta_0, s_0, om_0, g_0 \Rightarrow e_N, \eta_N, s_N, om_N, g_N \) is continued \( e_N, \eta_N, s_N, om_N, g_N \Rightarrow e, \eta, s, om, g \). In most steps the two invariants holding by induction hypothesis are obviously preserved. For the other steps we will make use of the \textit{typeability} of redex \( \hat{e} \) that follows from Theorem 6, and of the \textit{reserved ownership assumption} that Lemma 23 guarantees (without mention of the theorem/lemma).
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 \( \equiv \)-paths, thus preserving all old case of \( \neg \text{rmut}(u) \). The removal may however interrupt an old ownership path \( \pi \) or \( \equiv \)-path \( \varphi \), and destroy an ownership reservation \( \langle \pi, \varphi \rangle \). If this was the last ownership reservation on the target object \( \omega \), then \( \omega \) is now immutable. Together with \( \omega \), all objects \( u \) become immutable that are reachable from \( \omega \) via sequences of reservations \( \langle \pi_t, \varphi_t \rangle \) with rep path \( \pi_t \) or with free path \( \pi_t \) with field-captured initial handle. Consequently, whenever a relationship \( u \in R\text{Free}(h) \) ceases to hold by the loss of ownership reservations, \( u \) is now immutable. And if a relationship \( u \in R\text{Sub}_{H}(o) \) ceases to hold, there must be an intermediate object \( v \) in the ownership reservation connection from \( o \) to \( u \) i.e., with \( v \in R\text{Sub}_{H'}(o) \) and \( u = v \) or \( u \in R\text{Sub}_{H''}(v) \) 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.

Where a handle is stored is only relevant in the definition of \( r\text{mut} \). 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 \( r \)'s mutators. Hence the mutator access properties (Theorem 8) imply that \( r \) is mutable. Thus so are all its sub-objects, in particular, the targets of \( r \)'s free paths based on \( h \), and their sub-objects. The invariants are preserved.

The only thing that might change in the object graph by non-destructive read access is the addition of a read handle \( h_0 = h_0[\text{read/free}] \) for a free handle \( h_0 \) in the variable. All new paths \( \pi' \) have a precursor \( \pi = \pi'[h_0/h_0] \) with the same base-mode or base-mode read instead of free. Hence all region-couplings and \( \equiv \)-paths \( \varphi' \) via \( \Pi' \) have precursors with path-base \( \Pi = \Pi'[h_0/h_0] \). There is no change in \( R\text{Free}(h) \), \( R\text{Sub}_{H}(o) \), nor \( r\text{mut}(\omega) \). 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 \( R\text{Free}(h) \), or in \( R\text{Sub}_{H}(q) \) of an object \( q \) that is \( o \) or immutable. From all \( h \in H \) we know that there is an \( h \in \langle \pi, \varphi \rangle \) with \( h = h \cup \text{obsbd}(h, h) \). In \( \langle \pi', \varphi' \rangle \), this \( h \) still exists if it is not \( h_0 \). And if \( h = h_0 \), then in \( \langle \pi', \varphi' \rangle \) we have \( h_0 \) instead. The typeability of redex \( \hat{e} \) ensures that read-alias \( h_0 \) is created only in an observer and only from handle \( h_0 \) in the location of a local variable: \( \text{obsbd}(h_0, h_0) \). Hence the invariants are preserved.

In an object creation step, a free edge \( h_0 = r \, \text{free} \, o \) to a fresh object is added: \( g = g_N \oplus r \, \text{free} \, o \).

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 \( u \in R\text{Free}(h) \) and \( u \in R\text{Sub}_{H}(o) \). The only really new ownership paths
target some \( \mathbf{o.\bar{\alpha}} \) (Lemma 6), and the only really new \( \equiv \)-paths have some \( \mathbf{o.\bar{\alpha}} \) 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 \( \pi \) can combine with new \( \varphi \) only if it differs from its counterpart \( \sigma(\varphi) \) by having target \( \mathbf{o.\bar{\alpha}} \). New \( \pi \) can combine with an unchanged \( \varphi \) on \( \mathbf{o} \)'s side, but \( \varphi \)'s target will always be the same one as \( \pi \)'s, i.e., some \( \mathbf{o.\bar{\alpha}} \). Internally new \( \pi \) can combine with a really new \( \varphi \) 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, \( \neg \text{rmut}(u) \) is preserved.

All ownership reservations on a region object \( \mathbf{o.\bar{\alpha}.\bar{\alpha}'} \) are witnessed by ownership reservations on \( \mathbf{r.\mu_o(\bar{\alpha}).\bar{\alpha}'} = \sigma(\mathbf{o.\bar{\alpha}.\bar{\alpha}'}). \) Hence if sequences of old ownership reservations on \( \mathbf{r.\mu_o(\bar{\alpha}).\bar{\alpha}'} \) left it immutable or placed it in \( RFree(h) \) or \( RSub_H(o) \) in \( g_N^\equiv \), then the corresponding new sequences of ownership reservations on \( \mathbf{o.\bar{\alpha}.\bar{\alpha}'} \) respectively leave it immutable or place it in \( RFree(h) \) or \( RSub_H(o) \).

Consider the objects \( \mathbf{o.\bar{\alpha}} \) where \( \mu_o(\bar{\alpha}) = \text{read} \): The nesting constraint on valid modes like \( \mu_o \) ensures that no extraction \( \mu_o(\bar{\alpha}') \) for a prefix of \( \bar{\alpha} \) can be an association mode. Hence the only non-dummy path targeting \( \mathbf{o.\bar{\alpha}} \) is the initially new path \( \pi_\bar{\alpha} \) of a \( \text{read} \) mode. \( \mathbf{o.\bar{\alpha}} \) has no owner. Moreover, there is no chance to construct any ownership reservation on \( \mathbf{o.\bar{\alpha}} \) (see above). \( \mathbf{o.\bar{\alpha}} \) is immutable.

Second, consider the objects in reservations \( \langle \pi, \varphi \rangle \) with unchanged ownership path \( \pi \) or unchanged \( \equiv \)-path \( \varphi \): An unchanged \( \pi \) cannot pass through fresh \( \mathbf{o} \) and its region objects, and thus can combine only with unchanged \( \equiv \)-paths, which also do not pass through fresh \( \mathbf{o} \) and its region objects. Since nothing changed for these objects, reservations with unchanged \( \pi \) still satisfy the invariants.

The only unchanged \( \equiv \)-paths that can combine with a new ownership path \( \pi \) are those based on dummy edge sequences \( \mathbf{o.\alpha_1 \ldots \alpha_k \overset{a_{k+1} \ldots a_n}{\rightarrow} o.\bar{\alpha},} \) since all new \( \pi \) end in some \( \mathbf{o.\bar{\alpha}} = (o.\alpha_1 \ldots \alpha_k).\alpha_{k+1} \ldots \alpha_n \). These have the same target as \( \pi \) and contain only edges already contained in \( \pi = h_{o^*} \circ o.\bar{\alpha}. \) \( o.\bar{\alpha} \), or \( \pi = \pi_1 \circ h_{o^*} \circ o.\bar{\alpha}. \) \( o.\bar{\alpha} \). Hence these reservations are safe iff the new ownership path \( \pi \) is safe. We can ignore the objects in old ownership paths and old \( \equiv \)-paths, and concentrate on the (objects in) in new ownership paths (in combination with new or old \( \equiv \)-paths) and in (the path-bases of) new \( \equiv \)-paths (in combination with new or old ownership paths).

Third. Consider the objects \( u \) in new ownership paths \( \pi \) in reservations \( \langle \pi, \varphi \rangle \):

<table>
<thead>
<tr>
<th>in ( g_N^\equiv )</th>
<th>in ( g^\equiv )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( u \in RFree(h), u \in RSub_H(o) ) ( \Rightarrow ) ( u \in RFree(h), u \in RSub_H(o) )</td>
<td>( \neg \text{rmut}(u) ) ( \Rightarrow ) ( \neg \text{rmut}(u) )</td>
</tr>
<tr>
<td>( \neg \text{rmut}(\mathbf{r.\mu_o(\bar{\alpha}).\bar{\alpha}'} ) ( \Rightarrow ) ( \neg \text{rmut}(\mathbf{o.\bar{\alpha}.\bar{\alpha}')} )</td>
<td>( \mathbf{r.\mu_o(\bar{\alpha}).\bar{\alpha}'} \in RFree(h), RSub_H(o) ) ( \Rightarrow ) ( \mathbf{o.\bar{\alpha}.\bar{\alpha}'} \in RFree(h), RSub_H(o) )</td>
</tr>
</tbody>
</table>
| \( \mu_o(\bar{\alpha}) = \text{read} \) \( \Rightarrow \) \( \neg \text{rmut}(\mathbf{o.\bar{\alpha}}) \) | \n
Figure 6.13: Change of relationships in creation steps
(a) \( \tau \) can be a \textit{initially new ownership path} \( \pi_\alpha = h_0 \cdot \text{o} \cdot \alpha \cdot \text{o} \cdot \alpha \) with initial edge \( h_1 = h_0 \): In the case \text{free} \( \pi_\alpha \), the nesting constraint on valid modes like \( \mu_o \) ensures that \( \alpha = \epsilon \). Hence there is only \( \text{o} = \text{o} \cdot \epsilon \) in \( \pi_\alpha \), which is obviously in \text{RFree}(h_0).

In the case of \text{rep} path \( \pi_\alpha \), \( \mu_o(\alpha) = \text{rep} \) the nesting constraint ensures that all its non-trivial prefixes are initially new \text{free, rep, or read} paths \( \pi_{\alpha'} \) of modes \( \mu_{\alpha'} \) with \( \mu_{\alpha'}(\alpha''') = \text{rep} \). In the first two cases, we have \( \text{o} \cdot \alpha'' \in \text{RSub}_0(r) \) for \( \pi_\alpha \)'s intermediate object \( \text{o} \cdot \alpha'' \), and for the latter case it was shown above that \( \text{o} \cdot \alpha'' \) is immutable.

(b) If \( \pi \) is a \textit{internally new ownership path} \( \pi_1 \cdot \pi_\beta \cdot \pi_3 \) with \( \mu_o(\alpha) = \beta \), it has a witness \( \sigma(\pi) = \pi_1 \cdot \pi_2 \) of the same mode with \( \pi_2 = r \cdot \beta \cdot \alpha' \cdot r \cdot \beta \cdot \alpha'' \). Since \( \pi \) contains non-initially the non-co handle \( h_o \), the nesting constraint on modes means that it cannot be \text{free}. \( \pi \) and \( \sigma(\pi) \) must be \text{rep} paths. Since in \( g_N^\circ \), there was the old reservation \( \langle \sigma(\pi), r, \beta, \alpha'', \text{E} \rangle \), the induction hypothesis covers all objects in \( \sigma(\pi) \). Since the reservation is old, the objects in \( \sigma(\pi) \) still satisfy the invariant in \( g^\circ \), as shown above. Observe that in the third and fourth case, the handle \( h \) guaranteed to be in \( \sigma(\pi) \) must be in the \( \pi_1 \)-segment common with \( \pi \) since the \( \pi_2 \)-segment consists of dummy edges that are neither \text{free} nor absent from the extended graph \( \text{fgrom}_N(\sigma_N) \).

1. Hence the objects \( u \) in the \( \pi_1 \)-segment of \( \pi \) satisfy the invariant in \( g^\circ \).
2. The objects \( u = \text{o} \cdot \alpha' \) in the \( \pi_\beta \)-segment of \( \pi \) are immutable if \( \mu_o(\alpha') = \text{read} \) (see above). The case of \( \mu_o(\alpha') \in \{\text{free}, \text{rep}\} \) is covered by looking at the possibilities which the induction hypothesis guarantees for \( r \) in \text{rep} path \( \sigma(\pi) \). There are no other cases of \( \mu_o(\alpha') \): \( \mu_o \) must contain a (nested) correlation to \( \beta \) by \( \mu_o(\alpha) = \beta \), so that the nesting constraint on valid mode \( \mu_o \) ensures that all extractions \( \mu_o(\alpha') \) by a prefix \( \alpha' \) of \( \alpha \) are \text{free, rep, or read}.

First, \( r \) may be immutable. Then in the \text{rep-case} also the target \( u = \text{o} \cdot \alpha' \) of its initially new \text{rep} path \( \pi_\alpha \) is immutable. In the \text{free} case, we have \( u \in \text{RSub}_{\{h_0\}}(r) \) with immutable \( r \) and with \( h_0 \) in \( \pi \) and \text{repdn}(h_0) since \( \mu_o(\alpha'') = \beta \). Second, \( r \in \text{RSub}_H(q) \). Hence initially new ownership path \( \pi_{\alpha'} \) from \( r \) to \( u \) of mode \( \mu_{\alpha'} \) means that also \( u \in \text{RSub}_H(q) \) with \( H' = H \) in the \text{rep} case and, in the \text{free} case, \( H' = H \cup \{h_0\} \) with \( h_0 \) in \( \pi \) and \text{repdn}(h_0) since \( \mu_o(\alpha'') = \beta \).

3. The objects \( u = \text{o} \cdot \alpha \cdot \alpha'' \) in the \( \pi_3 \)-segment of \( \pi \) are immutable, in \text{RFree}(h_1), in \text{RSub}_H(o) or in \text{RSub}_H(q) since the objects \( \sigma(u) = r \cdot \beta \cdot \alpha'' \) in the \( \pi_2 \)-segment of \( \sigma(\pi) \) are by induction hypothesis guaranteed to be immutable, in \text{RFree}(h_1), in \text{RSub}_H(o) or in \text{RSub}_H(q) (see above). Whether free \( h \in H \) is in \( \sigma(\pi) \) or there is an uncaptured handle \( h \) in \( \sigma(\pi) \) with \text{obsbd}(h, h), this handle still exists in \( \pi \), as shown above.

Fourth. Consider the objects \( u \) in the path-base \( \Pi \) of new \( \Rightarrow \)-path \( \varphi = q \cdot \varphi' \cdot \omega \) via \( \Pi \). Observe that \( h_0 \) must be in \( \Pi \) since otherwise \( \varphi \) could not be new: \( \{h_0\} \subseteq \Pi \).

It has a precursor \( \sigma(\varphi) = \sigma(q) \cdot \varphi' \cdot \omega \) via \( \Pi' \equiv \sigma(\Pi \setminus \Pi_N \setminus \Pi_\varphi) \). This precursor exists also with a path-base that includes the dummy edges from \( r \) to \( \sigma(q) \) or \( \sigma(\omega) = r \cdot \mu_o(\alpha) \cdot \alpha' \), respectively: The dummy edge sequence \( r \cdot \mu_o(\alpha) \cdot \alpha' \cdot r \cdot \mu_o(\alpha) \cdot \alpha'' \) as a \( \Rightarrow \)-path means the region-coupling \( r \cdot \mu_o(\alpha) \cdot \alpha' \cdot \epsilon \cdot (r \cdot \mu_o(\alpha) \cdot \alpha' \cdot \epsilon \) via \( \Pi'' = \{r \cdot \mu_o(\alpha) \cdot \alpha', r \cdot \mu_o(\alpha) \cdot \alpha' \} \). This redundant
region-coupling extends the path-base of trivial \( r_\mu_o(\tilde{x}).\tilde{x}' \rightarrow \tilde{\epsilon} \rightarrow \tilde{\gamma}_{s_\mu} r_\mu_o(\tilde{x}).\tilde{x}' \) via \( \emptyset \) to the \( \leftarrow\rightarrow \)-path \( \varphi_e = r_\mu_o(\tilde{x}).\tilde{x}' \rightarrow \tilde{\epsilon} \rightarrow \tilde{\gamma}_{s_\mu} r_\mu_o(\tilde{x}).\tilde{x}' \) via \( \Pi' \). Its redundant concatenation with \( \sigma(\varphi) \) to \( \varphi' = \varphi_e \cdot \sigma(\varphi) \) or \( \sigma(\varphi) \cdot \varphi_e \), respectively, extends the path-base to \( \Pi' \cup \Pi'' \).

(a) In the case of unchanged \( \pi \), new \( \varphi \) must be new at its target, i.e., \( \sigma(q) = q \) and \( \omega = \mathbf{o} \tilde{x} \tilde{x}' \) with \( \sigma(\omega) = r_\mu_o(\tilde{x}).\tilde{x}' \). Then \( \pi \) combined with counterpart \( \varphi' = q \rightarrow \tilde{\epsilon} \rightarrow \tilde{\gamma}_{s_\mu} r_\mu_o(\tilde{x}).\tilde{x}' \) via \( \Pi' \cup \Pi'' \) to a reservation \( \langle \pi, \varphi' \rangle \) in \( \mathbb{g}_N^\emptyset \) to which the induction hypothesis applies.

(b) In the case of new \( \pi, \pi \) cannot be initially new since initially new ownership paths do not reach region objects \( \mathbf{o} \tilde{x} \tilde{x}' \in \mathcal{R} \) or beyond and thus cannot combine with new \( \leftarrow\rightarrow \)-paths. Internally new \( \pi \) can combine with new \( \varphi = \mathbf{o} \tilde{x} \tilde{x}' \rightarrow \tilde{\epsilon} \rightarrow \tilde{\gamma}_{s_\mu} \omega \) via \( \Pi \). If \( \varphi \) is of the third kind in Lemma 15, it is the combination of an old \( \leftarrow\rightarrow \)-path \( \varphi_1 = \mathbf{o} \tilde{x} \tilde{x}' \rightarrow \tilde{\epsilon} \rightarrow \tilde{\gamma}_{s_\mu} \omega \) and a \( \leftarrow\rightarrow \)-path \( \varphi_2 = \mathbf{o} \tilde{x} \tilde{x}' \rightarrow \tilde{\epsilon} \rightarrow \tilde{\gamma}_{s_\mu} \sigma(\omega) \) via \( \Pi' \cup \Pi'' \), where \( \tilde{\gamma} = \tilde{\gamma}_1 \cdot \tilde{\gamma}_2 \). All edges of \( \varphi_1 \) must also be contained in \( \pi \) with target \( \mathbf{o} \tilde{x} \tilde{x}' \) and \( \mathbf{o} \tilde{x} \tilde{x}' \) with counterpart \( \varphi'_2 = r_\mu_o(\tilde{x}).\tilde{x}' \rightarrow \tilde{\epsilon} \rightarrow \tilde{\gamma}_{s_\mu} \omega \) via \( \Pi' \cup \Pi'' \). Hence their sources and targets are already covered. The reservation \( \langle \pi, \varphi_2 \rangle \) with just \( \varphi_2 \) 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 \( \leftarrow\rightarrow \)-paths of the first kind: Its counterpart \( \varphi'_2 \) combined with internally new \( \pi \)'s witness \( \sigma(\pi) = \sigma(q) \rightarrow^\pi \mathbf{r} \mu_o(\tilde{x}).\tilde{x}' \) to a reservation \( \langle \sigma(\pi), \sigma(\varphi) \rangle \) in \( \mathbb{g}_N^\emptyset \) to which the induction hypothesis applies.

In both cases, the \( \leftarrow\rightarrow \)-path in the respective reservation \( \langle \pi, \varphi' \rangle \) or \( \langle \sigma(\pi), \varphi' \rangle \) has the same source \( \mathbf{o} \) and initial edge \( h_1 \) like \( \pi \). Consider what all this means for the objects \( u \) in new \( \varphi \)'s path-base \( \Pi \):

1. If \( u \) is the source or target of a handle \( h \notin \Pi_1 \cup \Pi_0 \) then its precursor \( \sigma(u) \) is the source or target, respectively, of \( \sigma(h) \) in \( \Pi' \) and thus covered by the induction hypothesis. In the case of \( u \neq \mathbf{o} \tilde{x} \tilde{x}' \), \( \sigma(u) = u \), so that the induction hypothesis covers \( u \) directly: \( u \) is immutable, in \( RFree(h_1) \) or in \( RSub_H(\hat{\rho}) \). Otherwise, \( u = \mathbf{o} \tilde{x} \tilde{x}' \) such that \( \sigma(u) = r_\mu_o(\tilde{x}).\tilde{x}' \). But what the induction hypothesis guarantees for \( \sigma(u) \) in \( \mathbb{g}_N^\emptyset \), also holds for \( u \) in \( \mathbb{g}^\emptyset \), as explained at the beginning of the \{new\}-case.

2. If \( u \) is the source \( r \) of the initial handle \( h_\alpha \) in an initially new ownership path \( \hat{\pi} \in \Pi_N \), then it is covered as the source of \( r_\mu_o(\hat{\pi}) \mathbf{r} \mu_o(\tilde{x}).\tilde{x}' \) in \( \varphi' \)'s path-base \( \Pi' \cup \Pi'' \).

3. The other objects in initially new ownership paths \( \hat{\pi} \in \Pi_N \) and dummy edges \( \hat{\pi} \in \Pi_0 \) are region objects \( u = \mathbf{o} \tilde{x} \tilde{x}' \) with \( \mu_o(\tilde{x} \tilde{x}') = \beta \). As in the case of objects in internally new ownership paths, If \( \mu_o(\tilde{x}') = \text{read} \) then \( u \) is immutable. And if \( \mu_o(\tilde{x}') \in \{\text{free, rep}\} \), the possibilities which the induction hypothesis allows for \( r \) in \( \Pi'' \) (see above) imply that \( u \) is immutable like \( \mathbf{r} \) or, in the \text{rep} case, in \( RSub_H(\hat{\rho}) \) like \( r \), or, in the \text{free} case, in \( RSub_{H \cup \{h_\alpha\}}(\hat{\rho}) \) with \( repdn(h_\alpha) \) if \( r \) is in \( RSub_H(\hat{\rho}) \).

Observe that in all cases of \( u \in RSub_H(\hat{\rho}) \) or \( RSub_{H \cup \{h_\alpha\}}(\hat{\rho}) \), the handle \( h \) guaranteed to be in the old reservation for every \( h \in H \) exists also in \( \langle \pi, \varphi \rangle \): If \( h \) is in \( RSub_H(\hat{\rho}) \) or \( RSub_{H \cup \{h_\alpha\}}(\hat{\rho}) \), then \( h \) is in \( RSub_H(\hat{\rho}) \) or \( RSub_{H \cup \{h_\alpha\}}(\hat{\rho}) \).
can be a dummy edge, so that it cannot be in $\Pi''$. Hence non-dummy edge $h$ of the proof. The substep that decreases the multiplicity of the old handle at the left-hand side location is the same as in the case of $\{\text{iJ}T\} / \{\text{ifJ}\}$-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 $\tilde{c}$ ensures that $\tilde{\mu} \leq_m \tilde{\mu}'$. We need to look here only at one elementary conversion substeps $g'' = g' \oplus c \frac{\mu}{\mu'} o \ominus c \frac{\mu}{\mu'} o$, i.e., the substitution of an edge $h'_o = c \frac{\mu}{\mu'} o$ for an edge $h_o = c \frac{\mu}{\mu'} o$ with $\mu \leq_m \mu'$.

First, all ownership paths $\pi \in \text{PAP}_{g'}(o, \mu, q, \gamma)$ have a precursor $\pi' = \sigma(\pi) \in \text{PAP}_{g'}(o, \mu', q, \gamma)$ of the same mode $\mu' = \mu$ or directly compatible mode $\mu' \leq_m \mu$ (Lemma 7), where $\sigma = [h_o / h'_o]$ substitutes $h_o$ handles in in $\sigma$’s path-base to $h'_o$ in its precursor’s path-base. By the definition of $\leq_m$, the mode $\mu \geq_m \mu'$ cannot be free and can be rep only if $\mu'$ was free. Hence neither $h'_o$ nor—by the nesting constraint on valid modes—any of its extensions can be free. All free paths $\pi$ in $g'^{\circ}$ are unchanged ($\pi' = \pi$), and all rep paths $\pi$ 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 $\cap$-paths $\varphi = q \cap \omega \cap \sigma$ via $\Pi$ have a precursor $\varphi' = q \cap \omega \cap \sigma(\Pi)$ (Lemma 16). Hence for all reservations $(\pi, \varphi)$ in $g'^{\cap}$, there was already a reservation $(\pi', \varphi')$ of the same or directly compatible mode in $g'^{\cap}$.

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 $u \in \text{RFree}(h)$ remain so, change to $u \in \text{RSub}(c)$ (if $h = h_o$ was free and $h'_o$ is rep), or $u$ becomes immutable. Old cases of $u \in \text{RSub}(o)$ may be preserved, or the ownership reservation connection from $o$ to $u$ was severed, or a free reservation with initial $h_o \in H$ in the connection was converted to a rep reservation. In the latter case, $u \in \text{RSub}(o)$ 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 $v \in \text{RSub}(o)$ and $u = v$ or $u \in \text{RSub}(v)$ in $g'^{\circ}$ for some for $h' \subseteq H$.  

<table>
<thead>
<tr>
<th>in $g'^{\circ}$</th>
<th>in $g'^{\cap}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\neg \text{rmut}(u)$</td>
<td>$\text{rmut}(u)$</td>
</tr>
<tr>
<td>$u \in \text{RFree}(h)$</td>
<td>$u \in \text{RFree}(h) \lor u \in \text{RSub}(c) \lor \neg \text{rmut}(u)$</td>
</tr>
<tr>
<td>$u \in \text{RSub}(o)$</td>
<td>$u \in \text{RSub}(h_o) \lor \neg \text{rmut}(u)$</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\text{Figure 6.14: Change of relationships in conversion steps}
\end{align*}
\]
The loss of reservation makes $v$ immutable, perhaps also $u$.

**Second.** For all objects $u$ in each precursar $(\sigma(\pi), \varphi)$ with $\varphi$ via $\Pi$, the induction hypothesis allows three possibilities. Note that these are the same objects as in $(\pi, \varphi)$ since the substitution of $h_0$ for $h'_{0}$ does not change which objects are in the paths.

1. If $u$ was immutable, then it remains so in $g''_{0}$.
2. If $u$ was in $RFree(h_{1})$ then it remains so, or becomes immutable, or switches to $u \in RSub(o)$ if $h_1 = h_0$, so that $\pi$’s source $o$ is $c$.
3. There was an $\hat{q}$ that was $o$ or immutable and $u \in RSub(\hat{\eta}(\hat{q}))$ in $g''_{0}$ with a $h$ in $\hat{\Pi}$ or $\Pi$ for every $h \in H$ such that $h = h \cup obsbd(h, h)$. In $g''_{0}$, $u$ is immutable or $u \in RSub_H(h_{0})(\hat{q})$ or there is an immutable $\hat{q}'$ with $u \in RSub_{H'}(\hat{q}')$ and $H' \subseteq H \setminus \{h_{0}\}$.

In other words, $u$ is immutable or there is an object $\hat{q}'$ that is $o$ or immutable and $u \in RSub_{H'}(\hat{q}')$ with $H' \subseteq H \setminus \{h_{0}\}$. Since $H'$ contains only edges of $H$ and does not contain the changed edge $h_{0}$, for every $h \in H'$ there is a $\sigma(h)$ in $\hat{\Pi}$ or $\Pi$ such that $\sigma(h) = h = h \cup obsbd(\sigma(h), h)$.

**Third.** At the end of all elementary conversion substeps, the fully converted handle $h''_{0}$ is stored at location $\ell$. Storage is relevant for the invariants’ preservation in two cases: (1) $\ell$ is a field location and $h''_{0}$ is the uncaptured handle in a reservation $(\pi, \varphi)$ in a case with $obsbd(h_{0}, h)$. 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) $\ell$ is a field location and $h''_{0}$ is the uncaptured free handle with $u \in hRSanc_{H}(h''_{0})$ and $obsbd(h, h''_{0})$. Assignment to a field requires the top-level execution of a mutator by $h''_{0}$’s source $r$ (shallow state encapsulation). The observer invocations to which $h$ is local must be below that. Mutator execution in $r$ presupposes a sequence of mutator calls to $r$ along ownership paths starting with a free path $\pi_{0}$ and followed by a sequence of rep paths to $r$ (Theorem 8). The observer invocations with $h$ must be below all these mutator invocations. In particular, the initial free edge $h_{0}$ of $\pi_{0}$ that initiated the mutator calls is local to an invocation not below $h$. That is, $obsbd(h, h_{0})$. The corresponding membership $u \in hRSanc_{H}(h_{0})$ with $H' = \{h''_{0}\} \cup H \subseteq fgr_{on}^{\sigma}(s^{\pi_{0}})^{\circ}$ holds since the capturing of $h''_{0}$ in $r$ connects $u \in hRSanc_{H}(h''_{0})$ with the ownership paths leading to $r$. This shows that also the final storage substep preserves the invariants.

**{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 $\{if\} / \{if\}$-steps, the intermediate removal step $g' = g_{N} \oplus s(im(\pi'))$ preserve the invariants. The interesting aspect of return steps is the transfer $g = g' \oplus r \overset{h_{0}}{\rightarrow} o \oplus s \overset{hr}{\rightarrow} r \oplus s \overset{hr \circ h_{0}}{\rightarrow} o$, of the result value, i.e., the substitution of the imported edge $h'_{0} = s \overset{hr \circ h_{0}}{\rightarrow} o$ for the exported edge $h_{0} = r \overset{h_{0}}{\rightarrow} o$ and the call-link $h_{r} = s \overset{hr}{\rightarrow} r$.

**First.** There are no really new $\equiv$-paths (Lemma 17). And the only really new ownership paths $\pi$ are $s$’s initially new paths $\pi$ of a free mode $\mu_{r} \circ \mu$ with a counterpart $\epsilon_{po}(\pi)$ of free mode $\mu$ (Lemma 8). The new ownership paths $\pi$ that are not really new are initially new and have a precursor $h_{r} \cdot \epsilon_{po}(\pi)$.
This has the following implications summarized in figure 6.15: Old cases of immutability are preserved. Old cases of \( u \in RFree(h) \) with \( h \not\in \{ h_o, h_r \} \) are preserved unless the ownership reservation connection through \( h \) to \( u \) is severed by the removal of \( h_r \) and \( h_o \). And old cases of \( u \in RFree(h) \) with \( h = h_o \) or \( h_r \) are succeeded by \( u \in RFree(h_o') \) unless the free ownership reservation on \( u \) through the handle is lost. Since there are no new ownership reservations except for \( h_o' \)-based ones, this loss would mean that \( u \) is immutable now. In short, \( u \in RFree(h) \) in \( g^o \) entails \( \neg rmut(u) \) or \( u \in RFree(\delta(h)) \) in \( g^o \). Old cases of \( u \in RSub_H(o) \) may be preserved (if it contains no \( h_o \in H \)), or the ownership reservation connection from \( o \) to \( u \) was severed, or a free reservation in the connection with free path \( \pi = h_o \cdot \tilde{\pi} = \epsilon\rho\sigma(\pi') \) or \( h_r \cdot h_o \cdot \tilde{\pi} = h_r \cdot \epsilon\rho\sigma(\pi') \) was shortened to a free reservation with initially new \( \pi' = h_o' \cdot \tilde{\pi} \). In the latter case, \( u \in RSub_H(o) \) 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 \( v \in RSub_H(v) \) and \( u = v \) or \( u \in RSub_H(v) \) in \( g^o \) for some for \( H', H'' \subseteq H \). The loss of reservation makes \( v \) immutable, perhaps also \( u \).

Second. Let \( \sigma \) be the substitution \( [h_r \cdot h_o/h_o'; h_{r-1}/h_{o-1}'] \) in case of \( h_r \cdot h_o = co<> \), and \( [h_r \cdot h_o/h_o'] \) otherwise. By Lemmas 17 and 8, any reservation \( \langle \pi, \varphi \rangle \) in \( g^o \) with \( \pi = h_1 \cdot \tilde{\pi} \in PAP(o, \mu, q, \gamma) \) and \( \varphi = q \rightarrow \gamma \) entails a reservation \( \langle \tilde{\pi}, \tilde{\varphi} \rangle \) in \( g^o \) with \( \tilde{\pi} = h_1' \cdot \tilde{\pi}' \in PAP(o', \mu', q, \gamma) \) and \( \tilde{\varphi} = q \rightarrow \gamma \) via \( \sigma(\Pi) \) where normally \( \tilde{\pi} = \sigma(\pi) \) with source \( o' = o \) and initial edge \( h_1' = h_1[h_r/h_o] \), except in case of free initially new \( \pi \) without precursor, where \( \tilde{\pi} = \epsilon\rho\sigma(\pi) \) with source \( o' = r \) and initial edge \( h_1' = h_o \). To the objects \( u \) in this reservation, the induction hypothesis applies. It covers all objects \( \pi \) in \( g^o \) since substitution \( \sigma \) never removes any objects from a path (but at most adds object \( r \)) path-base \( \Pi \) contains no more objects than \( \sigma(\Pi) \), \( \pi \) with \( \tilde{\pi} = \sigma(\pi) \) contains no more objects than \( \tilde{\pi} \), and initially new paths \( \pi = h_o' \cdot \tilde{\pi} \) contain non-initially no more objects than their precursor \( \tilde{\pi} = \epsilon\rho(\pi) = h_o \cdot \sigma(\tilde{\pi}) \) contained non-initially. The induction hypothesis allows three possibilities for \( u \):

- \( u \) was immutable. As shown above, this is preserved.
- \( \tilde{\pi} \) is free and \( u \in RFree(h_1) \) in \( g^o \). Then in \( g^o \), as shown above, \( u \) is immutable or \( u \in RFree(h_o) \) with \( h_1' = \delta(h_1') \). If \( \pi \) is not initially new, then \( h_1' = h_1 \not\in \{ h_r, h_o \} \), so that in \( g^o \) we have \( u \in RFree(h_o) = RFree(h_1) = RFree(h_1) \), as desired. And if \( \pi = h_o' \cdot \tilde{\pi} \) is initially new—with equivalent precursor \( \tilde{\pi} = h_r \cdot \epsilon\rho(\pi) \) or with free counterpart \( \tilde{\pi} = \epsilon\rho(\pi) = h_o \cdot \tilde{\pi}' \)—in \( g^o \) we have \( u \in RFree(h_o') = RFree(h_1) = RFree(h_1) \), 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 \( g^o \) with a \( h \) in \( \hat{\pi} \) or \( \sigma(\Pi) \) for every \( h \in H \) such that \( h = h \lor obsbd(h, h) \). In \( g^o \), \( u \) is immutable or \( u \in RSub_{\hat{\pi}(\Pi)}(\hat{q}) \) or there is an immutable \( \hat{q} \) with \( u \in RSub_{\hat{\pi}(\Pi)}(\hat{q}) \) and \( H' \subseteq \hat{\sigma}(H) \). Since there are no really new coreceived rep paths \( \pi, \hat{\pi} \) is equivalent to \( \pi \), 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 \( h' \in H' \) by looking at all the \( h \in H \). All of these are either in \( H' \) and thus exist in \( g^o \), or they are \( h_o \) or \( h_r \). 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 \( \{ \hat{\pi} \} \cup \Pi \), there must be a corresponding path \( \hat{\pi}'' \) with \( \hat{\pi}' = \sigma(\hat{\pi}'') \) or \( \epsilon\rho\sigma(\hat{\pi}'') \).

(i) If \( h \) occurs in \( \hat{\pi}' \) not as part of a \( \sigma \)-replaced subsequence, this means it exists also in the \( g^o \)-path \( \hat{\pi} \). In the case of \( h = h \), this means that free \( h \) cannot be \( h_o \) or \( h_r \), since these would be reduced to zero multiplicity (Theorem 7). Hence \( \hat{\sigma}(h) = h = h \) is in \( \hat{\pi} \) as necessary for \( \hat{\sigma}(h) = h' \in H' \). In the case of \( obsbd(h, h), obsbd(h, \hat{\sigma}(h)) \) trivially follows if \( h \notin \{ h_r, h_o \} \). This is what we need for \( \hat{\sigma}(h) = h' \in H' \). Otherwise \( \hat{\sigma}(h) = h' = h'_o \in H' \) is at the new top-most call-level in \( g^o \) and unchanged \( h \) cannot be above it, so that necessarily \( obsbd(h, h'_o) \), i.e., \( obsbd(h, h') \).

(ii) \( h \) cannot be \( h_r^{-1} \) nor \( h_o^{-1} \), since \( h \) is the free \( h \) or a non-co-handle with \( obsbd(h, h) \). If \( h = h_r \) or \( h_o \) in a \( \sigma \)-replaced subsequence \( h_r \cdot h_o \) in \( \hat{\pi}' \) then \( \hat{\pi}'' \) contains \( h'_o \) instead. In the case with \( h = h \), this means it contains \( h' = \hat{\sigma}(h) = h'_o \), as desired.

(iii) In the \( obsbd(h, h) \) case, if \( h = h_r \), then the invocation containing \( h = h_r \), i.e., the invocation to which the computation returns, is an observer. Since \( h'_o \) is contained in the same invocation as \( h \), and since \( \hat{\sigma}(h) \) is \( h \) or \( h'_o \), we have the necessary \( obsbd(h'_o, \hat{\sigma}(h)) \).

(iv) In the \( obsbd(h, h) \) case, if \( h = h_o \), then \( obsbd(h, h) \) for \( h = h_o \) 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 \( obsbd(h'_o, \hat{\sigma}(h)) \) since either unchanged \( \hat{\sigma}(h) = h \) is still at the same call-level or \( \hat{\sigma}(h) = h'_o \) is at \( h'_o \)'s call-level \( n \). And if a mutator is executing at call-level \( n \), it means that \( h \) cannot be at a level below \( h = h_o \)'s level \( n + 1 \). But if it is local to the terminated call-level \( n + 1 \), the intermediate step \( g'' = g \ominus s(\text{im}(\eta')) \) destroys free \( h \) since its multiplicity was one by the reserved ownership assumption. This violates the assumption that \( h \in H \) exists still in \( g^o \) with the exception only of \( h = h_o \) and \( h_r \).

\{\text{call}\} As always, we decompose a \{call\}-step into substeps that convert the handle arguments’ mode to the import \( \mu_r \cdot \mu_o \) of the receiver’s parameter modes \( \mu_o \)—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 \( g'' = g' \ominus s \cdot \text{hr} \cdot h_o \cdot o \ominus r \cdot o \) of supplying one mode-adapted argument. This is trivial if the received handle \( h_o = r \cdot o \) is a nil-handle or if it is not new but existed already in \( g'' \). We are concerned
only with new handles $h_0$ and their inverses $h_0^{-1}$ (if they are co), if they are edges in $g''^\otimes$ but not in $g''^\otimes$.

First, all $\mu$-reservations $\langle \pi, \varphi \rangle$ of $o$ on $\omega$ in $g''^\otimes$ have a precursor $\langle \pi', \varphi' \rangle$ of the same mode in $g''^\otimes$—with the exception that $o$’s reservations with free initially new path $\pi$ have a counterpart $(\text{sent}(\pi), \varphi')$ that reserved the free ownership of $r$ on $\omega$ in $g''^\otimes$: If $\varphi$ is new with a qbridge as precursor that contains quadruples, then $\pi'$ is its last ownership path and $\varphi'$ its last $==>$-path. If there are no quadruples but $\pi$ has a counterpart with a bridge with a triple series then $\pi'$ is its last ownership path and $\varphi'$ is the closure of its last $==>$-path under $\varphi$’s precursor $==>$-path. Otherwise, $\pi' = \pi$ and $\varphi' = \varphi$.

Consequently, all cases of $\neg \text{rmut}(u)$ are preserved.

On the other hand, no old $==>$-path $\varphi$ gets really lost—only its path-base might change—and no ownership path $\pi$ gets really lost—but always has an equivalent successor—with the sole exception that the free paths starting with $h_0'$ are the witness $\text{sent}(\pi)$ of a free initially new path $\pi$ that succeeds them: If the removed handle $h_0'$ or its inverse occurred in an ownership path $\pi = \pi'$ or in an ownership path or association path $\pi \in \Pi'$ of old reservation $\langle \pi', \varphi' \rangle$ then it can be expanded with substitution $\sigma = [h_r \bullet h_0/h_0', h_o^{-1} \bullet h_r^{-1}/h_o'^{-1}]$. The corresponding path $\sigma(\pi)$ is equivalent to $\pi$ except if it is a free path starting with $h_0'$. First, $h_0'$ or $h_0'^{-1}$ may be in ownership or association path $\pi$ since they are co-handles, i.e., $\mu_r \bullet o = \text{co}<>$. Then the typeability of redex $\hat{e}$ ensures that $h_r$ and $h_0$ are co as well. In this case, the expansion always works. Second, $h_0'$ can be in ownership or association path $\pi$ since it is an association handle of mode $\beta<>$ or since it is extended to a $\beta$-path $\hat{\pi}' = h_0' \bullet \hat{\pi}''$ that is a subsequence of $\pi$: $\mu_r \bullet o \mu_o(\hat{\alpha}) = \beta$. Then typeability ensures a decomposition $\hat{\alpha} = \hat{\alpha}_1 \bullet \hat{\alpha}_2$ such that $\mu_o(\hat{\alpha}_1) = \gamma$ and $\mu_r$ is a rep or free mode with $\gamma = \hat{\mu}$ such that $\hat{\mu}(\hat{\alpha}_2) = \mu_r(\gamma.\hat{\alpha}_2) = \beta$. In this case, $h_r \bullet h_0 \bullet \sigma(\hat{\pi}'') = \sigma(\hat{\pi}')$ is a $\beta$-path that can substitute $\hat{\pi}'$ in $\hat{\pi}$. Third, $h_0'$ can be the initial edge of ownership path $\hat{\pi} = h_r' \bullet \hat{\pi}'$ of a shape $s$ $\overset{\text{free}<>}{\mu_r \bullet o} \bullet \overset{-\hat{\alpha}_1 \bullet -\hat{\alpha}_2}{\text{co}<>} \bullet$, i.e., $\mu_r \bullet o \mu_o(\hat{\alpha}) = m \in \{\text{free}, \text{rep}\}$. Either $\mu_o(\hat{\alpha}) = \text{free} = m$ (there is no rep in parameter mode $\mu_o$), i.e., $\mu_o = \epsilon$ and $\mu_o = \text{free}<>$. Then $\hat{\pi}$ is the free witness $\text{sent}(\pi)$ of a free initially new path $\pi$ (and $\sigma(\hat{\pi})$ is no potential access path). Or $\mu_o = \text{co}<>$. Then $h_r \bullet h_o = s$ $\overset{\text{free}<>}{\mu_r \bullet o} \bullet \overset{-\hat{\alpha}_1 \bullet -\hat{\alpha}_2}{\text{co}<>} \bullet$. $\hat{\pi}$’s substitute $\sigma(\hat{\pi})$ is $h_r \bullet h_o \bullet \sigma(\pi')$. Or there is a decomposition $\hat{\alpha} = \hat{\alpha}_1 \bullet \hat{\alpha}_2$ such that $\mu_o(\hat{\alpha}_1) = \gamma$ and $\mu_r$ is a rep or free mode with $\gamma = \hat{\mu}$ such that $\hat{\mu}(\hat{\alpha}_2) = \mu_r(\gamma.\hat{\alpha}_2) = m$. Then $h_r \bullet h_o \bullet \sigma(\pi')$ is a substitute $\sigma(\hat{\pi})$ for $\hat{\pi}$.
Consequently, all old rep reservations \( \langle \pi, \varphi \rangle \) with \( \varphi \) via \( \Pi \) have an equivalent successor \( \langle \sigma(\pi), \varphi \rangle \) with \( \varphi \) via \( \sigma(\Pi) \), so that old cases of \( u \in RSub_0(o) \) are preserved (cf. fig. 6.16). And all free reservations \( \langle \pi, \varphi \rangle \) with \( \varphi \) via \( \Pi \) either have an equivalent successor \( \langle \sigma(\pi), \varphi \rangle \) with \( \varphi \) via \( \sigma(\Pi) \) of \( r \) on the same object if \( \mu_o = co<> \), or \( \pi = sent(\pi') \) and the reservation of \( s \) switches to a free reservation \( \langle \pi', \varphi \rangle \) with \( \varphi \) via \( \sigma(\Pi) \) of \( r \) on the same object if \( \mu_o = free<...> \). Hence all cases of \( u \in RFree(h) \) with \( h \neq h'_o \) and of \( u \in RSub_H(o) \) with \( h'_o \notin H \) are preserved. If \( \mu_o = co<> \), all case of \( u \in RFree(h'_o) \) and of \( u \in RSub_H(o) \) with \( h'_o \in H \) become cases of \( u \in RFree(h_r) \) and of \( u \in RSub_H(o) \) with \( h_r \in H' = H \setminus \{h'_o\} \cup \{h_r\} \) because they were based on a free path \( \pi = h'_o \cdot \pi' \) that has the equivalent successor \( \sigma(\pi) = h_r \cdot h_o \cdot \sigma(\pi') \). If \( \mu_o \neq co<> \), all case of \( u \in RFree(h'_o) \) become cases of \( u \in RFree(h_o) \) because they are based on a free path \( \pi = sent(\pi') \) with free successor \( \pi' \). In order to say what becomes of \( u \in RSub_H(o) \) with \( h'_o \in H \) if \( \mu_o \neq co<> \), we have to consider what happens with the correlations to rep and association roles on free paths \( \pi = sent(\pi') \) where \( \pi' \) is free and of mode \( \mu_r \). If \( \pi' \)'s mode \( \mu_r.o.\mu_o \) contained such correlations, i.e., if \( \mu_r.o.\mu_o(\gamma_1, \gamma_2) \in \{\text{rep}\} \cup A \) for some \( \gamma_1, \gamma_2 \), then \( \mu_o \) contained a correlation to some association role \( \gamma \), i.e., \( \mu_o(\beta.\alpha_1) = \gamma \), and \( \mu_r \) contains a correlation \( \gamma = \hat{\mu} \gamma \) to a mode \( \hat{\mu} \) that is \( \alpha \) or from which \( \alpha \) can be extracted, i.e., with \( \hat{\mu}(\gamma_2) = \mu_r(\gamma.\gamma_2) = \alpha \). That is, also \( h_r \) and \( \pi' \) 's successor \( \pi' \) have correlations to a rep or association role. Since \( \mu_o \) contains the association mode \( \gamma \cdot <> \), i.e., \( \mu_o(\beta.\alpha_1) = \gamma \), and the corresponding \( \mu_r.o.\mu_o(\beta.\alpha_1) \) is not read, the typeability ensures that \( \mu_r \) is rep or free. Hence any free reservation starting with \( h'_o \in H \) can be expanded to a rep \( h_r \) or a free \( h_r \) and a free reservation starting with \( h_o \). Consequently, \( u \in RSub_H(o) \) with \( h'_o \in H \) is succeeded by \( u \in RSub_H(o) \) with \( H' = H \setminus \{h'_o\} \cup \{h_r, h_o\} \) if \( h_r \) is rep and \( H' = H \setminus \{h'_o\} \cup \{h_r, h_o\} \) if \( h_r \) is free.

Second. Consider reservations \( \langle \pi, \varphi \rangle \) in \( g''^o \) where \( \pi \) is unchanged or internally-only new and \( \varphi \) via \( \Pi \) is unchanged. In the former case, \( \pi \) is its own precursor \( \pi' \). In the latter case, \( \mu_o = \mu_r = co<> \) and there is a precursor \( \pi' = \pi[h_r^{-1} \cdot h'_o/h_o, h_o^{-1} \cdot h_r/h_o^{-1}] \) with which \( \pi \) shares the initial edge \( h_1 \). Observe that the initial edge of \( \pi \) and \( \pi' \) are the same. The only difference in the set of intermediate objects is that \( \pi' \) contains \( s \) which \( \pi \) 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 \( g''^o \).

The cases of \( \neg \text{mut}(u) \) and of \( RFree(h_1) \) are preserved, as shown above. Notice that free \( h_1 \) and \( h \) cannot be \( h'_o \): If \( \pi \) is internally-only new, \( h'_o \) is co. And an unchanged path \( \pi' = \pi \) or \( \pi \in \Pi \) cannot contain a \( h'_o \) that is free since the multiplicity of a free \( h'_o \) is by Unique Head (Theorem 7) reduced to zero in \( g''^o \). Finally, there could be a \( \hat{q} \) that is \( o \) or is immutable and \( u \in RSub_H(q) \) in \( g^o \) with a \( h \) in \( \hat{\pi} \) or \( \Pi \) for every \( h \) in \( H \) such that \( h = h \) or \( \text{obsbd}(h, h) \). As shown above, we have \( u \in RSub_H(q) \) in \( g''^o \) for some \( \hat{q} \) that is still \( o \) or still immutable. \( H' \) is \( H \) with any \( h'_o \) in it expanded to \( h_r \) and/or \( h_o \) (depending on which of them is free). It remains to check all the \( h \in H' \): If \( h'_o \) was in \( H \) then \( h = h_o \) and \( h_r \) in \( H' \) are covered: The case with \( h = h'_o \) in \( \pi \) or \( \Pi \) cannot apply since a free \( h'_o \) is reduced to zero multiplicity in \( g''^o \) by Unique Head (Theorem 7). And the case of \( \text{obsbd}(h, h'_o) \) means that also \( \text{obsbd}(h, h_r) \)
(for \( h = h_r \in H' \)) since \( h'_o \) and \( h_r \) are at the same call-level, and \( \text{obsbd}(h, h_o) \) (for \( h = h'_o \in H' \)) since \( h_o \) is one call-level above \( h'_o \). Other cases of \( h \in H' \) if \( h'_o \in H \), and all cases of \( h \in H' \) if \( h'_o \in H \) presuppose \( h \in H \). For every \( h \in H \) the necessary \( h \) in \( \Pi \) is guaranteed with \( h = h \) or \( \text{obsbd}(h, h) \).

**Third.** Consider reservations \( \langle \pi, \varphi \rangle \) in \( g^{\#} \) where \( \pi \) is *initially new* and \( \varphi \) via \( \Pi \) is unchanged. Then \( \pi = h_o \cdot \pi_1 \cdot \pi_2 \), and in \( g^{\#} \) there was a witness \( \text{wit}(\pi) = h'_o \cdot \pi_1 \cdot \pi_2' \in \text{PAP}^{\#}(s, \mu_\varphi, \mu_0, q_0, \delta_0) \) and a \( \delta_0 \)-bridge via \( \Pi \) from \( q_0 \) to \( \pi \)'s target \( q_0 \). The typeability of redex \( \pi \) ensures that \( \pi \) is not rep but free. Because of the nesting constraints on valid modes, this means that \( \mu_\varphi \) is a free mode, i.e., \( h_o \) is free, and that \( h_o \) can only be extended by co-edges. Hence \( \pi_1 \) must a sequence of co-edges, and \( \pi_2 \) must be empty and the bridge be trivial since otherwise \( \pi_2 \) would start with (free) \( h_o \). Consequently, \( \text{wit}(\pi) \) has the same target as \( \pi \) and the same edges except for the initial edge \( h'_o \) in place of \( h_o \). Since \( \text{wit}(\pi) \) must be free like \( \pi \) is, all non-initial objects \( u \) in \( \langle \pi, \varphi \rangle \) are covered as non-initial objects in \( \langle \text{wit}(\pi), \varphi \rangle \). For this free reservation, the induction hypothesis allows the following possibilities:

- \( u \) was immutable in \( g^{\#} \). Hence it still is so in \( g^{\#} \).
- \( u \in \text{RFree}(h'_o) \) in \( g^{\#} \) since \( h'_o \) is \( \text{wit}(\pi) \)'s initial edge. In \( g^{\#} \), as shown above, we then have \( u \in \text{RFree}(h_o) \) since \( h_o \) is free, not co. This is just right since \( h_o \) is \( \pi \)'s initial edge.

**Fourth.** Consider reservations \( \langle \pi, \varphi \rangle \) in \( g^{\#} \) where \( \pi \) is *internally really new* and \( \varphi \) via \( \Pi \) is unchanged. Then \( \pi = h_1 \cdot \pi = \pi_1 \cdot \pi_2 \), and in \( g^{\#} \) there was a witness \( \text{wit}(\pi) = h_1 \cdot \pi = \pi_1 \cdot \pi_2' \in \text{PAP}^{\#}(s, \mu_\varphi, \mu_0, q_0, \delta_0) \) and a \( \delta_0 \)-bridge from \( q_0 \) to \( \omega \) via \( \Pi' \) such that \( \{ \pi_2 \} \cup \Pi_1 \cup \Pi_2 \cong \Pi \cup \Pi_0 \). From these, the following reservations \( \langle \varphi, \varphi \rangle \) can be constructed in \( g^{\#} \): If there are triples in the bridge, then witness \( \text{wit}(\pi) \) and the bridge’s initial \( \leftarrow \)-path \( \varphi_0 = q_0 \xrightarrow{\delta_0} \gamma_0 \) \( \omega_1 \) via \( \Pi_0 \) constituted a reservation \( \langle \text{wit}(\pi), \varphi_0 \rangle \), and each triple in the bridge means two reservations \( \langle \pi_i, \omega_i \xrightarrow{\gamma_i} \omega_i \rangle \) and \( \langle \pi_i, \varphi_0 \rangle \). The last \( \leftarrow \)-path \( \varphi_n = q_n \xrightarrow{\delta_0} \gamma_n \omega_n \) via \( \Pi_n \) combined with \( \varphi \) to \( \varphi_n = q_n \xrightarrow{\delta_0} \gamma_n \omega \) via \( \Pi_n \cup \Pi \) (Lemma 12). It combined to a reservation \( \langle \pi_n, \varphi_n \rangle \) with the bridge’s last ownership path \( \pi_n = n \xrightarrow{\delta_n} q_n \omega_n \). If there are no triples in the bridge, then \( \Pi' = \Pi_0 \). \( \varphi_0 \) and \( \varphi \) combined to \( \varphi_n = q_n \xrightarrow{\delta_0} \gamma_n \omega \) via \( \Pi_0 \cup \Pi = \Pi_n \cup \Pi = \Pi' \cup \Pi \). Hence in \( g^{\#} \) there was the \( \mu \)-reservation \( \langle \text{wit}(\pi), \varphi_n \rangle \) of \( o \) on \( \omega \).

If there are any triples in the bridge, the reserved ownership assumption in \( g^{\#} \) ensures that \( \text{wit}(\pi) \)'s source \( o \) is their source \( s \) and that all ownership paths have the mode of \( \text{wit}(\pi) \). Hence if \( \langle \pi, \varphi \rangle \) in \( g^{\#} \) is a free reservation then all corresponding reservations \( \langle \pi, \varphi \rangle \) were also free reservations. In each reservation \( \langle \pi, \varphi \rangle \) with \( \pi = h'_1 \cdot \pi \), the sources and targets \( u \) of all edges in \( \pi \) or \( \Pi \) are covered by the induction hypothesis. In case of \( \langle \pi, \varphi \rangle = \langle \text{wit}(\pi), \varphi_0 \rangle \) or \( \langle \text{wit}(\pi), \varphi_n \rangle \), the missing edge \( h'_1 \) is the initial edge \( h_1 \) of \( \pi \), and thus needs no coverage. Otherwise it is the initial edge \( h_r \) or \( h'_o \) of an ownership path in the bridge, which is skipped in the construction of the new path \( \pi \) and thus irrelevant for \( \langle \pi, \varphi \rangle \). (Of course \( h_r \) and \( h'_o \) will be contained in
\[\pi\] if they are contained elsewhere in the bridge: in one of the \(\equiv\)-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 \((\pi, \varphi), i.e., for all objects in \(\pi\) which need to be covered, the induction hypothesis allows the following possibilities:

- \(u\) was immutable in \(g^{\pi}\). Hence it still is so in \(g^{\nu}\).
- \((\pi, \varphi)\) is a free reservation with initial edge \(h_1'\) and \(u \in RFree(h_1')\). In the case of \((\pi, \varphi) = (\nu(p), \varphi_0)\) or \((\nu(p), \varphi_\alpha)\), \(h_1'\) is \(\pi\)'s unchanged initial edge \(h_1\). If free \(h_1'\) were \(h_1'\), then its multiplicity of one (the reserved ownership assumption) would be reduced to zero in \(g^{\nu}\), so that it could not be unchanged—a contradiction. But if \(h_1' \neq h_1'\), then \(u \in RFree(h_1')\) is preserved, i.e., \(u \in RFree(h_1)\).

Other cases of \((\pi, \varphi)\) presuppose triples in the bridge contains. The nesting constraint on modes ensures that the free internally really new \(\pi\) is made of an initial free \(h_1\) followed by co-edges. Hence the \(h_1\) in it must be co: \(\mu_0 = co<.\)

But then there is one \(\beta\) such that every ownership path \(\tilde{\pi} = \pi_1'\) in \((\pi, \varphi) = (\nu(p), \varphi_0)\) or \(\tilde{\pi} = \pi_1\) in \((\pi, \varphi) = (\nu(p), \varphi_\alpha)\) has shape \(s \xleftarrow{\mu_{10}} o \xrightarrow{\rho_{o}} \cdot -\beta - \cdot \cdot \). For the quadruple \((\nu(p), \varphi_0, \pi_1', \omega_1 \xleftarrow{\nu_{10}} \omega_1)\) of the first and second reservation, this means by the reserved ownership assumption that \(\nu(p)'s\) unchanged initial edge \(h_1\) must be \(h_1\) (since it cannot be \(h_1'\), just shown before). Hence if \((\pi, \varphi)\) is a reservation with \(\tilde{\pi}\) of the case with shape \(s \xleftarrow{\mu_{10}} r \xrightarrow{\rho_{o}} \cdot -\beta - \cdot \cdot \), then \(u \in RFree(h_1')\) means \(u \in RFree(h_1)\), and this is preserved since \(h_1 \neq h_1'\). And if \((\pi, \varphi)\) is a reservation with \(\tilde{\pi}\) of the case with shape \(s \xleftarrow{\mu_{10}} o \xrightarrow{\rho_{o}} \cdot -\beta - \cdot \cdot \), then \(u \in RFree(h_1')\) means \(u \in RFree(h_1')\).

As shown above, this is succeeded in \(g^{\nu}\) by \(u \in RFree(h_1) = RFree(h_1)\), since the alternative \(u \in RFree(h_1')\) is impossible with an \(h_1\) of mode \(\mu_0 = co<.\).

The other case applies to rep reservations. If, in this case, \(h_1\) is a free handle then there is an \(h'\) in \(\pi\) such that \(h' = h_1\) or \(obsbd(h', h_r)\): \(\nu(p) = \pi_1 \cdot \pi_2\) passes through \(r\) since the prefix \(\pi_1\) common with \(\pi = \pi_1 \cdot \pi_2\) ends in \(r\) (from where it continues with \(h_1\) initially in \(\pi_2\)). Hence for object \(r\) in rep reservation \((\nu(p), r, \lambda_0 \xleftarrow{\lambda_1} r, \lambda_0)\), the induction hypothesis allows the following possibilities: The case of immutable \(r\) cannot apply since it is target of uncaptured free \(h_r\). There must be an object \(\hat{\varphi}'\) that is \(o\) or is immutable such that \(r \in RSub_H(\hat{\varphi}')\). All ownership reservation sequences to \(r\) targeted by free \(h_r\) have to include a free reservation with initial \(h_r\) (the reserved ownership assumption). Hence \(h_r\) must be included in \(H\). But then we are guaranteed an \(h'\) in the rep reservation, i.e., in \(\nu(p)\), such that \(h' = h_r\) or \(obsbd(h', h_r)\). However, whether \(h'\) is free \(h_r\) or uncaptured \((obsbd(h, h))\), it cannot be in the dummy edge-path \(\pi_2\). But then \(h'\) is in \(\pi_1\) and thus in \(\pi = \pi_1 \cdot \pi_2\).

- There was a \(\hat{\varphi}'\) that was \(o\) or immutable and \(u \in RSub_H(\hat{\varphi}')\) in \(g^{\pi}\) with a \(h\) in \(\hat{\pi}\) or \(\hat{\Pi}\) for every \(h \in H\) such that \(h = h \vee obsbd(h, h)\). As shown above, we have \(u \in RSub_H(\hat{\varphi}')\) in \(g^{\nu}\) for some \(\hat{\varphi}'\) that is still \(o\) or still immutable. \(H'\) is \(H\) with
any \( h'_o \) in it expanded to \( h_r \) and/or \( h_o \) (depending on which of them is freet). It remains to check all the \( h \in H' \): (1) \( h = h_o \in H' \) is always in \( \langle \pi, \varphi \rangle \) since any internally really new \( \pi = \pi_1 \cdot \pi_2 \) contains \( h_o \) as the first element of its \( \pi_2 \)-segment.

(2) For \( h = h_r \in H' \), it was shown just before, there is an \( h' \) in \( \pi \) with \( h' = h_r \) or \( \text{obsbd}(h', h_r) \). (3) The remaining \( h \in H' \) were also in \( H \): For every \( h \in H \), there was the necessary \( h \) in \( \tilde{\pi} \) or \( \tilde{\Pi} \). This means that \( h \) was in witness \( \text{wit}(\pi) = \pi_1 \cdot \pi_2 \), or in an ownership path \( \pi'_r \) or \( \pi_r \), or in a \( \leftarrow \)-path \( \varphi_i \) via \( \Pi_i \) or \( \varphi_n \) via \( \Pi_n \cup \Pi \). In short, \( h \) was in \( \{ \pi_1 \cdot \pi'_2 \} \cup \Pi' \cup \Pi \), or formally, \( \{ h \} \subseteq \{ \pi_1, \pi'_2 \} \cup \Pi \). Since we know about \( \pi \)'s bridge that \( \{ \pi_2 \} \cup \Pi_\Lambda \cup \Pi_\Theta = \Pi' \cup \Pi_\Theta \), this means \( \{ h \} \subseteq \Pi \cup \{ \pi_1, \pi_2, \pi'_2 \} \cup \Pi_\Lambda \cup \Pi_\Theta \). Whether \( h \) is freet \( h_o \) or non-co and uncaptured (\( \text{obsbd}(h, h) \)), it cannot be in the dummy edge-path \( \pi'_r \) nor in dummy edge set \( \Pi_\Theta \), nor can it be an inverse co-edge \( h^{-1}_r, h^{-1}_o \in \Pi_\Lambda \).

(i) If \( h \) was in \( \pi_1 \) or \( \pi_2 \) or \( \Pi \), then it is in \( \langle \pi, \varphi \rangle \). Hence we have for \( h \in H' \) the desired handle \( h \) with still \( h = h \) or \( \text{obsbd}(h, h) \), respectively.

(ii) If, in the \( h = h \) case, \( h \in \Pi_\Lambda \) then it cannot be \( h = h' \in \Pi_\Lambda \) since \( h'_o \) is not in \( H' \). And if \( h = h = h_r \in \Pi_\Lambda \) then, it was shown above, there is an \( h' \) in \( \pi \) with \( h' = h_r \) or \( \text{obsbd}(h', h_r) \).

(iii) If, in the \( \text{obsbd}(h, h) \) case, \( h = h'_o \) or \( h_r \in \Pi_\Lambda \) and the invoked method is an observer, we switch from \( h \) to \( h_o \), which is always in \( \pi = \pi_1 \cdot \pi_2 \) as first edge of \( \pi_2 \). Notice that \( h_o \) was assumed to be new at the beginning of the \{call\}-case. Hence in \( g''^\Theta \), \( h_o \) is local to the called method and neither local at other call-levels nor captured in a field. That is, \( h_o \) is just one observer higher than \( h \). Hence \( \text{obsbd}(h, h) \) implies the necessary \( \text{obsbd}(h_o, h) \) for \( h \in H' \).

(iv) If, in the \( \text{obsbd}(h, h) \) case, \( h = h'_o \) or \( h_r \in \Pi_\Lambda \) and the invoked method is a mutator, then the typeability of redex \( \check{e} \) requires that call-link \( h_r \) is freet since the calling invocation containing \( h \) is an observer. It was shown above that then there is an \( h' \) in \( \pi \) with \( h' = h_r \) or \( \text{obsbd}(h', h_r) \). In the former case, \( h' = h_r \) and \( h \in \Pi_\Lambda \) are identical or at least at the same call-level. Hence \( \text{obsbd}(h, h) \) implies the \( \text{obsbd}(h', h) \) necessary for \( h \in H' \) with \( h' \) in \( \pi \). In the latter case, \( \text{obsbd}(h', h_r) \) means that \( h_r \) was above \( h' \) or only some observer calls below it. On the other hand, we had \( \text{obsbd}(h, h) \) for \( h \in \Pi_\Lambda \), i.e., \( \text{obsbd}(h_r, h) \) or \( \text{obsbd}(h'_o, h) \). That is, \( h \) was above \( h_r \) or \( h'_o \) (which is the same) or only some observer calls below it. In combination, \( h \) was above \( h_r \) or only some observer calls below it. Hence we have \( \text{obsbd}(h', h) \) necessary for \( h \in H' \) with \( h' \) in \( \pi \).

**Fifth.** Consider reservations \( \langle \pi, \varphi \rangle \) with a new \( \leftarrow \)-path \( \varphi = q \text{--}_\varphi \ominus \omega \text{ via } \Pi \). The existence of reservation \( \langle \pi, \varphi' \rangle \) with the unchanged trivial \( \leftarrow \)-path \( \varphi' = (q, \tilde{\gamma}) \cdot e \text{--}_\varphi \ominus q, \tilde{\gamma} \text{ via } \Theta \) ensures, as shown above, that all objects in \( \pi \) satisfy the invariants for ownership reservations. We still need to check the objects \( u \) in the path-base \( \Pi \). For new \( \varphi \), there was in \( g^\Theta \) a \( \tilde{\gamma} \)-qbridge from \( q \) to \( \omega \text{ via } \Pi' \) such that \( \Pi' \cup \Pi_o = \Pi \cup \Pi_\Lambda \cup \Pi_\Theta \) (Lemma 18). The objects in \( \Pi \subseteq \Pi' \cup \Pi_o \) are the source or target of edges occurring also in \( \Pi' \), or of \( h_o \) and \( h^{-1}_o \in \Pi_o \). Source and target of \( h_o \) and \( h^{-1}_o \) are covered as the targets of \( h_r \) and of \( h'_o \) or sources of \( h^{-1}_r \) and of \( h'^{-1}_o \), which are guaranteed to be
contained in $\Pi'$. All handles in $\Pi'$, and thus all objects in $\Pi$ are contained in one of the following reservations:

Each quadruple in the qbridge means two reservations $\langle \pi'_1, \varphi'_1 \rangle$ and $\langle \pi'_1, \varphi_1 \rangle$. The qbridge’s initial $\Rightarrow$-path $\varphi_0 = q \xrightarrow{\delta} \pi_1 \omega_1$ via $\Pi_0$ is covered by the reservation $\langle \pi'_0, \varphi_0 \rangle$ if $\pi$ is unchanged or internally-only new and $\pi'_0$ is its precursor $\pi'$. And if $\pi$ is internally really new or initially new, then $\varphi_0$ combines with the last $\Rightarrow$-path $\varphi_n$ via $\Pi'_n$ of $\pi$’s bridge to a $\Rightarrow$-path $\varphi'_0 = q \xrightarrow{\delta \varphi - \pi_1} \omega$ via $\Pi'_n \cup \Pi_0$. It combined to the reservation $\langle \pi'_0, \varphi'_0 \rangle$ with witness $\pi'_0 = \text{wit}(\pi)$ or with the last ownership path $\pi'_0 = s \xrightarrow{\delta \varphi} q'_n \alpha'_n$ in $\pi$’s bridge. Either directly or through $\pi$’s bridge, the reserved ownership assumption guarantees that $\pi'_0$ has the same mode $\mu'$ and source $\sigma'$ as, respectively, $\pi'$ or $\text{wit}(\pi)$ (cf. the internally really new case above). Observe that the initial edges of each ownership path in the qbridge and in the bridge is $h_1$ or $h'_0$.

If $\mu'$ 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 $\pi$’s by the reserved ownership assumption that it is $h_r$ or $h'_0$. Actually, it must be $h_1 = h_r$ since unchanged $h_1$ cannot be the free $h'_0$ that is decreased to zero multiplicity (cf. the internally really new $\pi$ case further above). If $\mu'$ is free and $\mu_o \neq \text{co}<>$ then there are no triples and no quadruples: In the non-co case, one ownership path $\pi$ of the ownership path-pair in each triple or quadruple has shape $s \xrightarrow{\delta \varphi} q'_n \alpha'_n$ in $\pi$’s bridge. The nesting constraint on modes, on one hand, allows a free $\pi$ only if it has shape $s \xrightarrow{\delta \varphi} q'_n \alpha'_n$ with $\mu_o(\alpha) = \beta$. The nesting constraint on modes, on one hand, allows a free $\pi$ only if it has shape $s \xrightarrow{\delta \varphi} q'_n \alpha'_n$ with $\mu_o(\alpha) = \beta$. On the other hand, it disallows correlations $\beta = \text{Free}<>$. If $\mu_o(\alpha) = \beta =$ Free<> cannot be free.

In each reservation $\langle \tilde{\pi}, \tilde{\varphi} \rangle$, the sources and targets of all edges except for $\tilde{\pi}$’s initial $h'_1$ are covered by the induction hypothesis. $h'_1$ is either the initial edge of $\pi'_0$, which has nothing to do with $\varphi$ and was already covered above. Or it is the initial edge $h_r$ or $h'_0$ of an ownership path in the qbridge, which is skipped in the construction of the new $\Rightarrow$-path $\varphi$ 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 $\langle \tilde{\pi}, \tilde{\varphi} \rangle$, i.e., for all objects in $\varphi$ we need to cover, the induction hypothesis allows the following possibilities:

- $u$ was immutable in $g^{\oplus}$. Hence it still is so in $g^{\otimes}$.
- $u \in RFree(h'_1)$ in case of a free reservation. If there are no quadruples in $\varphi$’s qbridge and no triples in $\pi$’s bridge then $\tilde{\pi} = \pi'_0$ is $\pi'$ or $\text{wit}(\pi)$. In the initially new $\pi$ case then $\mu_o$ is free and $\text{wit}(\pi)$’s initial edge $h'_1$ is $h'_0$. Hence $u \in RFree(h'_1)$ is succeeded by $u \in RFree(h_o)$, which is just right since $h_o$ is $\pi$’s initial edge $h_1$ (cf. the initially new $\pi$ case further above). If $\pi$ is not initially new then $\pi''$’s initial edge $h'_1$ coincides with $\pi'$’s initial edge $h_1$. Since $h'_1 = h_1$ is unchanged, it cannot be free $h'_0$ (cf. the internally really new $\pi$ case further above). Hence $u \in RFree(h'_1) = RFree(h_1)$ is preserved. If there are quadruples or triples then, as shown above, $\mu_o = \text{co}<>$, so that there are no initially new paths, and $\tilde{\pi}$’s initial edge is $h'_1 \in \{h_r, h'_0\}$, and $h_1 = h_r$ for non-initially new $\pi$. If $h'_1 = h_r$, $u \in RFree(h'_1) = RFree(h_r) = RFree(h_1)$ is preserved, and if $h'_1 = h'_0$, we have new $u \in RFree(h_r) = RFree(h_1)$ in $g^{\otimes}$.

193
• $u \in RSub_H(\hat{q})$ in case of a rep reservation for some $\hat{q}$ that was $\alpha'$ or immutable and for every $h \in H$ there was a $h$ in $\hat{\pi}$ or $\hat{\Pi}$ such that $h = h \lor obsbd(h, h)$. In the rep case, $\pi$ cannot be initially new since there are only free initially new ownership paths, and these have a free precursor $sent(\pi)$, so that also the mode of $\hat{\pi}$ is free. For the other $\pi$, $\alpha' = \alpha$. As shown above, we have $u \in RSub_H(\hat{q})$ in $g^\ast$ for some $\hat{q}$ that is still $\alpha' = \alpha$ or still immutable. $H'$ is $H$ with any $h'_o$ in it expanded to $h_r$ and/or $h_o$ (depending on which of them is free). It remains to check all the $h \in H'$: If free $h$ is $h_r$ or $h_o$, then $\pi$ cannot be internally-only new since this would presuppose $\mu_r = \mu_o = \text{co}<>$. And a internally really new $\pi = \pi_1 \cdot \pi_2$ always contains $h_o$ as the first element of its $\pi_2$-segment for $h = h_o$, and always contains an $h'$ with $h' = h_r$ or $obsbd(h', h_r)$ for free $h = h_r$. For details see the internally really new reservation case further above. The remaining $h \in H'$ were also in $H$: For every $h \in H$, there was the necessary $h$ in $\hat{\pi}$ or $\hat{\Pi}$. This means, $h$ is in some path in $\Pi'$. Since $\Pi' \subseteq \Pi \cup \Pi_\Lambda \cup \Pi_\varphi$, this means that $h$ must be in $\Pi$ or must be $h_r$ or $h'_o \in \Pi_\Lambda$. Whether $h$ is free $h$ or non-co and uncaptured ($obsbd(h, h)$), it cannot be in the dummy edge-path $\pi'_2$ nor in dummy edge set $\Pi_\varphi$.

If $h$ was in $\Pi$, then it is in $\langle \pi, \varphi \rangle$. Hence we have for $h \in H'$ the desired handle $h$ with still $h = h$ or $obsbd(h, h)$, respectively. If $h = h_r$ or $h'_o \in \Pi_\Lambda$, then $\pi$ cannot be internally-only new since this would presuppose $\mu_r = \mu_o = \mu_r \circ \mu_o = \text{co}<>$. Otherwise the reasoning is like in the internally really new reservation case: In case of $h = h$, $h$ cannot be $h'_o$, and if it is free $h_r$ then there is an $h'$ in $\pi$ with $h' = h_r$ or $obsbd(h', h_r)$. In case of $obsbd(h, h)$, if a mutator is called, $h_r$ must be free, which ensures an $h'$ in $\pi$ with $obsbd(h', h)$. And if an observer is called is then $h_o$ is just one observer higher than $h$, so that we have $obsbd(h_o, h)$ for $h \in H'$. ■
Chapter 7

Discussion

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, integrated deadness checking would be preferable 
since it avoids null pointer errors by calls through an object reference variable $v$ after 
it was read destructively (to transfer the target object as a parameter or to call a 
mutator).

2. Interpretation ≠ 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 $O$ of elementary objects as one composite object $O$ 
with representative $o \in O$ 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 compu-
tations 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, OheOl]. 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 ⊥ 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 \textit{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: \texttt{pointer of} \( c \) in Pascal, \texttt{access} \( c \) in Ada, \texttt{ref} \( c \) in ML, and \texttt{*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 \texttt{implements} and \texttt{extends} clauses, nor in qualified names. Modes \( \mu \in \mathcal{M} \) may appear like (a family of) \textit{type constructors} \( \mu : \mathcal{C} \rightarrow \mathcal{T} \) 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 \textit{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. \textbf{COMPARISON: DIMENSION QUALIFICATION.} Modes are similar to the qualification of numeric types and numerals with \textit{physical dimensions} (length, time, mass, voltage, …) in physical formulæ and in some proposed extensions to programming and specification languages [Ken94, HM95, Ken97]. Through qualifications \texttt{4m}, \texttt{4s}, \texttt{4g}, \texttt{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 \textit{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., “\texttt{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 $12Ω \cdot 4ΩA$
of two dimensioned values by an operation called multiplication produces a value $48ΩA$
$= 48V$ with multiplied dimension. Similarly, the combination $s \cdot 4A \cdot r \circ r \cdot 4A \cdot o$
of two references to one by an operation called return (of result $r \cdot o$, $o$ through call-link $s \cdot r \circ r \cdot r$) yields a value $s \rightarrow o$ whose mode is the “multiplication” $\mu_r \circ \mu_o = \mu'$ 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 un­
declared 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 $\mu \circ \mu \leq_m$
$\mu' \circ \mu$ does not mean that a $\mu \circ \mu$ value is also a $\mu' \circ \mu$ value. It merely says that it is safe
to change a value of mode $\mu$ into another value of mode $\mu'$. The mode compatibility
relation $\leq_m$ is no is-a or generalization relation like the subclass relation $\leq_c$ in Java
(cf. §7.2.2).

For instance, a reference value $h = \langle s, \text{free} \rangle, \omega \rangle$ 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 $\omega$ to the source’s
sanctuary Sanc($s$). In type inference, it would be unsafe to “approximate” an object
reference’s type $\mu \circ \mu$ by a type $\mu' \circ \mu \geq_m \mu \circ \mu$ 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 $s \cdot$ free$<>$. $r$ was typed with mode rep$<>$ instead of free$<>$,
then the typing rule for operation call expressions with $e$ as receiver expression would
allow $s$ to supply a reference $s \cdot$ rep$<>$. o to $r$ as a co parameter value $r \cdot$ co$<>$. o. But
then an old alias of the rep reference and the new free path $s \cdot$ free$<>$. r \cdot$ co$<>$. o would
violate the Unique Head property.

8. **Covariant Parameter Modes.** If an object reference is mode-converted
from compatible $\mu \leq_m \mu'$ to $\mu'$ then the import of the the target’s co results and
parameters in the handle's signature changes covariantly from $\mu$ to $\mu'$. 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 $\mu$ to $\mu'$. For example, the method `SetNext` of `Node` objects has the type $(\text{co}<> \text{Node})$ $\text{mut}$, $\text{co}<> \text{Node}$ in signature $\Sigma(\text{Node})$. Its type in the signature $\Sigma(\mu \text{Node})$ of $\mu$-references to `Node` objects is $(\mu \text{Node})$ $\text{mut}$, $\mu$ $\text{Node}$. Hence if $\mu$ changes to $\text{rep}$-$\text{elem}$-$\text{rep}$, then result and parameter mode both change covariantly:

<table>
<thead>
<tr>
<th>mode $\mu$</th>
<th>type of <code>SetNext</code> in $\Sigma(\mu \text{Node})$</th>
</tr>
</thead>
<tbody>
<tr>
<td>free$\langle$elem$=$rep$\rangle$</td>
<td>$(\text{free}$-$\text{elem}$-$\text{rep}$-$\text{Node}$) $\text{mut}$, free$\langle$elem$=$rep$\rangle$-$\text{Node}$</td>
</tr>
<tr>
<td>$\leq_m$ rep$\langle$elem$=$rep$\rangle$</td>
<td>(rep$\langle$elem$=$rep$\rangle$-$\text{Node}$) $\text{mut}$, rep$\langle$elem$=$rep$\rangle$-$\text{Node}$</td>
</tr>
</tbody>
</table>

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 modespecified higher-level interpretation’s integrity (§6.2.3). The latter restrictions are encoded in the rule for the signature $\Sigma(\mu \ c)$ of $\mu$-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 $\mu$ contain only the base-modes free and read ($\forall \vec{a} \in \Lambda^*. \mu(\vec{a})$ is defined $\Rightarrow \mu(\vec{a}) \in \{\text{free, read}\}$). Such modes never change by mode import ‘$\cdot$’ ($\mu_\tau \cdot \mu = \mu$), irrespective of the call-link’s mode $\mu_\tau$. 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
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 ----> w is passed to the free object w itself or a sub-object of it, then this creates a cycle of ownership paths through w. 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 w possesses its own free reference, it cannot be on the stack, so that w cannot be executing a mutator. And sub-objects of w could obtain it as parameters of a mutator only from w or other sub-objects of w.

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 freeα→ oo ω and o repα→ oo ω 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 ω α.q can arrive in ω by parameter only if representative o supplies a reference o αμ ω. And only the representative o can give ω a reference ω readβα→ oo ω′ or ω freeα→ oo ω′ through which ω can obtain association references ω α→ q as results (and through which further references ω μβα→ oo ω′ 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=\text{read}(\ldots), the representative was not allowed to pass a \text{read} reference as an association moded parameter of its component. (While this prevents to store \text{read} references in generic containers, \text{read} references can of course always be stored in other objects as \text{read} references.) Extending the support for container objects to containers of \text{read}, \text{co} and \text{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 \( \omega \) from its owner’s mutators only by flowing downward along a rep reference either directly to \( \omega \) or first to a co-object of \( \omega \) 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 \text{main}, the path of calls from program start to the current method always has a form captured in the following regular expression:

\[
\left( \left( \mathcal{L}_{\text{obs}} \right)^* \text{ free, mut( co, mut)* rep, mut( co, mut)*} \right)^* \]

where \( \mathcal{L} \), \( \kappa \) stands for the call of a \( \kappa \)-method through a \( \mu \)-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 \( \mu_{i+1} = \mu_r \circ \mu \), 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 \( h = s \triangleleft o \) to an object \( r \) through a reference \( h_r = s \triangleleft r \) as a parameter reference \( h' = r \triangleleft o \) if \( \mu_r = \mu_r \circ \mu \). \( s \) could convert \( h_r \) to \( h'_r \) of mode \( \mu'_r \geq_m \mu_r \), and \( r \) could convert \( h'_r \) to \( h''_r \) of mode \( \mu' \geq_m \mu_r \). If then \( h''_r \) is returned back to \( s \), we get a reference \( h'' = s \triangleleft o \) of mode \( \mu'' \). Consistency in this case demands that the original \( h \) was compatible to \( h'' \). In other words, we should have the algebraic property of monotonicity: \( \mu_r \circ \mu_\leq_m \mu'_r \circ \mu' = \mu'' \). (The monotonicity is a partial one since \( o \) is a partial mapping.)

To see that mode import \( o \) is monotone in the right argument, consider elementary conversions \( \mu \leq_m \mu' \): If \( \mu \) was \( \text{co}<> \) or \( \alpha<> \) then \( \mu' = \text{read}<> \), so that \( \mu_r \circ \mu = m<\delta> \leq_m \text{read}<\delta> \leq_m \text{read}<> = \mu_r \circ \mu' \). If \( \mu = m<\ldots, \alpha_i=\mu_i, \ldots> \) with \( m = \text{free} \) or \( \text{rep} \) then \( \mu_r \circ \mu_m = m<\ldots, \alpha_i=\mu_o, \ldots> \) with \( m = \text{free} \) or \( \text{read} \), and \( \mu' \) is \( m'<\ldots, \alpha_i=\mu_r, \ldots> \) with \( m = \text{rep} \) or \( \text{read} \), so that we have \( \mu_r \circ \mu' = \text{read}<\ldots, \alpha_i=\mu_r, \ldots> \). Hence \( \mu_r \circ \mu \leq_m \mu_r \circ \mu' \). If \( \mu = \text{read}<> \), \( \alpha_i=\mu_i, \ldots \) and \( \mu' \) has less or weaker correlations, then \( \mu_r \circ \mu = \text{read}<\ldots, \alpha_i=\mu_r, \ldots> \), and \( \mu_r \circ \mu' \) has fewer correlations or a correlation \( \alpha_i=\mu_o, \mu'_i \) with \( \mu_i \leq_1 \mu'_i \). In the first case, \( \mu_r \circ \mu \leq_m \mu_r \circ \mu' \) by width compatibility. In the second case, \( \mu_r \circ \mu = m<\ldots, \alpha_o=\mu_o, \ldots> \) and \( \mu' \) has fewer correlations or a correlation \( \alpha_i=\mu_o, \mu'_i \) with \( \mu_i \leq_1 \mu'_i \). The same obviously holds for if we add correlations \( \alpha_i=\mu_i \) with \( \mu_r \circ \mu_i = \mu'_r \circ \mu_i \). If \( \mu \) contains a correlation \( \alpha_i=\mu_i \) with \( \mu_r \circ \mu \neq \mu'_r \circ \mu_o \), this presupposes that \( \mu_i \) is an association mode, or contains one. The latter case can be followed by structural induction. If \( \mu_i \) is \( \beta<> \) with \( \mu_r \circ \beta<> \neq \mu'_r \circ \beta<> \), \( \mu_r \) must have been depth-converted (see above), so that \( \mu_r \circ \beta<> \leq_m \mu'_r \circ \beta<> \). If \( \mu = m<\ldots, \alpha_i=\mu_i, \ldots> \) is read or
then
rep then
modes are depth-compatible in
however, is irrelevant for the consistency of reference value flow: To convert the call-
link hr = s
field requires destructive read and thus a mutator call, while depth-conversion of hr
and a second call
implied that
there are two ways of passing a reference value
That is,
first be shortened
f.L-reference
hq
13.
passes its reference value
associativity:
(AoB)of.L
formulated in ownership-based systems that give moded type terms
Types
interpretation
t A-
alence
relation sets, width-compatibility (more or fewer correlations) and depth-compatibility
compatibility relation
In §6.2.3, the compatibility between modes with changed corre­
striction, it would be possible for an object to set up object references through which
'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 ψ(t_A)(t_B) or “σ(t_B) with σ =
ψ(t_A)” and [t_B]_p would in [MP01, MP99a] be written τ(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>.setInput();
    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* ≤ const T*. But pointers to variables of these pointers are not compatible, T** ≤ 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** ≤ const T*const*. Converting a T** pointer to const T*const* is safe since the new pointer 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<> ≤_m rep<data=read<> and rep<data=rep<> ≤_m rep<>, but read<data=rep<> ≤_m read<data=read<> and read<data=rep<> ≤_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].

206
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 \(\alpha \in A\), 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 \(\text{arg } \alpha\) and \(\text{var } \alpha\) (with/without limitation to the target’s “clean” interface). Ownership Types [CPN98] had only plain identifiers \(\alpha\) with the same function as \(\text{var } \alpha\), 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 \(\nu\) at an r-value position can be desugared by the following rules:

   - \(\nu\) stands for \(\text{val}(\nu)\) if \(\nu\) is a non-free variable or if \(\nu\) is a free variable and a typing with a \text{read} mode suffices to type the statement in which \(\nu\) occurs.
   - \(\nu\) stands for \(\text{destval}(\nu)\) if \(\nu\) is a free variable otherwise.

2. Empty correlations \(<>\) on base-modes \(m\) (in reference types and in correlations), and on \text{new} and \text{null} can be omitted. This reduces the bulkiness of all modes, making declarations and \text{new} expressions much easier to read. In particular, association modes \(\alpha<>, \text{mode } \text{co}<>, \text{and } \text{null}<\) can always be abbreviated to just \(\alpha\), \text{co}, and \text{null}.

3. Also Java’s void methods can be supported as syntactic sugar: The result type \text{void} in the header of a method in a class \(c\) stands for the type \text{co}< >\(c\). The body of void methods is extended by the return statement \text{return } \text{val}(\text{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 \( \text{dummy}_e = e.f(...) \), where \( \text{dummy}_e \) is an implicit local variable of type \( \text{read<>} \ c \).

• The case of a destructive read receiver expression \( e \equiv \text{destval}(\nu) \) 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 \( \nu = e.f(...) \) (equivalent to desugared \( \nu = \nu.f(...) \)). That is, variable \( \nu \) is reset for use as a call-link in a mutator call, but when the call is finished, \( \nu \) gets back its old value back since the mutator returns this. This temporary reset can only be observed during the made mutator call if \( \nu \) is a field. Hence if \( \nu \) 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 startup expression \( \text{new<>} \ c_n().\text{main()} \) is evaluated in an environment \( \varnothing^{\text{obs}}_{(\text{nil,read<>},\text{nil})} \) 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 \( D_{n+1} = \text{class Main2}\{\text{obs void main()} \{\text{new } c_n().\text{main();}\}\} \). 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"):

$$\models s \iff \forall \tau \in \mathcal{M} \times \mathbb{C}, \ell \in \text{dom}(s) \models \exists \tau' \leq_c \tau. \quad \models s(\ell) \in \llbracket \tau' \rrbracket$$

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):

$$(e, \eta, s, om, g \Rightarrow e', \eta', s', om', g') \land \Gamma, \kappa \vdash e : \tau \quad \land \quad \eta \models \mu, \Gamma, \kappa, X \land \models s, om \quad \Rightarrow \quad \exists X', \tau' \leq_c \tau. \quad \Gamma, \kappa \vdash e' : \tau' \land \eta' \models \mu, \Gamma, \kappa, X' \land \models s', om'$$

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 $\tau'$ to a supertype $\tau$. That is, we can add a so-called subsumption rule to the typing rules:

**[sub]**

$$\frac{\Gamma, \kappa \vdash e : \tau' \land \tau' \leq_c \tau}{\Gamma, \kappa \vdash e : \tau}$$

(Through subsumption for runtime terms, the type preservation property can be recovered from the subject reduction property since then $\exists X', \tau' \leq_c \tau. \Gamma, \kappa \vdash e' : \tau'$ entails $\exists X'. \Gamma, \kappa \vdash e' : \tau'$.)

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 `ref $\tau$ of $\tau$-variables, on the other hand, have no subtypes, even if $\tau$ is a reference type.) The subtypes $\tau' \leq_c \tau$ of a type $\tau$ 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:

$$p \equiv D_1 \ldots \text{class } c_i \text{ extends } d \{ \ldots \} \ldots D_n$$

$$\vdash c \text{ ok} \quad \vdash c \leq_c c$$

$$\vdash c \leq_c c' \quad \vdash c' \leq_c c''$$

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 $\mu$-references to $c'$-instances is a subtype of the type $\tau = \mu c$ of $\mu$-references to $c$-instances:

$$\vdash c \leq_c c' \quad \vdash \mu c \leq_c \mu c'$$

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 $\text{FldsMths}(c_i)$ of subclasses $c_i$:

$$p \equiv D_1 \ldots \text{class } c_i \text{ extends } d \{ \ldots \} \ldots D_n$$

$$\vdash \text{FldsMths}(d) = (\Gamma, F)$$

$$\vdash \text{FldsMths}(c_i) = \{ \{x_i : \text{ref } t_i\} \cup \Gamma, \{\bar{f}_i : \kappa_i t_i \bar{f}_i(\pi_i)\{b_i}\} \cup F |_{\bar{f}_i(\pi_i)}\}$$

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:

$$\vdash M_1 \text{ defs } x_1 \ldots \vdash M_n \text{ defs } x_n$$

$$\forall i, j = 1, \ldots, n. x_i = x_j \Rightarrow i = j$$

$$\text{FldsMths}(d) = (\Gamma, F) \quad \text{dom}(\Gamma) \cap \{x_1, \ldots, x_n\} = \emptyset$$

$$x_i \in \text{dom}(F) \Rightarrow M_i = \kappa t x_i(b_i)\{\ldots\}$$

$$\wedge F(x_i) = \kappa t x_i(b_i)\{\ldots\}$$

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 subtype-consistency of stores weakens
old implications of the form \( \ell \in \text{Loc}_r \vdash^s s(\ell) \in \llbracket \tau \rrbracket \) to implications \( \ell \in \text{Loc}_r \vdash^g \exists \tau' \subseteq_c \tau \ast s(\ell) \in \llbracket \tau' \rrbracket \). For read access reduces \( \hat{e} \) reduced to \( s(\ell) \), this entails \( \Gamma, \kappa \vdash_X s(\ell) : \tau' \), which by subsumption can be widened to the original \( \Gamma, \kappa \vdash_X s(\ell) : \tau \). For field access this \( .x \), it means that the instance record \( g' \) of the this object is consistent with the instance record type \( \Gamma_c \) of a subclass \( c' \subseteq_c c \) of the class \( c \) in this's static type: \( g' \models \Gamma_c \). 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 \( \Gamma, \kappa \vdash_X \langle a, \mu, \omega \rangle : \mu \ c \Rightarrow \langle a, \mu, \omega \rangle \in \llbracket \mu \ c' \rrbracket \) to implications \( \Gamma, \kappa \vdash_X \langle a, \mu, \omega \rangle : \mu \ c \Rightarrow \exists c' \subseteq_c c, \langle a, \mu, \omega \rangle \in \llbracket \mu \ c' \rrbracket \). For return steps reduced to \( h \), the correspondingly adapted conclusion is \( \Gamma, \kappa \vdash_{X'} h : \mu^* \circ \mu'' \ c' \), which by subsumption can be widened to the original \( \Gamma, \kappa \vdash_{X'} h : \mu^* \circ \mu'' \ c \). For operation call expressions \( \langle s, \mu, r \rangle . f(\ldots) \), it means a weakening of Lemma 2 so that the target's method suite \( F_r \) and the method suite \( F_c \) of the receiver expression's target class \( c \) do not coincide any more. But since receiver \( r \) must be instance of a subclass of \( c \), \( F_r \) and \( F_c \) can differ at \( f \) only in the irrelevant declarations \( \pi \) and \( \lambda \) of parameters and local variables: \( F_r(f) = \kappa^* \mu \ d f(\pi) \{ \lambda \ s \} \) while \( F_c(f) = \kappa^* \mu \ d f(\pi') \{ \lambda' \ s' \} \). 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 Owner and 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 \( \mu_{o_i} \) is compatible to the import \( \mu_c \circ \mu_{o_i} \) of the modes \( \mu_{o_i} \) of the parameters in the receiver's method relative to the call-link's mode \( \mu_r \). This still holds even if the receiver expression \( \langle s, \mu_r, r \rangle \) was typed via the subsumption rule with a supertype \( \mu_r c \), so that receiver object \( r \) is an instance of a subclass \( c' \subseteq_c c \). The receiver's method \( F_r(f) \) has the same parameter modes \( \mu_{o_i} \) as the superclass's method \( F_c(f) \) used to calculate the parameter modes \( \mu_r \circ \mu_{o_i} \) in the receiver expression's signature \( \Sigma(\mu_r c) \) 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 straightforward 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\( \delta \rceil \)’.

- 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 enabling 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 \( \mu_r \)-reference, \( \mu \)-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 \( \alpha\rangle \) result not returnable through a reference without \( \alpha \)-correlation could be converted to a read\( \langle \rangle \) reference and then returned to the sender as a read\( \langle \rangle \) reference. And a result of mode free\( \langle \beta=m\rangle \rangle \) not returnable through a read reference could be converted to read\( \langle \beta=m\rangle \rangle \) to arrive at the sender as read\( \langle \beta=\mu_r\otimes m\rangle \rangle \).

b) If through a \( \mu_r \)-reference, \( \mu' \)-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
The adapted rules for receiver-side conversion are as follows:

- For a method with an inaccessible parameter of mode rep<> in the caller, 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 μ ≥_m μ' for which import μ' of μ 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 μ = rescnv(μ', μ_r) to which μ' is compatible (μ' ≥_m μ) and that is always importable as the result of operations imported into the signatures of μ_r-references.

Figure 7.3 shows the adapted rules for handle signatures and for return steps.

rescnv(μ', μ_r) should adapt μ' no more than necessary. In the examples above, this was the adaption α<> ≤_m read<> and free<β=μ> ≤_m rep<β=μ> ≤_m read<β=μ>. A look at the handle signature rule and at the definition of ≤_m and proper modes 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 rescnv(μ', μ_r) and that rescnv(μ', μ_r) passes the conditions on the result of operations imported into the signatures of μ_r-references. Limiting receiver-side conversion of results to conversion to rescnv(μ', μ_r)
and no further is no limitation for the sender: It is possible to show that the imported \( \text{rescnv}(\mu', \mu_r) \) 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 \( \mu'' \geq_m \text{rescnv}(\mu', \mu_r): \mu_r \circ \mu'' \geq_m \mu \circ \text{rescnv}(\mu', \mu_r) \). This is consequence of the algebraic property of \textit{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 \( \text{JaM} \) by implicit mode receiver-side conversion preserves \( \text{JaM} \)'s ownership and state encapsulation properties.

3. \textbf{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” \texttt{rep} references and association references with itself (since the \texttt{this} reference, like all \texttt{co} references, has no correlations). These restrictions are superfluous for self-calls, where sender and receiver coincide, since here the \textit{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 \texttt{this} without restrictions and without adaption is crucial for factoring out operations (even on references with \texttt{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 \( \text{JaM} \) by self-calls with implicit receiver \texttt{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 \texttt{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. \textbf{REFACTORYING EXAMPLE.} For example, in the \texttt{MapImp} class in appendix B, the iteration over the \texttt{entryset} component in search for a given (potential) key object \( k \) is defined three times (in methods \texttt{Add}, \texttt{Remove}, and \texttt{lookup}). This search can be factored out into one separate (private) method \texttt{find_entry} that returns the reference to the desired entry pair with the mode \texttt{rep<fst=key, snd=value>} (or, in desugared \( \text{JaM} \), \texttt{rep<fst=key>, snd=value>})). This leads to a considerable simplification of the class. Special support for self-calls permits method \texttt{Add} to import the result as a \texttt{rep} reference, and not just a \texttt{read} reference, thus enabling it to update old entries with the given key to the new value. The restructured \texttt{MapImp} class is shown below.
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 $w$ is executing a mutator, any object $o$ to whose state representation it belongs is guaranteed to be executing a mutator (mutator control). Hence while $w$ 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 $\alpha$ and arg $\alpha$ in [NVP98], and by context parameters $\alpha$ 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 captured. 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 [BoyOl], 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 associ-
ation 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 disserta-
tion 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 sub-
typing”) [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 implementa-
tions) 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 ex-
ternal 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 compon-
ent. 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 implementa-
tion 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 means that the implementations 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 Poetsch-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.

```java
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.”

```java
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).

```java
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 \texttt{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 encapsu-
lation 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'

§ 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 trans­
ferred 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 to­
gether 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 arbi­
trary 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
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<List<String>> titles);
    mut void AddTools(int bgcol, 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<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 transferred into an X3mForm widget that combines them and fixes their relative geometrical placement (horizontally left/right, or vertically one above the other). This X3mForm 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.

228
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 transferred 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)

\[ p \in P := D^* \]
\[ D \in D := \text{class } C \{ (T \text{Id};)^* \text{Mth}^* \} \]
\[ \text{Mth} := \sum T \text{Id}((T \text{Id})^*) \{(T \text{Id};)^* S\} \]
\[ k \in K := \text{mut} \mid \text{obs} \]
\[ t \in T := M \subset \sum \]
\[ \mu \in M := B < \Delta > \]
\[ m \in B := \text{free} \mid \text{rep} \mid \text{co} \mid A \mid \text{read} \]
\[ \delta \in \Delta := (A=M)^* \]
\[ s \in S := SS \mid N=E; \mid \text{return } E; \mid \text{if}(E \forall E)\{S\} \mid \text{while}(E \forall E)\{S\} \]
\[ \psi \in \Psi := \text{==} \mid \text{!=} \]
\[ e \in E := \text{val}(N) \mid \text{destval}(N) \mid \text{null} < \Delta > \mid \text{new} < \Delta > C() \mid E \leftarrow Id(E^*) \]
\[ v \in N := Id \mid \text{this}.Id \]

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 \( x, y, z, f \in \text{Id} \) (includes this, excludes null)

2. RUNTIME TERMS — extension of program’s statements and expressions

\[ R := R \mid S \mid R = R; \mid \text{return } R; \]
\[ \mid \text{if}(R \forall R)\{S\} \mid \text{while}(E \forall E)\{S\} \]
\[ \mid N \mid \text{val}(R) \mid \text{destval}(R) \mid \text{null} < \Delta > \mid \text{new} < \Delta > C() \mid R \leftarrow Id(R^*) \]
\[ \mid \text{Loc} \] location of a variable (l-value)
\[ \mid V \] expression value (r-value)
\[ \mid \llangle R \rrangle \] inlined executing method
3. **Formal Type Terms** — extension of program’s type terms for type checking and semantic consistency

\[ \tau \in T ::= \text{ref } M \ C \quad \text{l-values (locations)} \]

\[ M \ C \quad \text{values (handles)} \]

\[ \text{obj } C \quad \text{object values} \]

\[ \text{Cmd} \quad \text{continuing statements} \]

A.2 Type System

All definitions are relative to a given program \( p \in P \).

4. **Valid Mode, Range Type, Class Name**

\[ \forall i, j \in \{1, \ldots, n\}, \alpha_i = \alpha_j \Rightarrow \mu_i \equiv \mu_j \]

\[ m \in \{ \text{co} \} \cup \mathbb{A} \Rightarrow n = 0 \]

\[ \forall i \in \{1, \ldots, n\}, \mu_i \neq \text{free} \land \mu_i \neq \text{co} \land \mu_i \quad \text{ok} \]

\[ m < \alpha_1 = \alpha_i, \ldots, \alpha_n = \alpha_j \quad \text{ok} \]

\[ [\text{mode}] \quad \vdash \mu \quad \text{ok} \quad \vdash \mu \quad \text{ok} \]

\[ \vdash \mu \quad \text{ok} \quad \vdash \mu \quad \text{ok} \]

\[ \vdash \mu \quad \text{ok} \quad \vdash \mu \quad \text{ok} \]

where \( \mu \equiv \mu' \iff \forall \alpha \in \mathbb{A}^* \cdot \mu(\alpha) = \mu'(\alpha) \)

where \( \mu(\epsilon) = m \) if \( m = \diamond \)

\[ \mu(\alpha, \overline{\alpha}) = \begin{cases} \mu'(\overline{\alpha}) & \text{if } m = \diamond, \alpha = \mu', \ldots \\ \bot & \text{otherwise} \end{cases} \]

5. **Mode-Compatibility Judgement**

\[ \vdash \tau \leq_\mu \tau' \]

\[ m \diamond \leq_\mu \text{read} \diamond \]

\[ \text{free} \diamond \leq_\mu \text{rep} \diamond \]

\[ \text{read} \diamond, \alpha = \mu, \delta' \leq_\mu \text{read} \diamond, \delta' \]

\[ \text{read} \diamond, \alpha = \mu, \delta' \leq_\mu \text{read} \diamond, \alpha = \mu', \delta' \quad \text{if } \mu \leq_\mu \mu' \]

6. **What Class Modules Define**

\[ \vdash \text{FoldsMths}(c) = (T, F) \]

\[ p = D_1 \ldots \text{class } c_i \{ t_i, x_i; \kappa_i t_i f_i(\pi_i) \{ b_i \} \ldots D_n \]

\[ \vdash \text{FoldsMths}(c_i) = \{ \{ x_i : \text{ref } t_i \}, \{ f_i \mapsto \kappa_i t_i f_i(\pi_i) \{ b_i \} \} \}

7. **Handle Signature Judgements**

\[ \vdash (f : \text{ref } x, \tau) \in \Sigma(\mu \ c) \]

232
\( \vdash FldsMths(c) = (\Gamma, F) \quad F(f) = \kappa \mu d f (\mu_i d_i y_i) \{ \ldots \} \)

\( \forall i, \alpha. (\mu_\alpha v_\alpha)(\alpha) = \text{read} \Rightarrow \mu_i(\alpha) = \text{read} \) \land \( (\mu_i(\alpha) \in \text{co}) \cup \text{A} \Rightarrow \mu_\tau(\epsilon) \not\in \{\text{read}\} \cup \text{A} \)

\( \forall \alpha, \tau. \mu(\alpha) = \text{free} \land \mu(\alpha, \alpha) \in \text{A} \land \mu \circ \mu(\alpha, \alpha) \not\in \text{read} \Rightarrow \mu_\tau(\epsilon) \not\in \text{read} \)

\( \vdash (f : \mu_\tau \circ \mu_i d_i x_i, \mu_\tau \circ \mu d) \in \Sigma(\mu_\tau c) \)

where \( \mu(\epsilon) = m \) if \( \mu = m < \ldots \)

\( \mu(\alpha, \alpha) = \mu'(\alpha) \) if \( \mu = m < \ldots, \alpha = \mu', \ldots \)

otherwise

and \( \mu_\tau \circ \text{read}<\alpha_i = \mu_i > = \mu_\text{free}<\alpha_i = \mu_\tau \circ \mu_i > \)

\( \mu_\tau \circ \text{free}<\alpha_i = \mu_i > = \mu_\text{free}<\alpha_i = \mu_\tau \circ \mu_i > \)

\( \mu_\tau \circ \text{rep} <\alpha_i = \mu_i > = \mu_\text{free}<\alpha_i = \mu_\tau \circ \mu_i > \)

\( \mu_\tau \circ \text{free} <\alpha_i = \mu_i > = \mu_i \) if \( \mu_\tau = m < \ldots, \alpha = \mu', \ldots \)

8. Wellformed Program, Definition, Type Assignment \( \vdash p \text{ start } e_0 \), \( \vdash D \text{ defs } x \)

\( \vdash \Gamma \text{ ok} \)

\( \vdash D_1 \text{ defs } c_1 \cdots D_n \text{ defs } c_n \quad \forall i, j = 1, \ldots, n. \ c_i = c_j \Rightarrow i = j \)

\( \vdash FldsMths(c_n) = (R, F), \quad F(\text{main}) = \text{obs } \tau \text{ main() } \{ \ldots \} \)

\( \vdash D_1 \cdots D_n \text{ start new< } c_n \text{. main()} \)

\( \vdash M_1 \text{ defs } x_1 \cdots M_n \text{ defs } x_n \quad \forall i, j = 1, \ldots, n. \ x_i = x_j \Rightarrow i = j \)

\( \vdash \text{class } c \{ M_1 \cdots M_n \} \text{ defs } c \)

\( \vdash t \text{ ok} \quad \vdash t_i \text{ ok} \quad \vdash t_j \text{ ok} \)

\( \Gamma = \text{this: ref co<> } c, \ x_i: \text{ref } t_i, \ z_j: \text{ref } t_j' \quad \vdash \Gamma \text{ ok} \quad \Gamma, \kappa \vdash s : t \)

\( \vdash \kappa \ f (t_i, x_i) \{ t_j', z_j ; s \} \text{ defs } f \)

\( \vdash t \text{ ok} \quad \vdash \text{t } x; \text{ def } x \)

\( \forall i, j = 1, \ldots, n. \ x_i = x_j \Rightarrow i = j \)

\( \vdash x_1 : \tau_1, \ldots, x_n : \tau_n \text{ ok} \)

9. Term Typing Judgements \( \Gamma, \kappa \vdash e : \tau \)

\( \vdash \text{var< } x : \tau \in \Gamma \quad \vdash \text{var< } (\text{this: ref } \mu c) \in \Gamma \quad \vdash \text{FldsMths(c) = } \{ \ldots, x: \tau, \ldots, \}, F \)

\( \vdash \text{var< } \kappa, \tau \vdash \text{this.x} : \tau \)

\( \vdash \text{rd< } \kappa, \nu : \text{ref } \tau \quad \tau' = \tau[\text{read/ free}] \Rightarrow \kappa = \text{free}< \cdots c \Rightarrow \kappa = \text{obs} \land \nu \in \text{ Id} \)

\( \vdash \kappa, \nu \vdash \text{val}(\nu) : \tau' \)

\( \vdash \text{rd< } \kappa, \nu : \text{ref } \tau \quad \nu \not\in \text{ this } \quad \nu = \text{this.y} \Rightarrow \kappa = \text{mut} \)

\( \vdash \kappa, \nu \vdash \text{destval}(\nu) : \tau \)

\( \vdash \text{null< } \delta : \text{free< } \delta \cdot c \)

\( \vdash \text{new< } \delta \cdot c() : \text{free< } \delta \cdot c \)

233
\[
\begin{align*}
\Gamma, \kappa \vdash e : \mu & \vdash (f : \tau_i \rightarrow \kappa) \\
\kappa' = \text{mut} \Rightarrow \mu \in \text{Wr}(\kappa) \\
\Gamma, \kappa \vdash e_1 : \tau_i' \vdash \tau_i' \leq_m \tau_i \\
[\text{call}] \\
\end{align*}
\]

where \( \text{Wr(obs)} =_{df} \{\text{free<...>}\} \)

\( \text{Wr(mut)} =_{df} \{\text{free<...>,rep<...>,co<...>}\} \)

### A.3 Semantic Structures

#### 10. Semantic Domains

- **environment** \( \eta^e_b \in \text{Env} =_{df} (\text{Id} \leftrightarrow \text{Loc}) \times \kappa \times \mathcal{V} \)
- **store** \( s \in \text{Store} =_{df} \text{Loc} \leftrightarrow \mathcal{V} \)
- **object-map** \( \text{om} \in \text{Omap} =_{df} \emptyset \leftrightarrow (\text{Id} \leftrightarrow \text{Loc}) \times (\text{Id} \leftrightarrow \text{Mth}) \)
- **object graph** \( g \in \text{Graph} =_{df} \mathbb{N}^{\text{Loc} \times \mathcal{M} \times \mathcal{C}} \)
- **location (l-value)** \( \ell \in \text{Loc} =_{df} \bigcup_{\tau \in \mathcal{M} \times \mathcal{C}} \text{Loc}_\tau \)
- **handle (value)** \( h \in \mathcal{V} =_{df} (\emptyset \cup \{\text{nil}\}) \times \mathcal{M} \times (\emptyset \cup \{\text{nil}\}) \)
- **object-identifier** \( o \in \mathcal{O} =_{df} \bigcup_{c \in \mathcal{C}} \mathcal{O}_c \)
- **object value** \( \langle q, F \rangle \in (\text{Id} \leftrightarrow \text{Loc}) \times (\text{Id} \leftrightarrow \text{Mth}) \)
- **valid modes** \( \mu \in \mathcal{M} =_{df} \{\mu \mid \vdash \mu \text{ ok}\} \)

with infinite countable sets \( \mathcal{O}_c \) given for all \( c \in \mathcal{C} \) and \( \text{Loc}_\tau \) for all \( \tau \in \mathcal{M} \times \mathcal{C} \)

#### 11. Interpretation of Formal Type Terms and Type Consistency

\[
\begin{align*}
[\text{ref} \ \mu \ c] &=_{df} \text{Loc}_\mu c \\
[\mu \ c] &=_{df} (\emptyset \cup \{\text{nil}\}) \times \{\mu\} \times (\mathcal{O}_c \cup \{\text{nil}\}) \\
[\text{obj} \ c] &=_{df} \{\langle q, F \rangle \mid \vdash \text{FldsMths}(c) = \langle \Gamma, F \rangle \text{ and } \Gamma \vdash \Gamma\} \\
[\text{Cmd}] &=_{df} \{\epsilon\} \\
\eta \models \Gamma \iff_{df} \text{dom}(\eta) = \text{dom}(\Gamma) \land \forall x \in \text{dom}(\Gamma), \eta(x) \in [\Gamma(x)] \\
\models s \iff_{df} \forall \tau \in \mathcal{M} \times \mathcal{C}, \ell \in \text{dom}(s), \ell \in \text{Loc}_\tau \Rightarrow s(\ell) \in [\tau] \\
\models \text{om} \iff_{df} \forall c \in \mathcal{C}, o \in \text{dom}(\text{om}), o \in \mathcal{O}_c \Rightarrow \text{om}(o) \in [\text{obj} \ c] \\
\end{align*}
\]

234
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 $D_n \rightarrow ok \ c_n$

$e_0, \eta_0, s_0, om_0, g_0$ where $e_0 =_{df} new<>()\cdot main()

$\eta_0 =_{df} \emptyset_{\text{obs}}(\text{nil, read<>}, \text{nil})$

$s_0 =_{df} \emptyset$

$om_0 =_{df} \emptyset$

$g_0 =_{df} \emptyset$

13. Reduction Step Controlled by Reduction Context

\[
\frac{\mathcal{E} \in R_1 \quad e, \eta, s, om, g \Rightarrow e', \eta', s', om', g'}{
\mathcal{E}[e], \eta, s, om, g \Rightarrow \mathcal{E}[e'], \eta', s', om', g'}
\]

\[
\frac{\mathcal{E}[\langle e \rangle], \eta^*_0, s, om, g \Rightarrow \mathcal{E}[\langle e' \rangle], \eta^*_0, s', om', g'}{
\mathcal{E}[\langle e \rangle], \eta^*_0, s, om, g \Rightarrow \mathcal{E}[\langle e' \rangle], \eta^*_0, s', om', g'}
\]

\[
R_1^\square := \emptyset
\]

$\mid \text{val}(R_1^\square) \mid \text{destval}(R_1^\square) \mid R_1^\square \leftarrow \text{Id}(E^*), \mid \nu \leftarrow \text{Id}(\nu^*), R_1^\square(, E^*)$

$\mid R_1^\square S \mid R_1^\square = E; \mid \text{Loc} = R_1^\square, \mid \text{return} R_1^\square, \mid \text{if}(R_1^\square \Psi R)\{S\} \mid \text{if}(\nu \Psi R_1^\square)\{S\}$

14. Reduction at the Redex, where $\eta^*_0$ is the top element in $\eta$

\[
\begin{align*}
\{\text{var}\} & \quad \eta(x) \triangleq \ell \\
\quad & \quad \frac{x, \eta^*_0, s, om, g \rightarrow \ell, \eta^*_0, s, om, g}{\eta_{\text{(this)}}(x) \triangleq \ell} \\
\quad & \quad \frac{\text{this.s}, \eta^*_0, s, om, g \rightarrow \ell', \eta^*_0, s, om, g}{s(\ell) \triangleq \langle o, \mu, \omega \rangle} \\
\quad & \quad \frac{\mu' = \mu[\text{read}/\text{free}] \quad \text{val}(\ell), \eta^*_0, s, om, g \rightarrow \langle o, \mu', \omega \rangle, \eta^*_0, s, om, g \oplus o \mu', \omega}{{\text{rd.exp}}} \\
\quad & \quad \frac{\text{destval}(\ell), \eta^*_0, s, om, g \rightarrow \langle o, \mu, \nu \rangle, \eta^*_0, s, \ell \mapsto o \mu, \nu}{{\text{rd.val}}} \\
\quad & \quad \frac{\text{null} \triangleq \langle s, \mu, r \rangle \quad \text{null} < \delta > \triangleq \eta^*_0, s, om, g \rightarrow (r, \text{free} < \delta >, \text{nil}), \eta^*_0, s, om, g}{{\text{null}}} \\
\quad & \quad \frac{h \triangleq \langle s, \mu, r \rangle \quad F \text{ldsMth}(c) \triangleq \langle \{ x_i : \text{ref} \mu_i \ni c_i \}, F \rangle \quad h' = \langle r, \text{free} < \delta >, \nu \rangle \quad \text{fresh} o \in \mathcal{O}_c \quad \text{fresh} \ell_i \in \{ \text{ref} \mu_i, c_i \}}{\{\text{new}\} } \\
\quad & \quad \frac{\text{fresh} o \in \mathcal{O}_c \quad \text{fresh} \ell_i \in \{ \text{ref} \mu_i, c_i \} \quad \frac{\{ x_i : \text{ref} \mu_i \ni c_i \}, F \rightarrow \{ x_i : \ell_i \}}{\text{new} < \delta > c()}, \eta^*_0, s, om, g \rightarrow h', \eta^*_0, s, \ell_i \mapsto h_i, om[o \mapsto \langle o, F \rangle], g \oplus h'}{\text{new} < \delta > c()}}
\end{align*}
\]
\[ r \in \mathcal{O}_c, \quad \text{om}(r) = \langle \ldots, F \rangle, \quad F(f) = \kappa^* \tau f (\mu_i, c_i, \overline{y_i}) \{ \mu_j, c_j \overline{z_j}; s \} \]

fresh \( \ell \in \llbracket \text{ref co}< \rrbracket \), fresh \( \ell_1^q \in \llbracket \text{ref } \mu_i \rrbracket \), fresh \( \ell_2^q \in \llbracket \text{ref } \mu_j \rrbracket \)

\( \eta^* = \{ \text{this } \mapsto \ell, y_i \mapsto \overline{y_i}; z_j \mapsto \overline{z_j} \} \)

\( s' = s[\ell \mapsto \{ r, c_1, \overline{y_1}, r \}, \ell_1^q \mapsto \{ r, \mu_i, \overline{y_i}, \}, \ell_2^q \mapsto \{ r, \mu_j, \overline{z_j}, \text{nil} \}] \)

\( g' = g \uplus s \mu_i^{\mu_i}, o_i \uplus r \text{ co}< \rightarrow r \uplus r \mu_i \rightarrow o_i \)

\( (s, \mu_t, r) \leftarrow f(\langle s, \mu_t^q, o_i \rangle), \eta^*_h, s, \text{om}, g \rightarrow \llbracket s \rrbracket, \eta^*_h \cdot \eta^*_l \cdot (s, \mu_t, r), s', \text{om}, g' \)

\( \{	ext{call}\} \)

\( s' = s[\ell \mapsto \perp | \ell \in \text{imap}(\eta^*)] \quad g' = g \uplus s \mu_r^{\mu_r}, o \uplus s \mu_r \rightarrow r \uplus r \mu_r \rightarrow o \uplus \text{om}(\text{imap}(\eta^*)) \)

\( \llbracket \text{return } (r, \mu, o) \rrbracket, \eta^*_h \cdot \eta^*_l \cdot (s, \mu_r, r), s, \text{om}, g \rightarrow \langle \langle s, \mu_r \cdot o, \rangle, \eta^*_h, s', \text{om}, g' \rangle \)

\( \{	ext{upd}\} \)

\( \ell = (o, \mu, \overline{\omega});, \eta^*_h, s, \text{om}, g \rightarrow \epsilon, \eta^*_h, s[\ell \mapsto \langle o, \mu, \overline{\omega} \rangle], \text{om}, g \uplus o \mu_r \rightarrow \overline{\omega} \rightarrow s(\ell) \uplus o \mu_r \rightarrow \overline{\omega} \)

\( \{	ext{if} \} \)

\( \psi(\langle o, \mu, \omega \rangle \psi(\langle o, \mu', \omega' \rangle \{ s \}), \eta^*_h, s, \text{om}, g \rightarrow s, \eta^*_h, s, \text{om}, g \uplus o \mu_r \rightarrow \omega \rightarrow o \mu_r \rightarrow \omega' \)

\( \{	ext{if} \} \)

\( \psi(\langle o, \mu, \omega \rangle \psi(\langle o, \mu', \omega' \rangle \{ s \}), \eta^*_h, s, \text{om}, g \rightarrow \epsilon, \eta^*_h, s, \text{om}, g \uplus o \mu_r \rightarrow \omega \rightarrow o \mu_r \rightarrow \omega' \)

\( \{	ext{wh} \} \)

\( \text{while}(e_1, e_2) \{ s \}, \eta^*_h, s, \text{om}, g \rightarrow \text{if } (e_1, e_2) \{ s \text{ while } (e_1, e_2) \{ s \}, \eta^*_h, s, \text{om}, g \}

where \( \llbracket = \rrbracket (\omega, \omega') \leftrightarrow \epsilon \omega = \omega' \) and \( \llbracket != \rrbracket (\omega, \omega') \leftrightarrow \epsilon \omega \neq \omega' \)

15. Helper Functions

\[
g \uplus o \mu \omega =_{df} \begin{cases} \exists \text{ if } \epsilon \in \{ o, \omega \} \\ \exists \text{ otherwise } \end{cases}
\]

where \( \uplus \) is multiset-union and \( \setminus \) 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 $\nu$ is explicitly specified by $\text{val}(\nu)$ or $\text{destval}(\nu)$.

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 $\text{this}$. Returning $\text{this}$ is necessary for calling these mutator methods through a (destructively read) free reference without losing it forever. Since there is no boolean, SetImp's $\text{contains}$ operation returns null if the given object $o$ was not found in the set, and returns an $\text{elem}$ 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, $\text{res}$ variables are sometimes needed to transport the result to the end of the method ($\text{current}$, $\text{contains}$, $\text{lookup}$). 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).

```plaintext
/*********************** 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 {

237
```
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 PNodelt {
    dest<> PNode curnode;
    mut co<> PNodelt StartAt(dest<> PNode n) { this.curnode = val(n); return val(this); }
    mut co<> PNodelt Step() { this.curnode = val(this.curnode)<=next(); return val(this); }
    obs dest<> PNode current() { return val(this.curnode); }
}

// iterator over Pairs in a PNodelt's PNodes
class PDatalt {
    rep<dest=read<data=dest<>> PNodelt nodes;
    mut co<> PDatalt Wrap(free<dest=read<data=dest<>> PNodelt nn) { this.nodes = destval(nn); return val(this); }
    mut co<> PDatalt 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 PSetlmp {
    rep<data=elem<> PNode anchor;
    mut co<> PSetlmp 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<> PSetlmp 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<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) ); }
    }
}
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).

```plaintext
//******************** 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; }
```

242
// 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; }
}

/***************************** 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 );
        }
    }
}

243
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

[Alm97] Paulo Sergio Almeida: Balloon Types: Controlling Sharing of State in Data Types; ECOOP'97; LNCS 1241; Springer 1997.
[Ame87] Pierre America: Inheritance and subtyping in a parallel object-oriented language; ECOOP'87; LNCS 276; Springer 1987.
[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.