

ANALYSIS AND COMPILATION TECHNIQUES FOR
HARPO/L

LI XIANGWEN



**Analysis and compilation techniques for
HARPO/L**

by

© Li Xiangwen

A thesis submitted to the
School of Graduate Studies
in partial fulfilment of the
requirements for the degree of
Master of Engineering

Faculty of Engineering and Applied Science
Memorial University of Newfoundland

September 2008

St. John's

Newfoundland

Abstract

Coarse grained reconfigurable architecture (CGRA) is a reconfigurable architecture that uses word-width processing elements, and provides custom designed reconfigurable datapath units (rDPUs) as basic logic units. It combines some strength of both software and hardware to provide an easy to develop, and highly efficient platform. In this thesis, I make contributions to the development of an object oriented language, HARdware Parallel Object Language (HARPO/L) , which is suitable to describe the parallel execution of hardware, and hence can be compiled directly to CGRA platform.

This thesis will mainly concentrate on the front-end of the HARPO/L compiler, to address technical issues arising from some of the unique characteristics of our language. This thesis will develop a formal mathematical representation for HARPO/L, to help verify the semantics of the language. It will also develop a method to identify synchronization problems in shared variable access, and to simplify the implementation of atomicity in language. Furthermore, in addition to the research work in this thesis, a compiler front-end is also implemented in JAVA to compile the plain text source code to typed abstract syntax tree (AST). We will also discuss some techniques involved in implementing such a compiler front-end.

Acknowledgments

First of all, I would like to thank my supervisor, Dr. Theodore S. Norvell, for his guidance through my three-year graduate study in Memorial University of Newfoundland in both my current and previous degrees, and his continuous support in the research work and the writing of this thesis. I would like to also thank my classmates in Master of Applied Science Computer Engineering program. Your friendship encourages me to continue my study in a country so far away from home.

I am also grateful to my wife, Li Jing, for her love and company in Canada.

Finally, I dedicate this thesis to my parents, for their love and support through my whole life.

Contents

Abstract	ii
Acknowledgments	iii
List of Figures	x
1 Introduction and Background	1
1.1 Introduction to Reconfigurable Architecture	1
1.2 Introduction to HARPO/L	3
1.3 Structure of HARPO/L project	7
2 Language Design	9
2.1 Meta notation	9
2.2 Type system	10
2.2.1 Primitive type	10
2.2.2 Array	12
2.2.3 Interface	13
2.2.4 Class	14
2.2.5 Conditional Initiation	16

2.3	Statement	16
2.3.1	Thread	16
2.3.2	Variation of statements	17
2.4	Genericity	20
2.5	Example	21
3	Colored Petri Net representation of HARPO/L	23
3.1	Motivation	23
3.2	Introduction to Colored Petri Net	25
3.2.1	Petri Net	25
3.2.2	Colored Petri Nets	27
3.3	Basic elements of the CPN representation of HARPO/L	28
3.3.1	Token color sets for the representation	28
3.3.2	Basic transition types	29
3.4	Control flow of CPN representation of HARPO/L	33
3.4.1	Sequential control flow	34
3.4.2	Branch control flow	34
3.4.3	Loop control flow	35
3.4.4	Parallelism	37
3.5	Variable operation	37
3.5.1	Temp variable	38
3.5.2	Local variable	38
3.5.3	Shared variable	40
3.5.4	Array variable	43

3.6	Method calls	44
3.7	Atomic blocks	46
4	safety of fission	48
4.1	Motivation	48
4.2	Introduction to trace theory	51
4.2.1	Projection	52
4.2.2	Weaving	53
4.2.3	Equivalence of traces	55
4.2.4	Definition-use chain	56
4.2.5	Equivalence relationship between two symbol traces	56
4.2.6	Equivalence of trace sets	58
4.3	safety of fission	59
4.3.1	Fission and fusion	59
4.3.2	safety	63
4.3.3	Proof of equivalence of traces	64
4.3.4	Equivalence of result sets	66
4.3.5	Proof of decomposability of the context	68
4.4	Applying the method	70
4.4.1	Multiple variables	70
4.4.2	Loop and branch structure	71
4.4.3	Mutual exclusion	73
5	Implementation of the compiler front-end	74
5.1	Structure of the compiler front-end	74

5.1.1	Syntax analyzer	74
5.1.2	Type checker	76
5.1.3	Specialization/ instantiation	77
5.2	Type system	78
5.2.1	Types and expressions	79
5.2.2	Type dependence	82
5.3	Specialization & Instantiation	86
5.4	Implementation	89
5.4.1	Parsing	89
5.4.2	Representing types	94
5.4.3	Expressions	97
5.5	Working examples	100
6	Conclusion and future work	107
6.1	Thesis summary	107
6.2	Thesis Contribution	109
6.3	Open issues for future work	110
A	Language Design for CGRA project. Design 5 [Draft].	111
A.1	Classes and Objects	112
A.1.1	Programs	112
A.1.2	Types	112
A.1.3	Objects	113
A.1.4	Constants	115
A.1.5	Classes and interfaces	115

A.1.6	Class Members	116
A.1.7	Interface Members	117
A.2	Threads	118
A.2.1	Statements and Blocks	118
A.3	Expressions	121
A.4	Genericity	121
A.5	Examples	122
A.6	Lexical issues	123
B	The Static Semantics of HARPO/L	124
B.1	Abstract Syntax	124
B.2	Types	124
B.2.1	Typing relation	124
B.3	Building a class environment	126
B.4	Types of expressions	128
B.4.1	Identifiers are looked up in the context	128
B.4.2	Constants	128
B.4.3	Arithmetic expressions	128
B.4.4	Arrays	129
B.4.5	Inheritance	130
B.4.6	Fields and methods	130
B.4.7	Initialization Expressions	131
B.5	Type checking types	132
B.5.1	Primitives	132

B.5.2	Class and interfaces	132
B.5.3	Array types	133
B.5.4	Generic parameters	133
B.6	Type checking of commands	133
B.6.1	Assignments	134
B.6.2	Local variable declaration	134
B.6.3	Blocks	134
B.6.4	Method calls	134
B.6.5	Sequential control flow	134
B.6.6	Parallelism	135
B.6.7	Method Implementation	135
B.6.8	Atomicity	135
B.7	Type Checking Declarations	135
B.7.1	Class declarations	135
B.7.2	Interface declarations	135
B.7.3	Global object and field declarations	135
B.7.4	Method declarations	135
	Bibliography	136

List of Figures

1.1	The overall structure of HARPO/L project.	7
3.1	A simple petri net.	25
3.2	Switch transition	30
3.3	Copy transition	31
3.4	Merge transition	31
3.5	Product transition	31
3.6	Split transition	32
3.7	Operator transition	32
3.8	Write transition	33
3.9	Sequential execution of statements	34
3.10	Branch control flow	35
3.11	Loop control flow	36
3.12	Parallel control flow	37
3.13	Example of using a temp variable in expression $a + b + c$	38
3.14	Local variable subpage	39
3.15	An example of local variable read and write action	40
3.16	Writing of shared variable	41

3.17	Local part of the shared variable writing	42
3.18	Reading of local array	43
3.19	Called thread part of method call	45
3.20	CPN representation for atomic block	47
4.1	Example for loop structure that is not safe for fission.	71
4.2	Example for loop structure that is safe for fission.	72
4.3	Example for branch structure that is safe for fission.	72
4.4	Example or thread divided by semaphores	73
5.1	Structure of HARPO/L compiler front-end	75
5.2	Abstract Syntax Tree for $a := a + 1$	75
5.3	Typed Abstract Syntax Tree for $a := a + 1$	76
5.4	Example of symbol table hierarchy	78
5.5	Different types in the type system	79
5.6	Structure of an Array Type	80
5.7	Expression type for a.d	82
5.8	A simple example of type dependence	83
5.9	Symbol table entries and dependence relationship	84
5.10	An example of cyclic dependence	85
5.11	Symbol dependency stack for the cyclic dependency example	86
5.12	Structure for Specialized Class Type	88
5.13	Symbol tables of generic and specialized table	88
5.14	An example of parse tree	92
5.15	Class diagram for class <i>Type</i>	94

5.16	Class diagram for class <i>BaseType</i>	96
5.17	Class diagram for <i>ExpressionType</i>	98
5.18	Class diagram for <i>Expression</i> , <i>Operator</i> and <i>OperatorType</i>	99

Chapter 1

Introduction and Background

1.1 Introduction to Reconfigurable Architecture

When solving a specific problem using digital computing technology, there are two conventional methods: hardware and software. One of the most popular solutions of hardware implementation is Application-Specific Integrated Circuits (ASIC). ASICs are hardware chips built to solve a specific problem, hence they are very efficient when dealing with the problem they are designed for. Despite its high efficiency, ASIC has two major drawbacks: long development time and low flexibility. Designing and optimizing an ASIC circuit usually takes a lot of time, and once the circuit is fabricated, modification to any part of the circuit or the target application itself will result in a redesign of the whole circuit.

The second method is using software. Software utilizes microprocessors, which can perform a number of different operations depending on the instruction. This solution is very flexible because the major part of the system, the microprocessor,

remains unchanged from application to application. The user only needs to change the instructions to make the system suitable for different applications. However, the efficiency of such a solution is far lower than that of ASIC, because the processor needs to perform a series of extra operations, such as instruction fetching and decoding, to execute a single instruction. Moreover, the datapath in this solution is much longer than hardware implementations that are optimized for the specific problem.

Reconfigurable computing is intended to fill the gap between hardware and software, achieving potentially much higher performance than software, while maintaining a higher level of flexibility than hardware[1]. Reconfigurable devices contain a number of logic blocks that can be configured to perform different tasks and a set of configurable routing resources to connect logic blocks together. Hence, a custom circuit can be mapped into a configuration specifying the function of logic blocks and the wiring between them.

One of the most popular reconfigurable architectures is Field-Programmable Gate Array (FPGA) [12], which is a fine-grain reconfigurable architecture that combines several layers of AND-OR gates to implement its logic. Although this technique can be used to develop very efficient circuits, it also has some major drawbacks comparing to coarse-grained approaches.

Coarse-Grained Reconfigurable Architectures (CGRA) are reconfigurable architectures with datawidth greater than 1 bit.[4] CGRAs provides custom designed reconfigurable datapath units (rDPUs) as basic logic units. So instead of working on gate level configuration for basic computations in FPGA, the programmer can utilize these rDPUs to implement the application. Because these custom designed rDPUs have higher performance, area and energy efficiency than computation units assem-

bled from single bit Configurable Logic Blocks (CLB) in FPGA, the whole circuit built in this way will also be more efficient than the fine grain implementation.

The other major advantage of CGRA is that the placement and routing problems are greatly simplified by reducing the number of reconfigurable units. This will also result in a massive reduction of configuration memory and configuration time. Our goal in the HARPO project is to define a hardware development method that is as easy to code and configure as software solutions that produces high performance circuits. Therefore, we find CGRAs to be a very suitable class of target architectures.

1.2 Introduction to HARPO/L

HARPO stands for HARdware Parallel Objects Language. HARPO/L is an object oriented language that can be used to create hardware configurations [8]. When creating a hardware configuration, instead of working on electronic circuits directly or using some Hardware Description Language (DSL), the user can choose to use HARPO/L, which is a high level, object oriented language similar to C. This solution brings the simplicity of software coding into the otherwise cumbersome hardware configuration development.

Although there are approaches that exist to compile high level languages such as C[7][14] and Java[3] into reconfigurable architectures, most of them suffer from the same problem[2]: the source language, which is designed to describe sequential program running on microprocessors, is not suitable for describing the behavior of hardware effectively.

HARPO/L is hence a language designed to fulfill this task. As a result of this

objective, it contains a number of features and characteristics that are different from typical high level programming languages.

The first difference is the support of user-specified parallelism. Typical high-level programming languages are meant to be compiled and executed on microprocessor machines, which fetch and execute instructions one by one in a sequential order. This fact makes these languages only optimal for describing sequential behaviors. The level of parallelism the compiler can exploit from programs written in this way is hence very limited. However, because of the parallel nature of hardware, programs written in this way are very inefficient for hardware implementation.

For example, when performing an array operation on a microprocessor, because the processor is only capable of executing one instruction at a time, the operation to multiple array elements must be performed one by one. This is commonly represented in typical high level languages by a loop structure.

```
for(int i=0;i<length;i++)  
    array[i]++;
```

These loop structures are usually the most time consuming part of the whole program. However, in hardware, the programmer can create a much more efficient implementation of the same problem with concurrent execution. This requires the compiler to identify possible parallelism within source code to be efficient. This is often not possible without the input of the programmer.

To deal with this problem, HARPO/L defines a `co` statement to support explicit parallelism. Programmers can manually indicate portions of the code to be executed

in parallel. With this new feature, the code in the above example can be written as:

```
(co i:length do array[i] := array[i]+1 co)
```

This line of code indicates that different iteration of the loop body “array[i] := array[i]+1” will not interfere with each other and asks the compiler to execute these instructions in parallel if possible. With this extra information provided, the compiler can utilize a much higher level of parallelism to significantly improve the performance of the hardware configuration generated.

The second difference between hardware and software is the representation of objects. In software, an object is a block of memory that contains the value of all the fields for the object. Objects can be dynamically instantiated by allocating a block of memory at run time. However, in hardware, an object is a concrete block of gates and instruments, which must be created by the time the hardware configuration is generated. In typical object oriented languages, objects can be defined and instantiated dynamically, this can be implemented without any difficulty by memory operations. However, this is not possible for hardware implementation.

This difference also causes problem in object reference. In software, object can be accessed by its reference, which is a pointer to the address where the object is located in memory. References can also be assigned to point to another object with a same type. This behavior is also not possible in hardware implementation, where access to a logic block must be carried out by actual wiring. The connection on board cannot be modified once the configuration is generated. This constraint requires all objects in HARPO/L to be instantiated at compile time, and that object references can not

be assigned once initialized.

The implementation of function call in HARPO/L is also different from typical programming language. In software, a class may contain a number of different functions. Multiple functions can be called at the same time, and one function can be called by several different threads at the same time. Moreover, one function can call itself for recursive algorithms. This is not easy in hardware because, in hardware, a function call is a set of instruments containing the necessary units and wiring for the body of the function, these instruments can not be utilized by multiple threads at the same time. This fact also makes the structure of HARPO/L classes and functions very different from typical programming language.

In addition to these characteristics that come from the nature of hardware, HARPO/L also contains a number of other features worth mentioning:

- HARPO/L allows implicit type definition that permits type information to be omitted from the object definition. This feature improves the flexibility of programming by allowing operations to be performed on different data types using same block of code.
- HARPO/L employs a generic mechanism to provide polymorphism for class definitions. This feature further enhances the flexibility of programming by bringing in polymorphism, which will also greatly reduce the work involved in dealing with different data types.
- HARPO/L also supports explicit indication of atomicity. Instead of having to write a number of complex program controls such as semaphore and monitor,

the user can write a single statement to request a block of code to be executed atomically.

1.3 Structure of HARPO/L project

A compilation system is also being implemented for HARPO/L.

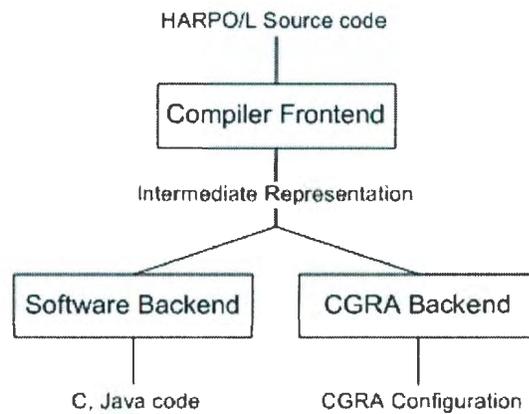


Figure 1.1: The overall structure of HARPO/L project.

As shown in the figure above, the HARPO/L system mainly consist of 3 parts: compiler's front end, software back-end and hardware back-end. As any typical compilation system, the HARPO/L compiler takes the plain text HARPO/L code defined in language design[8] as input to the system. The front-end then makes lexical analysis based on the language design to verify if the source code is using correct HARPO/L grammar. The front-end also performs semantic analysis according to language semantics[9] to find the meaning of the input code, and verify if the meaning is legal in language definition.

The output of the compiler front-end is an intermediate representation, called an object graph. The object graph is a data structure reflecting all information contained in the source code, with additional information such as variable type and execution flow being derived from context and added to their corresponding position. This intermediate representation is platform independent, and can be used alone to generate executable code, once the information about target platform is available.

The HARPO project targets two platforms: software and hardware. The hardware back-end is the major objective of this project. Used in combination with the front-end, it is capable of compiling programs written in HARPO/L directly onto reconfigurable architectures such as CGRA. The software back-end is used to generate code in languages with available compiler such as C. Results generated by this back-end can be used to debug and verify the correctness of the source code without having to first download it onto hardware boards.

The rest of this thesis is organized as follows: chapter 2 gives a detailed introduction on HARPO/L language design. Chapter 3 uses a mathematical representation — the Petri Net — to describe this language. Chapter 4 will discuss a way of analyzing the safeness of shared variable access. The techniques employed in the implementation of the front-end will be discussed in chapter 5. Chapter 6 concludes the thesis and provides future research directions.

Chapter 2

Language Design

2.1 Meta notation

This chapter used a number of meta notations to describe the language grammar, which are explained as follows:

$N \rightarrow E$	Nonterminal N can be an E
$\underline{(E)}$	Groups the enclosed content into one item
E^*	Zero or more E s
E^*F	Zero or more separated by F s
E^+	One or more E s
E^+F	One or more separated by F s
$E^?$	Zero or one E s
$\underline{[E]}$	Zero or one E s, same effect as above meta notation
$E F$	Choice of either E or F

Note that underlines are added to parentheses and brackets here to distinguish

them from their normal occurrence. So for example, when parentheses without underlining is encountered in the grammar rules, it means there should be literally a “()” pair in source code.

2.2 Type system

A HARPO/L program is a set of class, interface and object declarations. Each object in HARPO/L is characterized by its type.

2.2.1 Primitive type

Primitive types are predefined basic types that are used to represent values. They are the only types in HARPO/L that can be assigned to after being initiated.

Primitives type includes 3 major categories: integer, real and boolean. Integer and real types also have a number of subtypes with different precision. For example, an int16 is an integer value that is represented by a 16 bit bit-vector (16 bit precision) in hardware configuration. A complete list of primitive types is shown as follows:

- int8, int16, int32, int64, int
- real16, real32, real64, real
- bool

The declaration of a primitive variable should be in the following form:

$$\underline{(\text{obj} \mid \text{const})} \text{ Name } [: \text{Type}] := \text{Exp} \quad (2.1)$$

where *Name* is a string that contains letters, digits and underlines and starts with a letter, used to represent the name of the object. *Exp* is a compile time constant expression that is assignable to the type of this variable.

To be compile time constant means the value of such variable or expression can be determined at compile time. The constantness of expressions is determined by the following rule:

- Literals are constant. Literals are piece of data that are taken literally as a value, such as *10*, *12.5*, *false*.
- Variables defined by keyword **const** are constant. As shown in (2.1), instead of using **obj** keyword, the user can use **const** keyword to indicate the variable is constant thus can not be assigned to once initiated.
- Expressions formed by constant variables and expressions only are also constant, since their value can be computed from their constant sub-expressions.

The type information in (2.1) can be omitted. In this case, the type of the variable is the primitive type with smallest precision that can take the value. For example, for integers, we have following rules:

$$\begin{aligned}
 \{-128, \dots, +127\} &\in \text{int8} \\
 \{-2^{15}, \dots, +2^{15} - 1\} &\in \text{int16} \\
 \{-2^{31}, \dots, +2^{31} - 1\} &\in \text{int32}
 \end{aligned}
 \tag{2.2}$$

As a result, the initiation expression

obj *length* := 512

causes variable *length* to have type *int16*, because according to the integer typing rule, *int16* is the integer with the smallest precision that can take value 512.

Furthermore, a primitive type with lower precision is subtype of primitive type with higher precisions shown in (2.3), the $<$: operation indicates the former type is a subtype of later. Subtyping is transitive and reflexive, but not commutative.

$$\begin{aligned} int8 <: int16 & \quad int16 <: int32 \\ real16 <: real32 & \quad real32 <: int64 \end{aligned} \quad (2.3)$$

This rule means a primitive type with lower precision can be assigned to those with higher precision, but not the opposite. When a binary operator involves operands with a different precision, the lower one must be widened to be the same with the other operand. This fact effectively means the outcome of an expression is determined by the operand with the highest precision.

2.2.2 Array

Arrays are lists of objects which can be declared in following form:

$$\mathbf{obj} \ Name \ [: \ Type] := \left(\mathbf{for} \ Name : \ Bounds \ \mathbf{do} \ InitExp \ [\mathbf{for}] \right) \quad (2.4)$$

- *Bounds* is a compile time constant integer expression.
- *InitExp* can be a simple expression, an array initial expression, or a new object creation (discussed in section 2.2.4).

Multi-dimensional array can be declared by nested array initial expressions. The following example defines a 10*10 array.

$$\mathbf{obj} \ array2D := (\mathbf{for} \ i : 10 \ \mathbf{do} \ (\mathbf{for} \ j : 10 \ \mathbf{do} \ i * 10 + j \ \mathbf{for}) \ \mathbf{for})$$

Array items can be accessed by $Name[Index]$, where $Index$ is an integer expression with value less than $Bounds$ of the array. The type of the array items acquired this way is the type of its $InitExp$, and can only be assigned by values with type being subtype of $InitExp$ if it is assignable (being primitive type).

2.2.3 Interface

An interface is the definition of an abstract class without implementation. An interface can be defined in following form:

$$\begin{aligned}
 &(\mathbf{interface} \textit{ Name } GParams^? (\mathbf{extends} \textit{ Type}^+)^? (\textit{IntMember})^* \\
 &[\mathbf{interface} \textit{ [Name]}]) \tag{2.5}
 \end{aligned}$$

- $GParams$ are parameters used for generic expansion, which will be discussed in section 2.4.
- $Type$ is the interface this interface extends. An interface can only extend other interfaces.
- $IntMember$ is the member of the interface, which can be either a field or a method declaration.

A field is an object member declared inside a class or interface. A field declaration is an object declaration plus an access keyword specifying that it is a public or private field.

$$\begin{aligned}
 &Access \mathbf{obj} \textit{ Name}[: \textit{ Type}] := \textit{ InitExp} \\
 &Access \rightarrow \mathbf{private} \mid \mathbf{public} \tag{2.6}
 \end{aligned}$$

A method declaration is a declaration of method parameters without its implementation. The implementation should be inside a thread structure of each class that implements this interface. Thread and method implementation will be discussed in section 2.2.4.

$$\begin{aligned} & \text{Access } \mathbf{proc} \text{ Name}(\underline{Direction} \text{ [Name :] } \underline{Type}^*) \\ & \text{Direction} \rightarrow \mathbf{in} \mid \mathbf{out} \end{aligned} \tag{2.7}$$

keyword *in* declares input parameter while *out* declares output.

For example, the following code block declares an interface with two public methods:

```
(interface Queue
    public proc deposit(in value : int)
    public proc fetch(out value : int)
interface)
```

2.2.4 Class

A class is a user defined structure that specifies a family of types. A class can be defined in following form:

$$\begin{aligned} & (\mathbf{class} \text{ Name } \underline{GParams}^? \text{ (implements } \underline{Type}^+)^? \mathbf{constructor}(\underline{CPar}^+)) \\ & \underline{ClassMember}^* \text{ [class [Name]]} \end{aligned} \tag{2.8}$$

- *ClassMember* is the content of the class. In addition to method and field already discussed in section 2.2.3, it can also contain **thread** component. Thread will be discussed in section 2.3.
- *GParams* are parameters used in generic expansion. Generic classes will be discussed in section 2.4.
- Each *Type* is an interface this class implements. A class must implement all methods from all interfaces it implements.
- *CPar* are the constructor parameters the class. A constructor parameter can be either a constant or an object.

$$\mathbf{obj} \text{ Name} : \text{Type} \mid \mathbf{in} \text{ Name} : \text{Type} \quad (2.9)$$

in parameters are treated as constant value in class body, so the corresponding argument must also be compile time constant. *Object* parameters are objects connected to new object of this class in initialization. It will be known inside class body as *Name*.

A new object of a class can hence be declared as:

$$\mathbf{obj} \text{ Name} \underline{[: \text{Type}]} := \text{Type}(\text{CArg}^+)$$
 (2.10)

CArg is the argument corresponding to *CPar* in class definition.

2.2.5 Conditional Initiation

In addition to initialization statement for primitive, array and object, HARPO/L also contains a special form of conditional initialization.

$$\mathbf{obj} \textit{ Name} \textit{ [: Type]} := \left(\mathbf{if} \textit{ Exp} \mathbf{ then} \textit{ InitExp} \textit{ (else if} \textit{ Exp} \textit{ InitExp)}^* \mathbf{ else} \textit{ InitExp} \textit{ [if]} \right) \quad (2.11)$$

- *Exp* must be a compile time constant boolean expression.
- *InitExp* can be initiation expression for any type. Multiple *InitExp* in a same conditional initiation do not have to be the same type.

2.3 Statement

Statements are the basic elements of executable code. Statements must be embedded within a thread in HARPO/L grammar.

2.3.1 Thread

A thread is the executable part of class. A class may contain zero or more threads; all threads are executed concurrently. Parallelism can be achieved by having multiple threads in a class.

$$(\mathbf{thread} \textit{ Block} \textit{ [thread]}) \quad (2.12)$$

where the *Block* is just a sequence of statements:

$$\textit{ (Statement | ;)}^* \quad (2.13)$$

2.3.2 Variation of statements

- Local variable declaration

Local variable declaration is exactly the same as global variables discussed in last section, except the scope of the variable is the block it is declared.

- Assignment

$$ObjectId _ , ObjectId _)^* := Expression _ , Expression _)^* \quad (2.14)$$

The number of *ObjectId* must be the same as the *Expression*, the type of *ObjectId* can only be primitive. The *Expressions* are assigned to their corresponding *ObjectIds* according to the order they appear in the sequence.

The *ObjectId* can be either a variable, an array index, or a field reference.

$$Name \mid ObjectId[Expression] \mid ObjectId.Name \quad (2.15)$$

- Sequential control flow

$$\begin{aligned} & \left(\text{if } Expression \text{ then } Block _ (\text{else if } Expression \text{ Block})^* (\text{else } Block _)^? [\text{if}] \right) \\ & \mid \left(\text{wh } Expression \text{ do } Block _ [\text{wh}] \right) \\ & \mid \left(\text{for } Name : Bounds \text{ do } Block _ [\text{for}] \right) \end{aligned} \quad (2.16)$$

Sequential control flow in HARPO/L is the same as in typical high level languages.

- Parallelism

$$\begin{aligned} & \left(\mathbf{co} \textit{Block} _ || \textit{Block} _ * _ [\mathbf{co}] \right) \\ & | \left(\mathbf{co} \textit{Name} : \textit{Bounds} \mathbf{do} \textit{Block} _ [\mathbf{co}] \right) \end{aligned} \quad (2.17)$$

As discussed in chapter 1, HARPO/L utilizes **co** statement to support explicit parallelism. The compiler will execute code blocks within **co** structure concurrently. Ensuring the code blocks can be executed safely in parallel without shared variable conflict or synchronization problem is the responsibility of the programmer. The former form is used for blocks with distinct content, the latter form is for parallel execution of loop iterations such as **for** loop in typical high level language.

- Sequential consistency

$$\left(\mathbf{atomic} \textit{Block} _ [\mathbf{atomic}] \right) \quad (2.18)$$

The block inside **atomic** structure must be executed as if atomically. That means, it can only be either fully executed or not executed at all in other thread's point of view, but not partly executed. This property requires the system to employ some special mechanism for sequential consistency, which may seriously affect the system performance if not implemented carefully. Detailed discussion about techniques involved in this problem will be discussed in chapter 4.

- Method implementation.

$$\left(\text{accept } \underline{MethodImp} \ (| \ \underline{MethodImp})^* \ \underline{\text{accept}} \right)$$

$$\underline{MethodImp} \rightarrow \underline{Name} \left(\underline{Direction} \ \underline{Name} : \underline{Type} \right)^* \ \left[\underline{Guard} \right] \ \underline{Block}_0 \ \left[\underline{\text{then}} \ \underline{Block}_1 \right]$$

$$\underline{Guard} \rightarrow \underline{\text{when}} \ \underline{Expression} \tag{2.19}$$

- The method parameters must match the declaration in class body.
- *Guard* is a boolean expression.
- The block after keyword **then** will be executed after the called thread returns output parameters to the calling thread.
- Each method can only be implemented once in class body.

Although the syntax is similar, method implementation in HARPO/L is very different from that of typical high level languages. While threads can call methods of other threads at any time, the method call will not be handled unless the sequential execution flow of the called thread reaches the corresponding **accept** structure. A thread which reaches an **accept** statement will wait until the method is called and *Guard* expression is true. The thread will then select one call to one of the method and serve it.

The special behavior of HARPO/L method implementation is to imitate the behavior of hardware, where a thread is a concrete block of devices, which can only serve one request at a time. This fact introduces some special property of HARPO/L method call that should be noticed:

- A thread can not call a method implemented by itself, as this will inevitably cause a deadlock.

- The number of method calls and services will be balanced, unless the system deadlocks, each call will be matched by one execution of an accept statement.
 - As the servicing thread will only accept calls when *Guard* expression is true, failing to meet this requirement may also cause the calling thread to wait indefinitely.
- Method call

$$ObjectId.Name(Args) \mid Name(Args) \quad (2.20)$$

A method call can be a call to method of an object it knows, as in the former form; or a call to method of the same object as in the latter. As discussed in method implementation, a call of the latter form must call a method that is implemented in a different thread of this class.

2.4 Genericity

HARPO/L employs a generic approach to archive polymorphism. A generic parameter (the *GParams* discussed in 2.2.3 and 2.2.4) should be written in following form:

$$(\underline{\text{type}} \textit{Name} \ [\underline{\text{extends}} \ \textit{Type}])^+, \quad (2.21)$$

- When generic argument (a specific type) is passed in, this type will be known inside the generic class or interface body as *Name*.
- The *Type* after keyword **extends** is the bound of this generic parameter, the argument passed in when creating specialized class must extend this *Type*.

The object inheritance rule is:

- Each interface is a subtype of the interface it extends (as in section 2.2.3).
- Each class is a subtype of the interface it extends (as in section 2.2.4).
- Each primitive type has its own inheritance rule as discussed in section 2.2.1.
- Subtyping is transitive and reflexive.

2.5 Example

An example of HARPO/L code is shown as follows:

```
(class FIFO {type T extends primitive}
  constructor(in capacity : int)

  public proc deposit(in value : T)
  public proc fetch(out value : T)

  private obj a : T(capacity)
  private obj front := 0
  private obj size := 0

  (thread
    (wh true
```

```

    (accept
      deposit( in value : T ) when size < capacity
        a[ (front + size) % capacity ] := value
        size := size + 1
      |
      fetch( out value : T ) when size > 0
        value := a[front]
        front := (front + 1) % capacity
        size := size - 1
    accept)
  wh)
thread)
class)

```

This is a HARPO/L class that represents a bounded FIFO queue. It has a generic parameter (type T bounded by *primitive*), and a constructor parameter (variable *capacity* of type *int*). It also has two methods *deposit* and *fetch*, and a thread to implements both of the threads. When executing, the FIFO will loop serving *fetch* and *deposit* calls coming from other classes.

Detailed documentation of language design[8] and semantics [9] will be included in the appendix.

Chapter 3

Colored Petri Net representation of HARPO/L

3.1 Motivation

After defining the language, a behavioral model of the language is also developed. The major reasons for developing such a model are listed as follows [5]:

- The behavioral model is a formal description of the language. In addition to the languages semantics, the model can be used as both a specification to specify the behavior of the system corresponding to the code written in HARPO/L, and a representation to show the meaning of the code. With this model, we can investigate the system to see if it satisfies our goal, and discover design flaws in the language before actually constructing the compiler.
- This behavioral model can be analyzed by either simulation tools or formal analysis methods to see if it satisfies certain properties, so the programmer can

ensure the system under development will work as intended.

- A similar behavioral model can also be developed for the products of the compiler, such as intermediate representation or hardware configuration, to see if models of these products still satisfy properties of the initial model. This process can be used to make sure the meaning of the language remains the same after a number of conversions.
- Constructing such a model will dramatically improve our understanding of the language itself. We can intuitively see the effect of different language components and examine if they behave according to our intention.

Among various types of behavioral models, we chose Colored Petri Nets (CPNs) to be the modeling tool. Colored Petri Nets fit our requirement extremely well because:

- CPNs are graphically represented. This feature makes it very intuitive to understand. They resemble the flow graphs that are commonly used to analyze computer programs.
- CPNs have well defined semantics that describes the behavior of the system without ambiguity. This fact makes it possible to develop simulation and formal analysis tools for it.
- CPNs are state and action oriented at the same time. We can examine both the state of the system, and the actions taken in this system when necessary.
- CPNs are built on concurrency, instead of interleaving. This fact makes it suitable for describing the concurrent behavior of HARPO/L program.

- CPNs offer hierarchical descriptions; we can construct large and complex nets by connecting smaller nets together. This characteristic is very similar to classes and subroutines in high-level programming.

3.2 Introduction to Colored Petri Net

3.2.1 Petri Net

Petri Nets [11] are among the most popular formal mathematical representations for discrete distributed systems. They graphically depict the structure of distributed systems. Petri Nets are also known as place/transition nets because they are formed by a set of places and a set of transitions. An example of a simple Petri Net is shown in Figure 3.1:

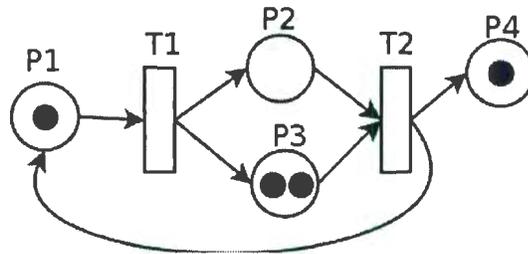


Figure 3.1: A simple petri net.

A Petri Net contains the following information:

- *Places*, represented by circles in diagrams. Places are locations used to hold tokens.
- *Transitions*, represented by rectangles. Transitions are used to indicate the

possible actions in the system.

- *Arcs*, represented by directed arrows. Arcs are used to connect places and transitions, to indicate the flow of tokens in the system. Arcs can only connect a place with a transition or a transition with a place, but not two nodes with a same type.
- *Arc weights*, represented by the number on arcs. Arc weights denote the number of tokens consumed from a place or produced to a place by a transition. Arc weights which are identical to 1 are omitted from the diagrams.
- *Tokens*, represented by solid black dots. Tokens are used to represent the resources or control flows of the system. A distribution of tokens among the places of the system is called a *marking*. The initial distribution of the tokens is called *initial marking*.

A place with a direct arc to a transition is called an *input place* to that transition. If in a marking, all input places of a transition have sufficient tokens (number of tokens required according to arc weights, which will be 1 in all nets in this chapter) in them, this transition is said to be *enabled* in this marking. In Figure 3.1, transition $T1$ is enabled because the input place $P1$ has 1 token in it; but $T2$ is not enabled because $P2$ does not contain tokens to be consumed.

When a transition is enabled, it may *fire* at any time. The effect of a firing of a transition is that tokens are removed from the input places and put into the output places (places that this transition has directed arc pointed to). The number of tokens added/removed is specified by the arc weight.

It should be noticed that the execution of Petri Net is nondeterministic; multiple transitions enabled in a marking can be fired in any order, or not fired at all. This behavior makes Petri Net very suitable for modeling the concurrent behavior of distributed system.

3.2.2 Colored Petri Nets

Colored Petri Nets (CPNs) [6] are extension to ordinary Petri Nets with the addition of *token colors*. In standard Petri Net, tokens are indistinguishable. CPN associates tokens with an attached data value. This value can be a predefined, arbitrarily complex type. Places in CPNs can only contain tokens of the same type.

CPN adds a number features into the ordinary Petri Nets, which includes:

- CPN includes a *declaration* part that contains the declaration of the *color set*. A color declaration contains the name of the color, and the data type that this color is based on. So for example,

$$\text{color } ctrl = \text{int}$$

declares a color named *ctrl* based on integer type, so the color can have value of any integer.

- The places in CPN is a multi-set of its corresponding color. Because the tokens in a place is no longer identical to each other. The use of multi-set instead of set is to permit multiple token of same value being added into the place.
- Instead of arc weight, arcs in CPN can now have arc expressions, which can be a fixed token value, a mathematical expression, or an if-else expression.

Other than these differences, CPN preserves all the characteristics and notations of the ordinary Petri nets.

This characteristic of CPN is exactly the same as the use of types in programming language. CPN combines the strength of ordinary Petri Nets with the strength of high-level programming languages, hence it is very suitable for modeling a language that emphasizes the utilization of concurrent execution of distributed systems, such as HARPO/L.

3.3 Basic elements of the CPN representation of HARPO/L

This section will introduce a number of basic elements of the CPN representation of HARPO/L, which include a token color set, and a set of basic transitions.

3.3.1 Token color sets for the representation

The token colors involved in this representation are shown as follows:

- A set of primitive colors:
 - **color** *int* = **int**
 - **color** *real* = **real**
 - **color** *bool* = **bool**
- Each thread will have a *ctrl* token with a unique value corresponding to thread ID:

– **color** *ctrl* = **int**

- Each method call will also have a specific token type consisting of a *ctrl* token and tokens for all **in** parameters:

– **color** *call* = **record** *ctrl* : *ctrl* * *Name*₁ : *Type*₁ * *Name*₂ : *Type*₂ * ... * *Name*_{*n*} : *Type*_{*n*}

The keyword **record** means that the color declared is an ordered list of {*name*, *color*} pair. The returning token of a method call is also of the same form, with a *ctrl* token and tokens for all **out** parameters.

- Classes do not have a specific token color associated with them, since in HARPO/L, objects can not be assigned to each other.

3.3.2 Basic transition types

The CPN representation consists of several different types of transition, each with its specific usage. A specific program can then be converted to a number of different places and such transitions linked together. Most transitions can be controlled by *ctrl* token, with a *ctrl* token passed in from the output place of last transition and put a *ctrl* token into the input place of the next transition. The arc consuming and producing the *ctrl* token is omitted from the graphs in this section for simplicity. The list of transition types involved in the CPN representation of HARPO/L is shown as follows:

- **Switch transition**

A switch transition takes one token A of any type and an *int* token i as input, and distributes token A to an output according to the value of i .

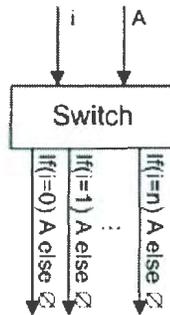


Figure 3.2: Switch transition

Specially, a token of *bool* type can also be used as i . In this case, the switch will have two outgoing arcs, one corresponding to $i = true$ and one corresponding to $i = false$. Moreover, a *ctrl* token can also be used as i , since it is also *int* based.

- **Copy transition**

A copy transition reads in a token A of any type and distributes a copy of that token into all its output places.

- **Merge transition**

A merge transition is the opposite to a copy transition, it takes a token A (which must be with same type and value) from all its input places and produces only one A token on the outgoing arc.

- **Product transition**

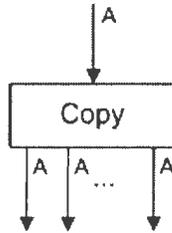


Figure 3.3: Copy transition

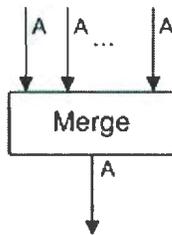


Figure 3.4: Merge transition

A product transition takes a sequence of tokens of any type and combines them to a record containing all the tokens as its fields; for example, group one *ctrl* token and tokens representing the parameters into a *call* token.

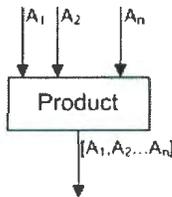


Figure 3.5: Product transition

- Split transition

A split transition is the opposite of a Product transition, it takes a record token as input, separates it, and distribute the parts on different outgoing arcs. For instance, it can be used to break down the returning token of a method call into several corresponding tokens of various types.

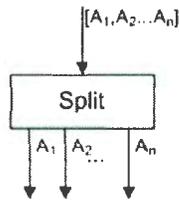


Figure 3.6: Split transition

- **Operator transition**

An operator transition reads in two operands A and B , and produces $C = A \text{ op } B$ on the outgoing arc. It is used to represent operations such as add or multiply. Moreover, this transition may have only one input arc for unary operators.

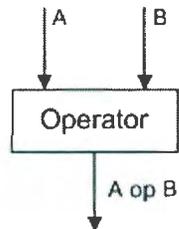


Figure 3.7: Operator transition

- **Read transition**

A read transition is a transition that is used specifically for variable reading. It is the same as a two output arc copy transition, only with a different name for readability.

- **Write transition**

A write transition is slightly different from a merge transition, it has two input arcs, an A_{old} and an A_{new} . The output value of this transition will be the same as A_{new} . Details about variable reading and writing will be discussed in later sections.

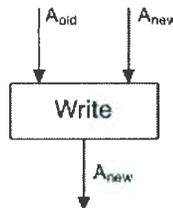


Figure 3.8: Write transition

We can construct a CPN for any HARPO/L program by a combination of these simple transitions.

3.4 Control flow of CPN representation of HARPO/L

The control flow of CPN is determined by the *ctrl* token. When a thread is initialized, a *ctrl* token is created; the *ctrl* will be passed through the statements, and destroyed if and when the thread terminates.

3.4.1 Sequential control flow

The sequential order of statements is ensured by passing a *ctrl* token from one statements to the next, which is illustrated as the following graph:

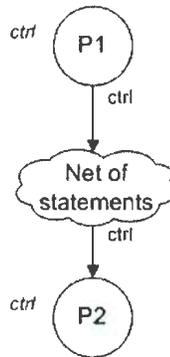


Figure 3.9: Sequential execution of statements

Each thread will have one and only one *ctrl* token, the firing of all transitions within the thread is controlled by this *ctrl* token. This fact makes sure each thread has only one statement under execution at any given time. The transition in figure 3.9 may be of various types, and will consume other tokens in addition to *ctrl*.

3.4.2 Branch control flow

Implementing the branch control flow in HARPO/L, the **if** statement, is quite straightforward. We first evaluate the guard expression and store the result for the expression into a temp variable, and then distribute the incoming *ctrl* token according to the value of the temp variable using Switch transition.

For example, for HARPO/L statement: “(if *a > b* then *X* else *Y* if)”, we have fol-

lowing representation:

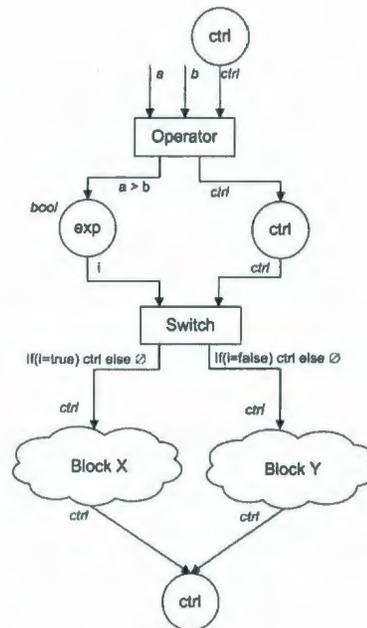


Figure 3.10: Branch control flow

The operator transition is used to evaluate the guard expression into a *bool* token; this token is then used as the input token of switch transition to distribute then *ctrl* token to the corresponding branch. The *ctrl* token is used in operator transition to control the firing of this transition to make sure the expression is only evaluated once before the switch transition is fired.

3.4.3 Loop control flow

A **while** loop can be implemented by an **if** statement following by the loop body block, and feed the *ctrl* token back to the **if** statement after the loop body is finished,

shown as figure 3.11. The single operator transition in this figure can be replaced by an arbitrarily complex combination of transitions to represent a more complex guard expression.

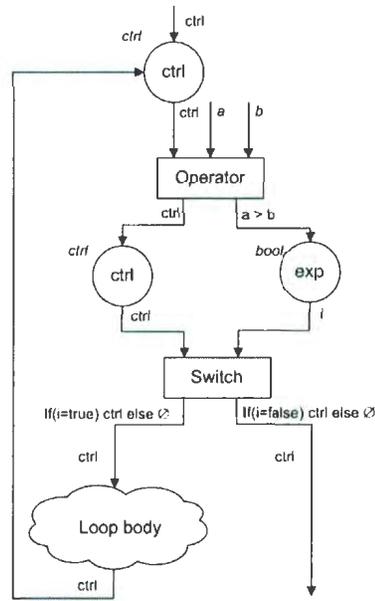


Figure 3.11: Loop control flow

A **for** loop can be rewritten as a **while** loop, and represented in a same way as the **if** loop structure. For example, the statement (**for** *Name:Bounds* *Block* **for**) can be rewritten as:

```

obj Name := 0
obj B := Bounds
(while Name < B
    Block

```

Name := Name + 1

while)

3.4.4 Parallelism

`co` statement can be implemented by producing multiple *ctrl* tokens using a Copy transition, and distributing one token to each of the parallel block. At the end of `co` statement, a Merge transition can be used to merge all finished *ctrl* tokens back to one thread *ctrl* token. This is shown in Figure 3.12.

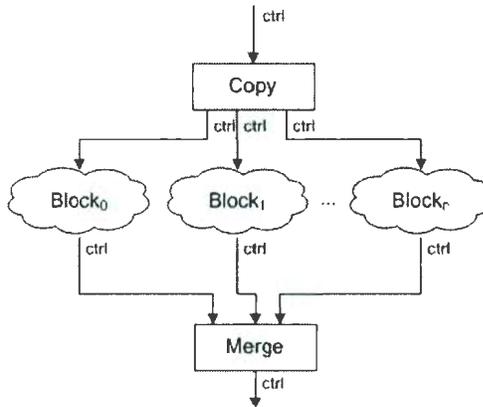


Figure 3.12: Parallel control flow

3.5 Variable operation

A program needs to access variables to perform either reads or writes. Based on the scope and usage, one variable operation can be handled in one of the several following

ways.

3.5.1 Temp variable

Temp variables that are written and read by only one transition respectively can be represented by a single place. For instance, the CPN representation of the expression $a + b + c$ is shown as in figure 3.13. Multiple temp variables of a same type can be

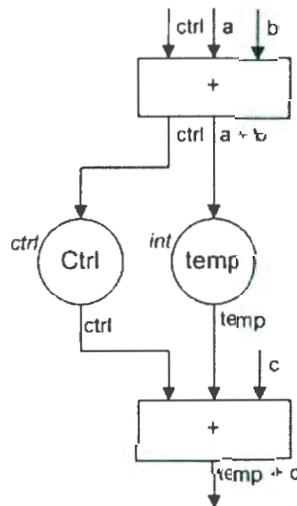


Figure 3.13: Example of using a temp variable in expression $a + b + c$

combined into a single place, providing they will not be accessed at a same time.

3.5.2 Local variable

Local variables are variables that are local to a particular thread. Because each thread will only have one executing statement at any given time, there will not be any simultaneous accesses to the the variable. A local variable can be represented as

a subpage shown as in Figure 3.14.

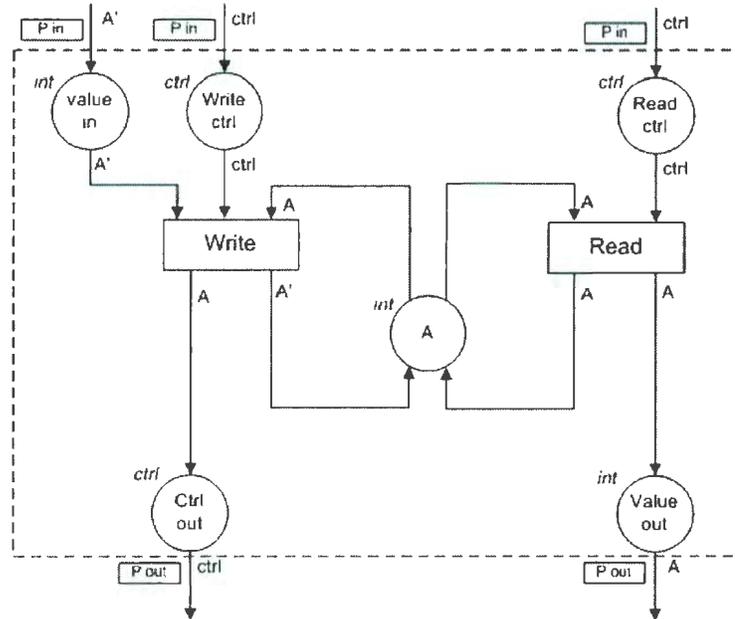


Figure 3.14: Local variable subpage

The local variable subpage has two input ports and two output ports. A read action puts a *ctrl* token in the read-ctrl port, and waits on the value-out port for the value, while a write action puts a *ctrl* token in the write-ctrl port, as well as a *A* token in the value-in port, and waits for the *ctrl* token being returned from ctrl-out port.

An example of read and write action as the statement $C := A + B$ is shown as Figure 3.15.

When an operator transition with local variable as incoming arc is reached, we also insert a copy transition before it, which distributes one *ctrl* token to the ctrl arc

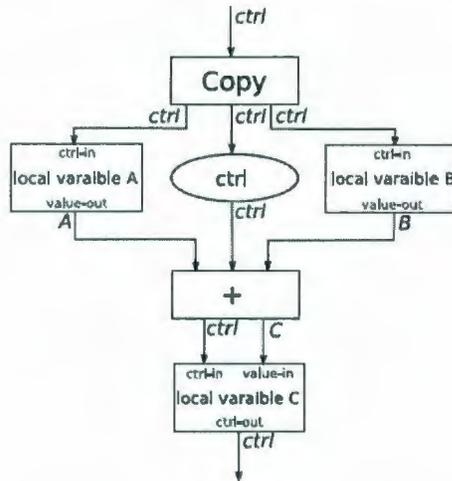


Figure 3.15: An example of local variable read and write action

of the operator transition, and the read-ctrl port of all local variables it needs to read. The operator transition waits on the value-out ports of the local variable, and output the *ctrl* and value tokens to the write-ctrl and value-in ports of the local variable it needs to read. Instead of waiting for *ctrl* token from the operator transition, the next transition following it will wait for *ctrl* token from the ctrl-out port of the local variable subpage.

3.5.3 Shared variable

A shared variable is a variable that may be accessed by more than one thread, for instance, global variables. Representing such a variable is much more complex than local variable, because sequential consistency is required.

For variable reading, it is the same as local variable reading, since the exact order of continuous read sequence does not make much difference. However, for variable

writing, the appropriate ordering is important to make the execution result of the program correct.

The special requirement of shared variable write comes from the requirement of sequential consistency, where the execution of all thread should be the same as if all operations are executed in some sequential order, in which operations from all threads preserve the sequential order of the program.

This requirements requires that when returning a *ctrl* token to a thread, the ctrl-out port needs to return it correctly to the thread which provides the value that has just been written. Otherwise the thread incorrectly received the token will assume the writing has finished while in fact it did not, and break the sequential consistency requirement.

Figure 3.16 shows the representation for the writing of shared variable(variable side):

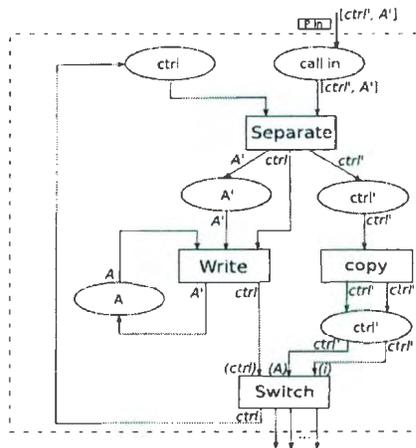


Figure 3.16: Writing of shared variable

Instead of taking the values and *ctrl* tokens separately, the shared variable writing requires them to be united into a record token, so the writing process will not associate the value and *ctrl* tokens from different threads. The writing process also has its local *ctrl* token that provides mutual exclusion between write from different threads. After finishing the writing, the writing process returns the incoming *ctrl* token to its original thread according to its *ctrl* ID.

The local part of the shared variable writing is shown as the following graph:

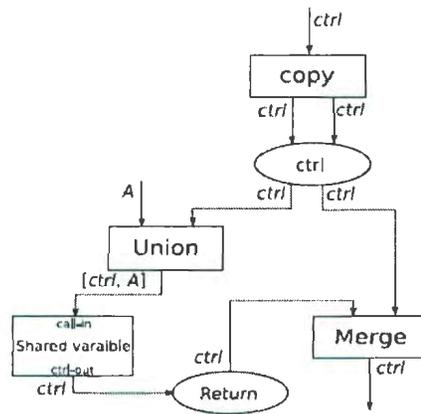


Figure 3.17: Local part of the shared variable writing

Because there may be multiple places in a thread that access a same shared variable, and the *ctrl* token is returned only based on the thread ID of *ctrl* token, the shared variable will not be able to return the *ctrl* to exactly the same line that performs the write. Hence a local copy of *ctrl* in a thread is needed to indicate which part of the thread is performing the writing. The Merge transition having the local *ctrl* token will wait for the shared variable to return the sent *ctrl* token and merge

them to a single token. Because each thread can wait for only one return at any given time, each thread needs only one return place that all returning *ctrl* token will be put into this place.

3.5.4 Array variable

Array variables are represented by a group of variables organized by index, an example of local variable reading is shown as follows:

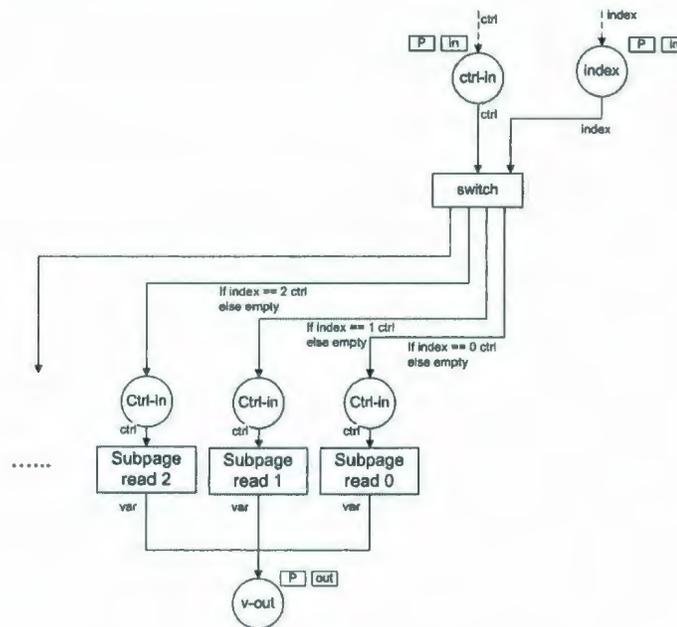


Figure 3.18: Reading of local array

The Switch transition distributes the *ctrl* token based on the index number, the Read transition getting the *ctrl* token will perform the reading, and put the value token into the output port. Because only one Read transition can get the *ctrl* token

at one reading, only one value token will be put into the output port, with the value corresponding to the index value.

The writing procedure will be quite similar to the reading procedure, which will switch the incoming value and *ctrl* token, and collect the returning *ctrl* token at a single output port.

This mechanism works for local array because there will be only one read operation on local variable at any given time, so an local array can be represented as a sequence of local variables grouped together. However, for shared array, we have to ensure that the *ctrl* token and value token are consumed correctly in pair. Moreover, because the access to different items of an array will not conflict with each other, we will only need to ensure the sequential consistency of each item of the array.

3.6 Method calls

Method call can be represented by passing the *ctrl* token between the calling thread and called thread. For example, on called thread side, an accept statement:

```
(accept put(in value:int) guard Block -- get() Block accept)
```

can be represented by Figure 3.19.

When the control flow reaches this accept statement (indicated in diagram by putting a local *ctrl* token into the ctrl-in place), the thread waits for either of the place for method parameter received an input. It will remove the token from the input place and test it against the guard. If the guard is not satisfied, the local *ctrl* token and call parameter will be returned back to their original place; otherwise the call parameter is passed into the call body for execution. After the call is finished,

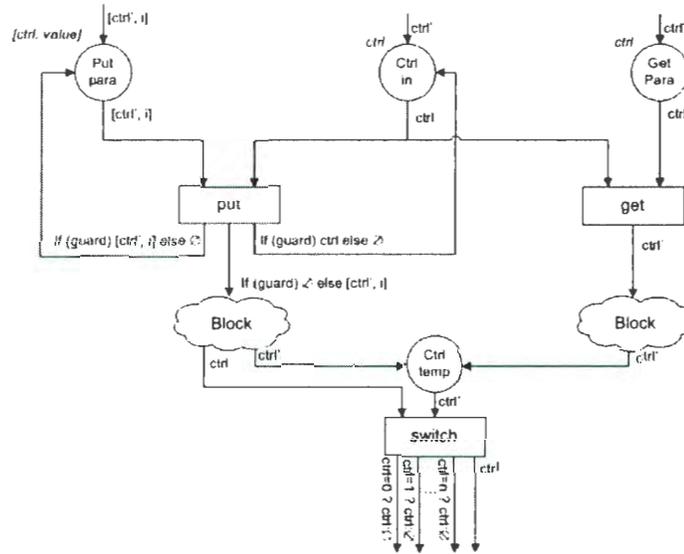


Figure 3.19: Called thread part of method call

the remote *ctrl* token is returned to the calling thread according to the *ctrl* ID, and local *ctrl* token is passed on to the next statement of the thread.

The calling thread part of the method call is in fact quite similar to the local part of shared variable writing (Figure 3.17). The thread will have only one return place for one kind of method call. When the call is performed, the calling thread copies the *ctrl* token, passing one to the called thread, and use the other to wait and merge with the returned *ctrl* token to determine the place where the thread will continue execution.

3.7 Atomic blocks

A special mechanism is needed in the CPN representation to implement atomic blocks. An extra place is added to the graph to hold a token that represents exclusive access.

In CPN representation for HARPO/L code that contains atomic block, in addition to the *ctrl* token, we will add an arc consuming or producing an *atomic* token to both input and output respectively. So any transition will require the *atomic* token to fire, and put the token back after firing. An atomic block, on the other hand, will consume the *atomic* token at the beginning, and only put the token back when the whole block finished execution.

A example for the code block:

```
(thread
  (co
    (atomic
      x
      y
    atomic)
  ||
  m
  n
  co)
thread)
```

can be shown as figure 3.20

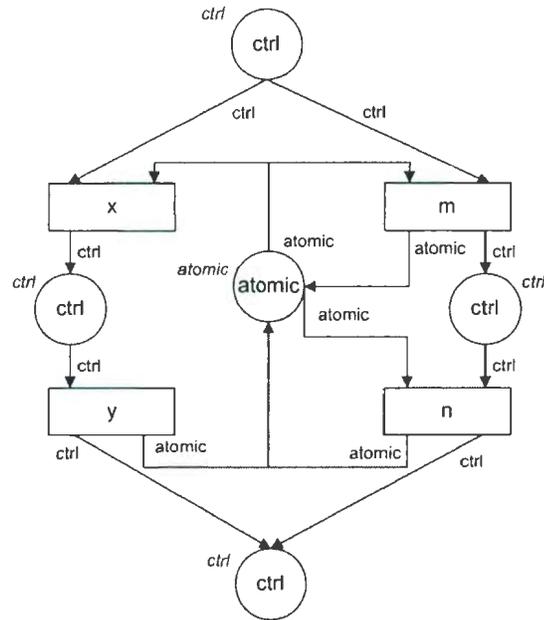


Figure 3.20: CPN representation for atomic block

Initially, the *atomic* place contains one token, so only one atomic section can possess atomic token and eligible for firing at a same time.

This implementation requires atomic sections to be mutually exclusive with each other, which makes the concurrent system to become interleaving-based. We will report on better implementations for atomic block in Chapter 4.

Chapter 4

safety of fission

4.1 Motivation

When compiling a HARPO/L program, we sometimes wish to determine if a block of code may be atomic to other blocks. The atomicity property is useful in code optimization. If this property is true, we can perform sequential code optimization techniques that otherwise would not be valid to the code block. Moreover, when implementing the **atomic** keyword, atomicity check may greatly improve the efficiency of configuration generated.

Being atomic means a certain block of code can only be either not executed at all, or fully executed in other threads' point of view. We use a $\langle \rangle$ pair or **(atomic)** block to indicate that the enclosed code block is atomic. For example:

(thread

obj a := 0

```

obj b := 0
(co
  (atomic
    a := 5
    a := 10
  atomic)
  ||
  b := a
  b := 15
co)
thread)

```

Because the two assignments to a is enclosed in an **atomic** structure, according to the definition of atomicity, after the thread has finished execution, the value of variable b can either be 0 (statement $b := a$ executed before the atomic block) or 10 ($b := a$ executed after the atomic block), but never 5 ($b := a$ executed after $a := 5$ but before $a := 10$).

While true atomicity can not be achieved without hardware support, in implementation, we can make the block appear to be atomic by the use of semaphores. Pseudo-code for the implementation of the above example using semaphore is shown as follows:

```

obj a := 0
obj b := 0

```

```
semaphore s := 1
```

```
thread 0 {
```

```
    P(s)
```

```
    a := 5
```

```
    a := 10
```

```
    V(s)
```

```
}
```

```
thread 1 {
```

```
    P(s)
```

```
    b := a
```

```
    V(s)
```

```
    P(s)
```

```
    b := 15
```

```
    V(s)
```

```
}
```

This method requires us to enclose every single statement in thread 1 by a pair of P() and V() operation. For longer code blocks, this method can be very costly as it adds two extra semaphore operations to every statement of thread 1. Alternatively, we can enclose the thread 1 in a single P() and V() pair, but using this method means the concurrency of the `co` structure is completely lost.

This method is based on the assumption that all statements in atomic block must be executed together without interleaving with any statement from other threads. However, this is usually not true. Shown as follows:

$$F = (a := 5; b := c;)$$
$$F' = (\mathbf{atomic} \ a := 5; b := c; \mathbf{atomic})$$
$$G = (b := 10; a := c;)$$

A careful examination of the code blocks shows that any interleaving of block F' and G will produce a result that is same as some interleaving of F and G . The implementation may satisfy the atomicity requirement even if no special mechanism is employed to ensure it. If we can identify such situation in more complex code blocks, we can partition the atomic blocks into smaller atomic blocks that can be implemented without complex synchronization method such as semaphore. We can continue this process until we identify the minimum blocks of statements that will interfere with the atomicity property, then we can only implement code flow control for only these blocks, and thus achieve a much more efficient implementation.

In this section, we will develop a method to determine if an atomic block code can be safely partitioned into separate parts and to identify the portion of code that makes the partition not valid if this operation can not be performed safely.

4.2 Introduction to trace theory

In order to describe the behavior of different interleaving of multiple thread blocks, we employ trace theory [13] as the mathematical tool.

We can view a block of code as a sequence of different operations. Because we are only interested in how different threads access shared variables, we divide different

operations into three categories: operations that read from a shared variable, operations that write to a shared variable and operations that do not perform variable access.

We use identifier W to represent a write operation and R to represent a read; moreover, a superscript is used to indicate the name of the shared variable and a subscript is used to specify the thread this operation belongs to. Hence, R_T^a is a read for shared variable a from thread T ; W_U^b is a write to variable b from thread U ; \emptyset_T is an operation in T that does not perform shared variable access.

Each of these identifiers of an operation is called a *symbol*. An *alphabet* is then a finite set of symbols. Because of the nature of threads, we define that an alphabet representing a thread T should contain access to all shared variables from this thread plus the symbol for no shared variable access \emptyset_T . An alphabet of a thread T is denoted by $\underline{a}T$.

A *trace* is a finite-length sequence of symbols from some alphabet. In our case, a trace is the sequence of operations corresponding to a possible execution of a thread. A *trace set* is a set of traces. The trace set of thread T is denoted by $\underline{t}T$. The thread T is then denoted by a *trace structure*, which is the pair $(\underline{a}T, \underline{t}T)$. We will write such a trace structure as T for simplicity for the remaining part of the thesis.

4.2.1 Projection

We define the *projection* of a trace t to an alphabet A to be the result of removing all symbols in this trace that does not belong to the alphabet, and keep the rest of the symbols in the original order.

A projection operation can be written as:

$$t \uparrow A$$

where t is a trace and A is an alphabet.

A simple example of projection is shown as follows:

$$\begin{aligned} t &= aXbYcdZe \\ A &= \{a, b, c, d, e\} \\ t \uparrow A &= abcde \end{aligned} \tag{4.1}$$

In the remaining part of the thesis, upper case identifiers such as T and U will be used to represent threads or trace structures, and lower case identifier such as t and u will be used to represent traces, unless otherwise specified.

4.2.2 Weaving

With the definition of projection, we can formally define the *weave* function to be [13]:

$$T\underline{w}U = (\{x \in (\underline{a}T \cup \underline{a}U)^* \mid x \uparrow \underline{a}T \in \underline{t}T \wedge x \uparrow \underline{a}U \in \underline{t}U\}, \underline{a}T \cup \underline{a}U) \tag{4.2}$$

The definition means that the result of weaving two trace structure T and U is a trace structure such that:

- The alphabet of this resulted trace structure is the union of the alphabet of the two trace structures U and T .
- A trace in the trace set of the resulted trace structure must be a sequence of the symbols from the new alphabet.

- All traces in the resulted set project to the alphabet of T ($\underline{a}T$) must be in the trace set of T ($\underline{t}T$).
- All traces in the resulted set project to the alphabet of U must be in the trace set of T .

For our case, because the symbol sets of trace structures representing two different threads are always disjoint (which is intuitive because the symbol set of a thread only contains operations from this thread itself), the trace set of the weaving of two trace structure $\underline{t}(T\underline{w}U)$ is in fact all possible interleaving of all pairs of traces $\langle t, u \rangle$ where t is from $\underline{t}T$ and u is from $\underline{t}U$.

A simple example of weaving is shown as follows:

$$\begin{aligned}
\underline{t}T &= \{W_T^b R_T^a\} & \underline{a}T &= \{R_T^a, W_T^a, R_T^b, W_T^b, \emptyset_T\} \\
\underline{t}U &= \{W_U^a R_U^b\} & \underline{a}U &= \{R_U^a, W_U^a, R_U^b, W_U^b, \emptyset_U\} \\
\underline{t}(T\underline{w}U) &= \{W_T^b R_T^a W_U^a R_U^b, W_T^b W_U^a R_T^a R_U^b, W_T^b W_U^a R_U^b R_T^a, \\
&\quad W_U^a W_T^b R_T^a R_U^b, W_U^a W_T^b R_U^b R_T^a, W_U^a R_U^b W_T^b R_T^a\} \\
\underline{a}(T\underline{w}U) &= \{R_T^a, W_T^a, R_T^b, W_T^b, \emptyset_T, R_U^a, W_U^a, R_U^b, W_U^b, \emptyset_U\} \quad (4.3)
\end{aligned}$$

In this example, two threads T and U accesses two shared variables a and b . The trace sets of the two trace structures both contain only one trace, each with two symbols. The result trace set $\underline{t}(T\underline{w}U)$ contains six traces. This trace set may seems complicated at the first glance; however, a careful inspection of the subscripts will reveal that it is just interleaving the symbols from the two traces in an arbitrary order. For example, a trace with subscript in order $TUUT$ ($W_T^b W_U^a R_U^b R_T^a$) means first execute one operation in thread T , followed by the two operation of U , then

execute the other operation of T .

As the number of traces in each trace set and the number of symbols in each trace grows, the size of result trace set will grow exponentially. But the rule of weaving remains the same. Deriving the result of weaving two trace structures representing two threads together will be a trivial task given enough time and patience.

4.2.3 Equivalence of traces

In this section, we will define *equivalent* relationship between traces. In common sense, two traces are equivalent only if they are equal, which means they are formed by the same set of symbols with the same order. However, because we are only interested in using traces to represent the execution of threads, we will say traces that produce the same execution result are equivalent (represented by \equiv) to each other. For example:

$$W_T^a R_T^a R_U^a W_U^a \equiv W_T^a R_U^a R_T^a W_U^a \quad (4.4)$$

because although the order of the two reads, R_T^a and R_U^a , are different in the two traces, they all return the same value (the value written by W_T^a). Hence the program execution result can not reflect this difference, the two execution orders are the same in the view point of the user of the program.

As a result, we define the equivalence relationship between two traces by definition-use chain.

4.2.4 Definition-use chain

We describe the relationship between a read R and a write W as a definition-use relationship, so that a write W^a is a definition to the variable a , and a read R^a is a use of the variable. We say a definition *reaches* a use of a if this read R^a returns the value written by the definition W^a .

Intuitively, a definition can reach a use only if it appears before the use, and there is no other definition between the definition and use that overrides the value. We define $D_t(R^a)$ to be the definition affecting this particular R^a in trace t .

Because we are only interested in the interleaving between operations from two threads, all resulted trace sets we are considering will contain the same set of operations and all operations from same thread will always follow a same sequential order. As a result, we can label the read and write operations by the order they appear in the respective trace. For instance, $R0_T^a$ is the first read to variable a in thread T , and Wn_U^a is the $n + 1$ th write operation to a in U .

We define two traces t and u from $T \underline{w} U$ to be equivalent if

$$t \equiv u \quad \Leftrightarrow \quad \forall R_i \in t \cdot D_t(R_i) = D_u(R_i) \quad (4.5)$$

This equation means a trace t is equivalent to a trace u only if all uses in t are determined by the same definition as they are in thread u .

4.2.5 Equivalence relationship between two symbol traces

Determining if two traces are equivalent can be achieved by performing a definition-use check for all uses of the traces. As the length of the source trace increases, the

trace set of weaving result grows dramatically. In order to efficiently analyze the equivalence relationship between large number of traces, we need to develop methods that are more efficient than simply performing definition-use check to all uses.

We can break down the equivalence checks between two trace sets into a series of equivalence checks of two symbol trace, Detailed discussion about this method will be provided in later sections. In this section, we develop a rule for the equivalence relationship between all two symbol traces.

1. $R_x^a R_y^a \equiv R_y^a R_x^a$, as we discussed before, read does not have a particular order, since a read will neither establish nor block a definition-use chain.
2. $W_x^a R_y^a \neq R_y^a W_x^a$, because the write will reach the read in left side, and will not reach the read in the right side, these two traces are not equivalent.
3. $W_x^a W_y^a \equiv W_y^a W_x^a$ if there is no read operation to a immediately following the trace. If there is a read operation R_z^a following these two symbols, then on the left side, $D(R_z^a) = W_y^a$, and on the right side, $D(R_z^a) = W_x^a$, the two traces are not equivalent. On the other hand, if there is no read or another write to a that follows these two symbols, the interchange would not affect the equivalence of the two traces.
4. $O_x^a O_y^b \equiv O_y^b O_x^a$, where O can be either read or write. Operations to different variables would not interfere with each other, since no definition-use chain is established between operations on different variables.
5. $\emptyset O^a \equiv O^a \emptyset$. Intuitively, operations without shared variable access would not affect the definition-use chain of shared variables.

4.2.6 Equivalence of trace sets

We define the refinement relationship between trace sets to be:

$$\underline{t}T \sqsubseteq \underline{t}U \Leftrightarrow \forall u \in \underline{t}U \cdot \exists t \in \underline{t}T \cdot t \equiv u \quad (4.6)$$

We say a trace set $\underline{t}U$ *refines* another trace set $\underline{t}T$ if for any trace u in $\underline{t}U$, we can find a trace t in $\underline{t}T$ so that u is equivalent to t . This statement also implies an simplicity restriction that

$$\underline{a}T \subseteq \underline{a}U \quad (4.7)$$

because a trace t can not be equivalent to other traces that has symbols that do not exist in t . For the rest part of the paper, we will not consider this restriction because the trace structures we are examining will always have the same alphabet.

An example of refinement relationship of trace structures will be shown as follows:

$$\begin{aligned} \underline{t}T &= \{W^bW^aR^bR^a, W^aW^bR^bR^a\} \\ \underline{t}U &= \{W^bW^aR^aR^b, W^aR^bR^aW^b\} \\ T &\sqsubseteq U \end{aligned} \quad (4.8)$$

Trace structure T is refined by U because, according to the definition of equivalence of traces, $W^bW^aR^bR^a \equiv W^aW^bR^bR^a \equiv W^bW^aR^aR^b$ since both uses in these three traces are affected by the same definition. According to 4.6, for both of the traces in $\underline{t}T$, there exists a trace $W^bW^aR^aR^b$ in $\underline{t}U$ that is equivalent to them.

However, we should note that U is not refined by T , because there is a trace $W^aR^bR^aW^b$ in $\underline{t}U$ that is not equivalent to either of the two traces in $\underline{t}T$. If this trace is removed from $\underline{t}U$, then the new trace structure U' will be refined by T . The two

trace structures will be *equivalent* to each other.

$$T \equiv U' \Leftrightarrow (\underline{t}T \sqsubseteq \underline{t}U') \wedge (\underline{t}U' \sqsubseteq \underline{t}T) \quad (4.9)$$

4.3 safety of fission

4.3.1 Fission and fusion

We introduce two transformations, **fusion** and **fission** [15], to help deal with the problem introduced in Section 4.1.

A transformation in concurrent program being **safe** means this transformation can not introduce new behaviors that will not occur in the original program. Suppose we use trace structures to represent all possible execution paths of a concurrent program, so that T represents the original program and T' represents the program after transformation; a transformation is *safe* if:

$$T \underline{w} U \sqsubseteq T' \underline{w} U \quad (4.10)$$

where U is an arbitrary trace structure. This equation means a transformation is safe if, when both trace structures before and after the transformation are weaved with another trace structure, the resulting trace structure after the transformation should always be refined by the resulting trace structure before the transformation.

Note that the safety of a transformation does not require the trace structure after the transformation to be equal to the one before the transformation. The transformation can eliminate some possible traces, but should not add new ones. Eliminating traces is safe because being safe means the trace set after transformation should not contain any trace that is not equivalent to some existing traces in the original result

trace set, and removing traces from the original trace set clearly would not affect this property.

A fusion is a transformation that converts two atomic blocks into a single atomic block:

$$\langle p \rangle; \langle q \rangle \quad \longrightarrow \quad \langle p; q \rangle \quad (4.11)$$

Fusion reduces the nondeterminism in parallel program. That means, if we describe the threads of parallel program by trace structures, and the concurrent execution of them by weaving, fusion will reduce the size of the trace set of the resulted trace structure, illustrated by the following example:

$$\begin{aligned} \underline{t}T &= \{\langle W_T^b \rangle \langle R_T^a \rangle\} \\ \underline{t}T' &= \{\langle W_{T'}^b, R_{T'}^a \rangle\} \\ \underline{t}U &= \{W_U^a R_U^b\} \\ \underline{t}(T \underline{w} U) &= \{\langle W_T^b \rangle \langle R_T^a \rangle W_U^a R_U^b, \langle W_T^b \rangle W_U^a \langle R_T^a \rangle R_U^b, \langle W_T^b \rangle W_U^a R_U^b \langle R_T^a \rangle, \\ &\quad W_U^a \langle W_T^b \rangle \langle R_T^a \rangle R_U^b, W_U^a \langle W_T^b \rangle R_U^b \langle R_T^a \rangle, W_U^a R_U^b \langle W_T^b \rangle \langle R_T^a \rangle\} \\ \underline{t}(T' \underline{w} U) &= \{\langle W_{T'}^b, R_{T'}^a \rangle W_U^a R_U^b, W_U^a \langle W_{T'}^b, R_{T'}^a \rangle R_U^b, W_U^a R_U^b \langle W_{T'}^b, R_{T'}^a \rangle\} \\ \underline{t}(T \underline{w} U) &\subseteq \underline{t}(T' \underline{w} U) \end{aligned} \quad (4.12)$$

We can see that fusion will only reduce the number of possible interleaving traces in concurrent execution but not introduce new traces. So fusion is always safe.

In the rest of the thesis, we will omit the atomic sign “ $\langle \rangle$ ” from atomic block that has only one symbol, since a single symbol is always atomic.

A fission is the opposite of fusion, which splits one atomic block into two separate

blocks:

$$\langle p; q \rangle \longrightarrow \langle p \rangle; \langle q \rangle \quad (4.13)$$

Fission will increase the nondeterminism in the concurrent program by increasing number of possible interleaving traces in the resulting trace structure.

Strictly speaking, a fission is never safe as long as both p and q contain at least one shared variable access. As for any fission transformation, we can always find a trace set $\underline{t}U$ that when weaves with the original trace set and the one after transformation, will cause:

$$T \underline{w} U \not\sqsubseteq T' \underline{w} U$$

However, we are interested in a special case of safety, that is, if the fission is safe in a certain context. For instance, in the case of Example 4.12, if U represents all threads that are executing in parallel with T , then a fission

$$\underline{t}T = \{\langle W_T^b R_T^a \rangle\} \longrightarrow \underline{t}T' = \{W_{T'}^b R_{T'}^a\}$$

is safe because the three new traces introduced by fission:

$$W_{T'}^b W_U^a R_{T'}^a R_U^b, W_{T'}^b W_U^a R_U^b R_{T'}^a, W_U^a W_{T'}^b R_U^b R_{T'}^a$$

are all equivalent to the trace

$$W_U^a \langle W_T^b R_T^a \rangle R_U^b$$

in the original trace set.

If we modify the thread U by just changing the order of the two symbols in its trace, so that U becomes:

$$\underline{t}U = \{R_U^b W_U^a\}$$

The fission

$$\langle p; q \rangle \longrightarrow \langle p \rangle; \langle q \rangle$$

is not safe anymore under this new context. Illustrated as follows:

$$\begin{aligned}
\underline{t}T &= \{\langle W_T^b R_T^a \rangle\} \\
\underline{t}T' &= \{W_T^b, R_T^a\} \\
\underline{t}U &= \{R_U^b W_U^a\} \\
\underline{t}(T \underline{w} U) &= \{\langle W_T^b R_T^a \rangle R_U^b W_U^a, R_U^b \langle W_T^b R_T^a \rangle W_U^a, R_U^b W_U^a \langle W_T^b R_T^a \rangle\} \\
\underline{t}(T' \underline{w} U) &= \{W_T^b, R_T^a, R_U^b W_U^a, R_U^b W_T^b, R_T^a W_U^a, R_U^b W_U^a W_T^b, R_T^a \\
&\quad W_T^b, R_U^b W_U^a R_T^a, W_T^b, R_U^b R_T^a, W_U^a, R_U^b W_T^b, W_U^a R_T^a\} \\
\underline{t}(T \underline{w} U) &\not\subseteq \underline{t}(T' \underline{w} U)
\end{aligned} \tag{4.14}$$

This fission is not safe because there is a trace

$$W_T^b, R_U^b W_U^a R_T^a$$

in the new result trace set that is not equivalent to any of the three traces in the original result trace set.

As a result, we will only consider the safety of fission with respect to the context of the concurrent program. We use M as the context, which is the trace structure representing all threads that are executing in parallel with the target thread undergoing fission transformation.

We introduce a new operator:

$$T \sqsubseteq_M U \Leftrightarrow T \underline{w} M \sqsubseteq U \underline{w} M \tag{4.15}$$

to mean the transformation from trace structure T to U is safe under context M .

So instead of Equation 4.10, we use

$$\{\langle p; q \rangle\} \sqsubseteq_M \{\langle p \rangle; \langle q \rangle\} \Leftrightarrow (\{\langle p; q \rangle\} \underline{w} M) \sqsubseteq (\{\langle p \rangle; \langle q \rangle\} \underline{w} M) \quad (4.16)$$

to represent the fission is safe under context M .

4.3.2 safety

The context M is the trace structure representing all other threads executed in parallel with the thread under fission transformation, hence it may contain many trace structures representing different threads. However, this context can be split to a number of different subsets. So if

$$M = M_0 \underline{w} M_1 \underline{w} \dots \underline{w} M_n$$

then

$$T \sqsubseteq_M U \Leftrightarrow (T \sqsubseteq_{M_0} U) \wedge (T \sqsubseteq_{M_1} U) \wedge \dots \wedge (T \sqsubseteq_{M_n} U) \quad (4.17)$$

This property will be proved in later section, in this section, we will only consider M to contain trace set for a single thread, as traces from multiple threads only require us to repeat the equivalence analysis multiple times.

Although trace set representing a single thread may also contain multiple traces because of branch and loop structure, we can develop methods to merge traces introduced by these structures into a single trace (detail discussed in later sections). As a result, we will only deal with the case that $\underline{t}M$ contain only one trace: m .

Because a trace m is just a sequence of different operations, we can label the operations by a symbol m_i corresponding to the order of presence in m . So the trace

m is a symbol sequence $m_0m_1m_2 \cdots m_{n-1}$. According to our definition, $\{\langle p; q \rangle\} \underline{w}\{m\}$ is the set of all possible interleaving of the two traces:

$$\underline{t}(\{\langle p; q \rangle\} \underline{w}\{m\}) = \{pqm_0m_1 \cdots m_{n-1}, m_0pqm_1 \cdots m_{n-1}, \cdots, m_0m_1 \cdots m_{n-1}pq\}$$

This is a set of $n+1$ traces with pq being placed before m_0 to m_{n-1} , or after m_{n-1} . Because $\langle p; q \rangle$ is atomic, no symbol can be placed in between of p and q . On the other hand, the trace of $\{\langle p \rangle; \langle q \rangle\} \underline{w}\{m\}$ contains not only all traces in $\{\langle p; q \rangle\} \underline{w}\{m\}$, but also traces that have a subset of m placed between p and q . There are $\frac{n^2}{2}$ such traces.

We can label each trace by the number of operations before p and the number of operations after q . Define the result trace set $\{\langle p \rangle; \langle q \rangle\} \underline{w}\{m\}$ to be I , where I_{xy} is the trace that has x operations before p and y operations after q , namely the trace

$$m_0m_1 \cdots m_{x-1}pm_x \cdots m_{n-y}qm_{n-y+1} \cdots m_{n-1} \quad (4.18)$$

Intuitively, $\{\langle p; q \rangle\} \underline{w}\{m\}$ only contains I_{xy} that does not have operations between p and q , hence $x+y = n$, while $\{\langle p \rangle; \langle q \rangle\} \underline{w}\{m\}$ contains all possible interleaving. So:

$$I_{xy} \in \{\langle p; q \rangle\} \underline{w}\{m\} \quad \Leftrightarrow \quad x+y = n \quad (4.19)$$

$$I_{xy} \in \{\langle p \rangle; \langle q \rangle\} \underline{w}\{m\} \quad \Leftrightarrow \quad x+y \leq n \quad (4.20)$$

Hence the proof of equivalence is to prove the two sets are equivalent, so for every trace W_{xy} that $x+y < n$, there is a trace with $x+y = n$ that is equivalent to it.

$$\{\langle p; q \rangle\} \supseteq_M \{\langle p \rangle; \langle q \rangle\} \quad \Leftrightarrow \quad \forall (I_{ij} | i+j < n) \cdot \exists (I_{xy} | x+y = n) \cdot I_{ij} \equiv I_{xy} \quad (4.21)$$

4.3.3 Proof of equivalence of traces

Given two traces from the interleaving set I_{xy} and I_{ij} , we can prove whether $I_{xy} \equiv I_{ij}$ by the equivalence rule of two symbol traces discussed in Section 4.2.5. In this section,

we only consider the case that p , q and m contains access to only one shared variable for simplicity. The method for dealing with multiple variables will be discussed in later sections.

From the definition, I_{xy} is the trace

$$m_0 \cdots m_{x-1} p m_x \cdots m_{n-y} q m_{n-y+1} \cdots m_{n-1} \quad (4.22)$$

Each m_i is either a R , W or \emptyset symbol. Because p and q may be multiple operations grouped together, they may contain both reads and writes. However, if there is a write operation before the first read operation in p or q , the use of the shared variable in them will always be determined by this write. Hence, read operations in this case will always satisfy the equivalent requirement in any interleaving.

$$WR \equiv W \quad \text{in } \langle p \rangle \text{ or } \langle q \rangle \quad (4.23)$$

If this is not the case, there is a read before the first write to the shared variable, then p or q must be treated as both read and write, and have the constraint of both operations. Hence, p and q are each either R , W , \emptyset or RW .

Suppose $x > i$ in I_{xy} and I_{ij} ; to prove $I_{xy} \equiv I_{ij}$, we can first prove $I_{x-1,y} \equiv I_{xy}$. We can see that $I_{x-1,y}$ is the trace

$$m_0 \cdots m_{x-2} p m_{x-1} \cdots m_{n-y} q m_{n-y+1} \cdots m_{n-1} \quad (4.24)$$

Compare this trace with the trace 4.22, we can see that

$$I_{x-1,y} \equiv I_{xy} \quad \Leftrightarrow \quad m_{x-1} p \equiv p m_{x-1} \quad (4.25)$$

We can judge whether this equivalent relationship is valid or not by the rule of the equivalence of two symbol traces developed in Section 4.2.5.

Following this method, we can further judge whether $I_{x-2,y} \equiv I_{x-1,y}$ by examining if $m_{x-2}p \equiv pm_{x-2}$. If this is also true, we can say that $I_{x-2,y} \equiv I_{x-1,y} \equiv I_{xy}$. Repeating k times until $x - k = i$, we can prove $I_{xy} \equiv I_{iy}$; moreover, the equivalence chain also contains any $I_{a,y}$ where $x \leq a \leq i$. Performing the same operation to q , we can establish an equivalence chain that shows $W_{xy} \equiv W_{iy} \equiv W_{ij}$.

Failing to establish this equivalence chain will not always mean the two traces are not equivalent. Because a read-write pair can be moved together even if they can not be moved individually. For example $R_U W_T R_T W_U \equiv R_U W_U W_T R_T$ even though $R_U W_T R_T W_U \neq R_U W_T W_U R_T \neq R_U W_U W_T R_T$.

However, this also means that there will be at least one trace in the chain that is not equivalent to the target trace. Because we are interested in the resulting trace set as a whole instead of the starting trace itself, this problem will not affect the result of our analysis.

4.3.4 Equivalence of result sets

As discussed earlier, if we want to prove

$$\{\langle p; q \rangle\} \equiv_{\{m\}} \{\langle p \rangle; \langle q \rangle\}$$

we have to prove

$$\forall(I_{ij} | i + j < n) \cdot \exists(I_{xy} | x + y = n) \cdot I_{ij} \equiv I_{xy}$$

Given an arbitrary trace I_{ij} with $i + j < n$, we need to find a trace I_{xy} with $x + y = n$ that is equivalent to it. From the analysis in trace equivalence, we can see the way to prove this is to increment i and j until the sum of i and j is equal to n . This increment is achieved by repeatedly performing the stepwise equivalence check.

We can see that the equivalence check is independent of each other and will not be affected by the order they are performed. If we can prove $I_{ij} \equiv I_{xy}$, we can prove that any trace W_{ab} with $i \leq a \leq x$ and $j \leq b \leq y$ is also equivalent to W_{ij} by changing the order of equivalence check.

Generally, if we can prove I_{00} is equivalent to some I_{xy} with $x + y = n$, then every trace I_{ab} with $0 \leq a \leq x$ and $0 \leq b \leq y$ will also be equivalent to this I_{xy} .

This is achieved by first performing equivalence checks to the front of the trace to increment i , until we meet some check that is not valid thus can not increase i anymore. After that, we switch to performing equivalence check to the end of the trace to increment j , until $i + j = n$ or another nonequivalent increment is encountered.

$$\forall(I_{ab}|a \leq i; b \leq j) \cdot I_{ab} \equiv I_{ij} \quad (4.26)$$

where

$$I_{ij} \in \{\langle p; q \rangle\} \underline{w}\{m\}.$$

For the case that $a > i$, following the same set of equivalence checks, we can easily prove that $I_{ab} \equiv I_{a,n-a}$ by performing the appropriate equivalence check to back of trace only, Obviously this $I_{a,n-a}$ also belongs to $\{\langle p; q \rangle\} \underline{w}\{m\}$. Same holds for the situation $b > j$. By summing up all these three cases, we have proved that

$$\forall(I_{ij}|i + j < n) \cdot \exists(I_{xy}|x + y = n) \cdot I_{ij} \equiv I_{xy} \quad (4.27)$$

and hence

$$\{\langle p; q \rangle\} \equiv_M \{\langle p \rangle; \langle q \rangle\} \quad (4.28)$$

4.3.5 Proof of decomposability of the context

In this section, we will prove the theorem mentioned in section 4.3.2, that the context can be considered one thread at a time.

Suppose we have two traces:

$$T = \langle p; q \rangle$$

$$T' = \langle p \rangle; \langle q \rangle$$

and context:

$$M = M_0 \underline{w} M_1 \underline{w} M_2 \underline{w} \dots \underline{w} M_n$$

then

$$T \sqsubseteq_M T' \Leftrightarrow (T \sqsubseteq_{M_0} T') \wedge (T \sqsubseteq_{M_1} T') \wedge \dots \wedge (T \sqsubseteq_{M_n} T') \quad (4.29)$$

Since we have

$$T \sqsubseteq_M T' \Leftrightarrow T \underline{w} (M_0 \underline{w} M_1 \dots \underline{w} M_n) \sqsubseteq_M T' \underline{w} (M_0 \underline{w} M_1 \dots \underline{w} M_n) \quad (4.30)$$

Because weaving is associative, the right hand side of above equation is equivalent to

$$(((T \underline{w} M_0) \underline{w} M_1) \dots \underline{w} M_n) \sqsubseteq_M (((T' \underline{w} M_0) \underline{w} M_1) \dots \underline{w} M_n) \quad (4.31)$$

If

$$T \sqsubseteq_{M_0} T'$$

which means

$$\langle p; q \rangle \underline{w} M_0 \sqsubseteq \langle p \rangle; \langle q \rangle \underline{w} M_0 \quad (4.32)$$

from our analysis in previous sections, Equation 4.32 is equivalent to:

$$\forall t' \in \langle p \rangle ; \langle q \rangle \underline{w} M_0 \cdot \exists t \in \langle p; q \rangle \underline{w} M_0 \cdot t \equiv t'$$

So every trace in

$$\langle p \rangle ; \langle q \rangle \underline{w} M_0$$

can be transformed into a form:

$$m_0 m_1 \cdots m_{x-1} \langle pq \rangle m_x m_{x+1} \cdots m_{n-1}$$

According to section 4.2.5, symbols that come before p will not affect the safety of fission of this trace and any other trace, since the equivalence check is not affected by symbols before the two exchanged symbols. Similarly, although we do take the symbols that come after q into consideration in the moving write across write case; having a read of a variable immediate following q basically means q does not have write access to that variable, otherwise the read will not be able to move across q .

So the resulted trace set of weaving

$$(T' \underline{w} M_0) \underline{w} M_1 \sqsubseteq (m_0 m_1 \cdots m_{x-1} \langle pq \rangle m_x m_{x+1} \cdots m_{n-1}) \underline{w} M_1$$

where the right side of equation is the equivalent to

$$(T \underline{w} M_0) \underline{w} M_1$$

Continue this process for all M_n , we can then prove that

$$T \underline{w} (M_0 \underline{w} M_1 \dots \underline{w} M_n) \sqsubseteq T' \underline{w} (M_0 \underline{w} M_1 \dots \underline{w} M_n)$$

if

$$(T \sqsubseteq_{M_0} T') \wedge (T \sqsubseteq_{M_1} T') \wedge \dots \wedge (T \sqsubseteq_{M_n} T') \quad (4.33)$$

The context of fission is associative.

4.4 Applying the method

When applying this result to general computer programs, we need to consider a number of more complex situations. Solution for these problems is discussed in this section.

4.4.1 Multiple variables

The discussion in earlier sections is based on single shared variables for simplicity. In real application, this is usually not the case; however, our method can be extended to deal with this problem without too much modification.

Although $\langle p \rangle$ and $\langle q \rangle$ may access multiple variables, for a single variable, it can still be only R , W , \emptyset or RW . Hence, we can derive a reading variable set \mathbf{R} , writing set \mathbf{W} , and read/write set \mathbf{RW} for $\langle p \rangle$ and $\langle q \rangle$ respectively. Moreover, each operation m_i in m also has such a set of access variables. When performing the equivalence check described above, instead of checking equivalence relationship against a single variable, we should check the set of accessed variables.

For the same variable, it follows the rule of single variable analysis; for different variables, because $O_x^a O_y^b \equiv O_y^b O_x^a$, they can be interchanged without any interference.

We can extend the rule for single variable equivalence to multiple variables:

$$\begin{aligned}
 ut \neq tu \quad & \text{if} \quad (\mathbf{W}_t \cap \mathbf{R}_u \neq \emptyset) \vee (\mathbf{R}_t \cap \mathbf{W}_u \neq \emptyset) \\
 ut \neq tu \quad & \text{if} \quad (\mathbf{W}_t \cap \mathbf{W}_u \neq \emptyset) \wedge (\exists R \cdot D(R) \in \mathbf{W}_t \cap \mathbf{W}_u)
 \end{aligned}
 \tag{4.34}$$

where t and u are atomic blocks, which can be either p , q or m_i .

Other than these two cases, exchanges of symbols with multiple shared variable access will be equivalent.

4.4.2 Loop and branch structure

We have only considered cases for sequential code block in previous discussion. For cases with other control flows such as branch and loop, the analysis is slightly different.

For a loop structure, the loop must be able to be moved as a whole in order for fission to be safe. For example see Figure 4.4.2:

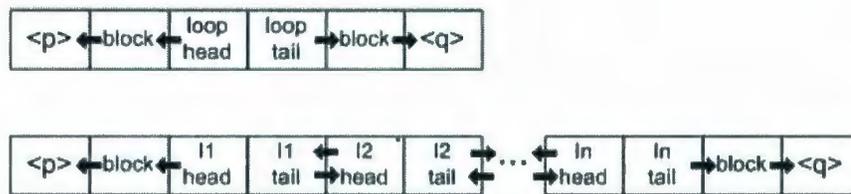


Figure 4.1: Example for loop structure that is not safe for fission.

Although the loop body can be divided into two separate parts, loop head and loop tail, and moved out of the $\langle p \rangle$ and $\langle q \rangle$ pair respectively, if we unroll the loops into different iterations, heads and tails from different iterations will block each other. Hence, this loop is not safe for fission in general.

In order to make the loop safe for fission, the loop body must be able to move to one direction as a whole, as in the example in Figure 4.4.2:

When unrolled, all the iterations of the loop can be moved toward the front of the trace, making the block safe for fission.

For branch flow, the code block within this structure can be divided into different

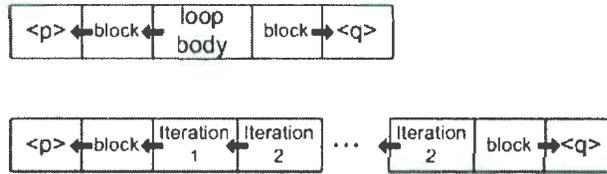


Figure 4.2: Example for loop structure that is safe for fission.

branches, the execution flow can take any of these branches. Because all branches can be chosen at an execution, we need to ensure none of the branches will interfere with the atomic block, as shown in Figure 4.4.2.

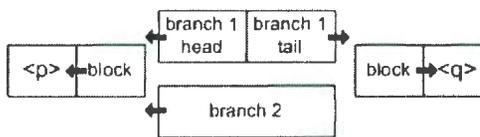


Figure 4.3: Example for branch structure that is safe for fission.

The branches in this cases are examined individually because only one of the branches will be chosen at any time. If all branches are safe for fission individually, the whole branch structure is safe for fission. The only thing that is should be noticed about branch structure is: if any one of the branches needs to be moved toward the end of the trace, all codes after the branch structure must be also move toward the end of the code; as this branch, if taken, prevents the code block after it from moving toward the head of the trace.

4.4.3 Mutual exclusion

We will extend the analysis to deal with flow control structures such as semaphore and wait statement. For example, if the atomic block is implemented by a semaphore, we wish to examine if the implementation correctly provides atomicity.

We divide the thread into different parts according to the position of semaphores, each part is then a block leading by either a P() or V() operation.

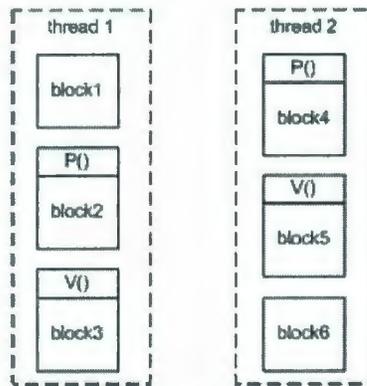


Figure 4.4: Example of thread divided by semaphores

When examining these two threads, we will find that block 2 and block 4 are mutually exclusive, hence they will not be executed at the same time with each other. As a result, we need only to examine the safety of fission between parts that can be executed concurrently, codes in blocks 2 and 4 will not cause any problem even if they may interfere with each other.

Chapter 5

Implementation of the compiler front-end

5.1 Structure of the compiler front-end

As we discussed in the first chapter, the HARPO/L compiler consists of a front-end and a number of back-ends targeting different platforms. The front-end takes the plain-text HARPO/L source code as input, and produces a platform-independent intermediate representation (Object graph) as the output. The HARPO/L front-end consists of several major steps: syntax analysis, type checking, and specialization/instantiation. Shown as Figure 5.1.

5.1.1 Syntax analyzer

The syntax analyzer checks the grammar of the source code and converts the plain-text source code into an abstract syntax tree (AST) reflecting the structure of the

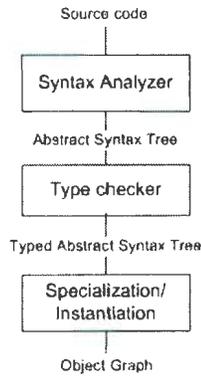


Figure 5.1: Structure of HARPO/L compiler front-end

program.

The abstract syntax tree built for statement $a := a + 1$ is shown as Figure 5.2.

The syntax analyzer first breaks the statement string into five tokens ($a, :=, a, +, 1$)

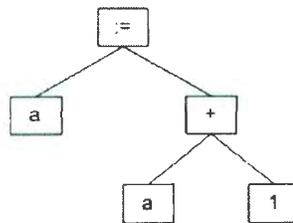


Figure 5.2: Abstract Syntax Tree for $a := a + 1$

according to the language definition, and then organize the five tokens into a tree structure according to the priority of the operators.

The syntax analyzer utilizes standard techniques.

5.1.2 Type checker

The AST derived by the syntax analyzer is not typed; the type checker checks the program for type errors and adds type information to each node. A typed abstract syntax tree for the example in Figure 5.2 is shown as Figure 5.3.

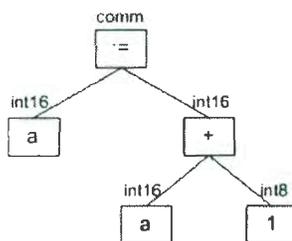


Figure 5.3: Typed Abstract Syntax Tree for `a := a + 1`

In this example, we assume variable `a` has been previously defined as an `int16`, which is an integer represented by a 16-bit word. According to the semantic rules of the language [9], the type of constant `1` is the shortest integer type that can hold its value, which is `int8`. The type of the expression combined by the `+` operator is the shortest super-type of both its operands, `int16`. The assignment command requires a check that the right operand can be assigned to the left operand. The type `comm` basically means the command is correctly typed; if an error had been found, the type of the statement will be set to `err` to reflect this fact.

The typing of generic classes is similar to that in Java [10]. Within generic classes, the types of nodes may be represented by ‘type variables’. Bounds information on type variables is used to determine the correctness of operations.

For example, within the generic class defined by

(class G {type T extends A } ...)

a declaration

obj x : T := new T ()

requires that type A supports a constructor with no parameters and a call

$x.m()$

requires that A exports a method m with no parameters.

Outside of G , a declaration

obj y := new $G\{U\}()$

requires type U to be a subtype of type A .

5.1.3 Specialization/ instantiation

HARPO/L uses generic classes to provide polymorphism. Specialization creates those specializations of generic classes that are needed for the given program.

Instantiation creates the objects from classes. Instantiation serves to connect objects together, by means of object references that are passed as constructor arguments. For example a consumer and a producer can be instantiated and connected together via the declarations

$$\begin{aligned} \mathbf{obj } p &:= \mathbf{new } Producer(c) \\ \mathbf{obj } c &:= \mathbf{new } Consumer(p) \end{aligned} \tag{5.1}$$

Detailed explanation about specialization and instantiation will be discussed in later sections.

5.2 Type system

The type checker is used to assign proper types to expressions and statements. In order to decide the type of variables used in expressions, the type checker must contain a symbol table that stores the type information for all the identifiers. All entities share the same name space; for example, to avoid ambiguity, once an identifier *List* is used as the name of a class, no object in the same scope can be named as *List* too. As a result, the symbol table contains not only types for objects, but also information about classes, interfaces, fields, methods, and generic parameters.

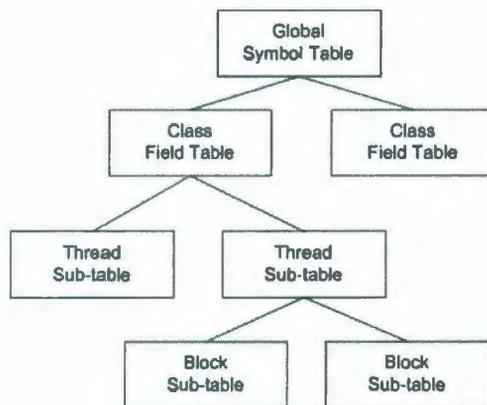


Figure 5.4: Example of symbol table hierarchy

The HARPO/L compiler uses a standard approach to organize its symbol table, an example shown as Figure 5.4. The symbol table is organized in a tree hierarchy, with each table being a hash table containing the (name, type) pair corresponding to all entries within it. A global symbol table is at the root of the tree, and all local symbol tables are descendants of the global table. When the compiler needs to find

the type of a specific identifier, it first check the current local table; if the entry can not be found, it then recursively checks the ancestral tables of the local table for that name, until it reaches the global table. If the name still can not be found in the global table, the type of the identifier will be set to *err* to indicate an error in the type checking.

Because the HARPO/L grammar allows use before declaration (as seen in 5.1, where consumer *c* is used as the constructor arguments before its declared), the type information of objects may not in the symbol table yet when it is used under this situation. Hence, the compiler should not query object types before the symbol table is fully filled, to avoid type error caused by missing entries in symbol table.

5.2.1 Types and expressions

There are various types in the HARPO/L language, each with different properties. As discussed in Chapter 2, type includes primitive type, class type, object type, array type, and special types such as *Err*.

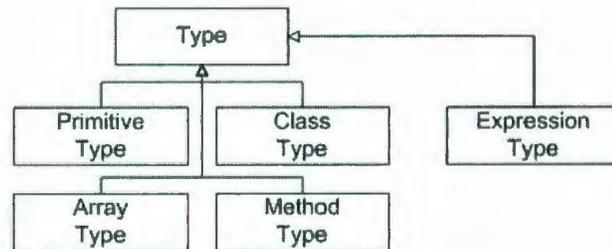


Figure 5.5: Different types in the type system

In the implementation, all classes representing specific types are extended from a

base class *Type*. This class provides functionalities common to all classes. Specific type classes may add new functions to the basic *Type* class, or override some of the functions to reflect unique behaviors for the specific type.

For example, *Array Type* is a type class that is used to describe any array. The array type must be based on some other type to reflect the content of the array.

Each *Array Type* object consists of reference to a *Type* object item that corresponds to the content of the array, and an integer *size* for the size of the array. In order to provide required functionalities for array operation, the basic *Type* class contains a *getArray* method to create an *Array Type* based on the *Type* called; and *Array Type* contains a method *getItem* to get the original array content type back. Hence, when the type checker sees a declaration like *int8[10]*, it first creates a *Primitive Type* of *int8*, then calls the *getArray* method to get a proper *Array Type* object.

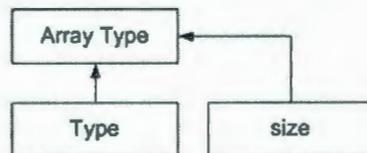


Figure 5.6: Structure of an *Array Type*

In addition to basic types, Figure 5.2.1 also contains a special type: *Expression Type*, this type is used to deal with a problem in the compiling process.

As discussed in Chapter 2, HARPO/L permits the type information of an object to be omitted from declaration when the type can be inferred from its initiation expression. However, the compiler may not always be able to figure out the type of an initiation expression when it reaches a variable declaration, see as the following

example:

```
obj a := new B()  
obj c := a.d  
obj e := c  
(class B  
    public obj d := 1  
class)
```

In this example, because the type is omitted in the declaration, the type of c is decided by the initiation expression. However, because its initiation expression contains a field d of class B , which is not yet declared, there is no way to decide the type of c at this point. In the next declaration, the type of e can not be decided either.

To deal with this problem, the HARPO/L compiler adopted a two-pass mechanism. The first pass fills in the symbol table with only variable names and their initial expressions, and type information for the variables that have an explicit type declaration. The second pass resolves initiation expressions for variables to decide their types.

For the first pass, the entries in symbol table for variable a will be a pair of its name and an *Object Type* B . However, the compiler will not try to collect any information about the type B , no matter whether it is available at this stage or not. Similarly, the compiler will create an *Expression Type* corresponding to expression $a.d$, but treat a and d only as two names, without concerning if they exist in the

symbol table. The compiler also records *Class Type B* to be a class with an *int8* field *d* equal to one in the first pass.

The second pass will evaluate all the *Expression Types* and decide the actual type of the variable. It will link the type *B* with its declaration, so when the compiler is trying to evaluate expression *a.d*, the information about field *d* will be available at this time.

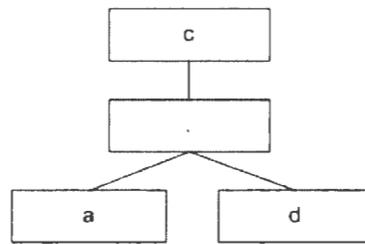


Figure 5.7: Expression type for *a.d*

For the above expression, the compiler will look up in the symbol table and find the left operand *a* to have type *B*, and then look up the field table of class *B* for the right operand *d* to decide the final type of the dot expression is *int8*. After the type of variable *c* is filled into the symbol table as *int8*, it can be used to determine the types of other variables such as *e*.

5.2.2 Type dependence

The example in the last section shows that the type of a variable may be decided by the type of other variables. In this case, we say that there is a type dependency relationship between these two variables. Furthermore, the example also shows that

the dependency relationship can run backwards in the source code, meaning that a variable may depend on another variable that has not been declared yet. For example, a field of a class is effective throughout the entire class, irrespective of in which part of the class it is defined. One may define fields of a class at the end of the class definition but use them earlier in the code to form initiation expressions of other local variables.

In the example from last section, this does not cause any complication because the type of class *a* and field *d* can be decided immediately at the first pass, so the type information is already available to the compiler in second pass. However, this situation may not always be true, see as the following example:

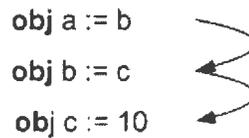


Figure 5.8: A simple example of type dependence

In this example, the type of *a* depends on type of *b*, and type of *b* further depends on type of *c*. When the compiler is trying to figure out the type of *a*, it needs to know the type of *b*, but the type information is not available in the symbol table because the initiation expression for *b* has not been evaluated yet. As a result, in order to correctly type this block of code, we must evaluate the variables in reversed order of their appearance in program. In actual programming, this order can be arbitrary so the situation may be even more complex.

One possible way to solve this problem is to perform a topological sort of the variables prior to the type checking. Variables that do not depend on any other vari-

able are evaluated first; variables that depend on available variables will be evaluated after. Repeat this step until no more variables can be evaluated. The variables that remain untyped after this is the set of variables that have non-satisfiable dependence requirement hence will all be typed as *err*.

This approach introduces a complex problem into the compiler; a good algorithm is needed in order to achieve both efficiency and correctness. In our implementation, we decided to employ a different approach.

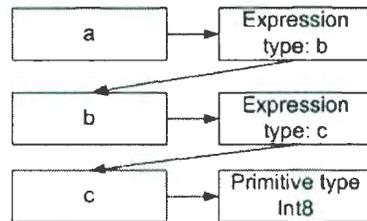


Figure 5.9: Symbol table entries and dependence relationship

We implement a method *getType* for the *Type* class. When this method is called, the *Type* class returns its own type if it is a normal *Type*, or evaluate itself to get its type if it is an *Expression Type*. So for the example in 5.2.2, the compiler performs type checking from top to bottom at the beginning, when it checks *a*, it will find it is an *expression type* that depends on variable *b*. The compiler then evaluates this expression by calling the *getType* method on the *expression type* (*c*) returned by the symbol table. The compiler further calls *getType* on this expression and it returns the type of *c*: *int8*. This type information is used to determine the type of *b* to be *int8*; type of *b* further determines the type of *a* to be also *int8*. If a variable depends on another variable that does not exist or have type *err*, this variable will be typed

as *err*.

This also introduces a new problem, cyclic dependency, as shown in Figure 5.2.2. This is a slightly modified version of the code block in last example. This time, the type of *c* is not a *Primitive Type int8*, but an *Expression type* that depends on the type of *a*.

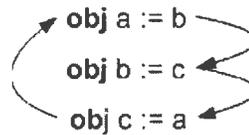


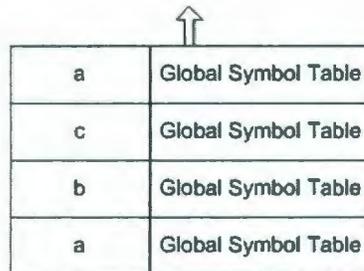
Figure 5.10: An example of cyclic dependence

Cyclic dependence will not cause any problem in topological sort approach because none of the dependence requirement of these three variable will be satisfied in the type checking process, so the compiler will never try to determine the type of these three variables. After the type checking finishes, these three variables that remain untyped will be all be typed as *err*.

However, in the recursive approach, this block of code will cause an infinite loop. When the compiler is type checking this block of code, it performs the same operation as the last example until it is trying to decide the type of *c*. As we discussed above, it will call *getType* on the *Expression Type (a)*, and this expression calls back the same method for *b* and the whole process starts over again and again.

In order to avoid this infinite loop problem and spot the cyclic dependency error, the compiler includes a stack to record all the identifiers involved in deciding the type of a single variable. The stack records the name and scope of a variable when it is

pulled from the symbol table, and pops it when the type of the variable is decided. The content of the stack when the compiler spots the cyclic dependency for this example is shown as Figure 5.2.2.



a	Global Symbol Table
c	Global Symbol Table
b	Global Symbol Table
a	Global Symbol Table

Figure 5.11: Symbol dependency stack for the cyclic dependency example

The stack contains two entries of symbol *a* from the global scope, this means the variable *a* depends on the type of itself, which clearly indicates a cyclic dependency. The compiler will output an error message and type the variable *a* as *err*. Consequently, all other variables in the stack will also be typed as error type because they depend on a variable that can not be correctly typed.

5.3 Specialization & Instantiation

The generic approach in HARPO/L makes it possible to write classes that can perform same operation to a number of different types. When writing such a class, fields and local variables can be typed as a *Generic Class Type*, so the actual type can be specified when an object of this class is defined.

An example of generic class is shown as follows:

```
(class List{type T extends int}
    public proc add(in value: T)
    ...
class)
```

In this example, the generic type T is bounded by the type int , which means it only accepts subtype of int ($int8$, $int16$, $int32$) as generic parameter. In order to implement this kind of bounded polymorphism, the compiler must be able to first verify the type parameter used in the declaration, and then fill in the type information required to replace the generic types.

Two possible methods can be used to implement the generic approach. One is to create a separate type for each different specialization of the *Generic Class Type*; the generic parameters are filled into the *Generic Class Type*, so generic classes that have different generic parameters are represented by two completely different *Class Type* objects. This approach requires a separate *Class Type* for every different specialization, but most information in the *Class Types* created this way will be duplicated, if the number or size of generic classes is large, this can be a great waste of system memory resource.

We use another approach to represent generic classes. This approach shares the common information between different specialization of a *Generic Class Type*; only the different parameters for each separate declaration are recorded.

As shown in figure 5.3, the data structure the compiler uses for a specialized class contains the *Generic Class Type* it is based on, and a table of generic parameters

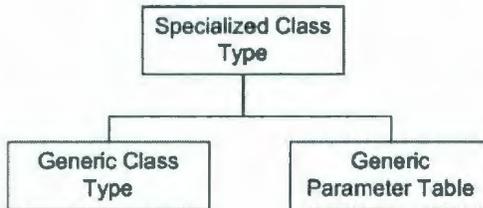


Figure 5.12: Structure for Specialized Class Type

used in declaration. In fact, the Generic Parameter Table is just a special form of symbol table that maps generic type variables to their actual defined types. When the compiler is trying to get information about the specialized class, the Specialized Class Type will redirect the method calls so the inquiry about common information goes into the *Generic Class Type*, only information about generic parameters specific to the specialized class is returned from the Generic Parameter Table.

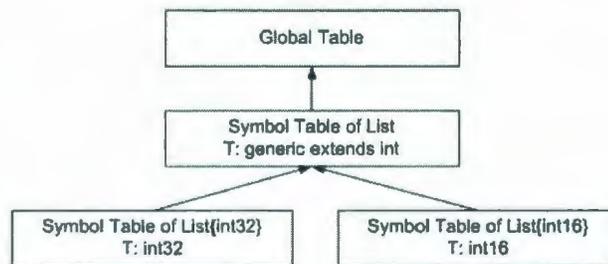


Figure 5.13: Symbol tables of generic and specialized table

In this way, different declarations of a same generic class will behave like two different classes; for example, they return different types for a same named field, and their methods also accept and return different types of the parameter, even though most of the information about them is from a same *Generic Class Type*.

This behavior is achieved by linking the symbol table of the generic and specialized class together, while the symbol table of generic class is the parent node, and generic parameter table of all declarations of specialized class is a child table linked to it. So when the compiler tries to get the information about the type of T in $List\{int32\}$, it will look up T in its generic table, which is $int32$, hence all occurrence of T in $List\{int32\}$ will become $int32$ instead. Because the generic table of $List\{int32\}$ only contains entry for generic parameter T , all other table look up call will be passed to its parent table as the default behavior of the symbol table. On the other hand, $List\{int16\}$ will have exactly the same behavior except the generic type T being replaced by $int16$ instead.

5.4 Implementation

We choose Java as the platform for the programming of the compiler front-end. In this section, we will discuss details of the actual coding of the compiler, including its structure, classes involved in the implementation, and interactions between classes.

5.4.1 Parsing

Parsing is used to convert the plain-text source code into a tree structure representing the meaning of the program.

We use JavaCC (Java Compiler Compiler) to implement the parser. Parsing a program using JavaCC contains two major steps, the first step is to break the source into a set of tokens, the second step is to organize these tokens into a tree structure.

When a string is read in from a source file, it is broken down into a number of

substrings divided by a set of divider strings, which include blank space, tab character, end of line character, and comments (comments in HARPO/L are defined as in Java). The substring set is then all substrings of the original string that are enclosed by two divider strings.

As a result, a string

$$\mathbf{obj} \text{ intExample} := 10 + 2 * (4 + 1) \quad (5.2)$$

is broken into four substrings: *obj*, *intExample*, *:=*, *10 + 2 * (4 + 1)* after this step.

The parser then checks the definition of different tokens to convert these substrings into tokens.

Token definitions involved in parsing 5.2 is shown as follows:

$\langle OBJ \rangle$	\leftrightarrow	obj
$\langle ASSIGN \rangle$	\leftrightarrow	:=
$\langle ADDITIVE \rangle$	\leftrightarrow	$\underline{+} \mid \underline{-}$
$\langle MULTIPLICATIVE \rangle$	\leftrightarrow	$\underline{*} \mid \underline{/} \mid \underline{\%}$
$\langle LP \rangle$	\leftrightarrow	(
$\langle RP \rangle$	\leftrightarrow)
$\langle \#DIGIT \rangle$	\leftrightarrow	$[0 - 9]$
$\langle \#LETTER \rangle$	\leftrightarrow	$[a - z] \mid [A - Z]$
$\langle INTEGER \rangle$	\leftrightarrow	$\underline{\langle DIGIT \rangle}^+$
$\langle IDENTIFIER \rangle$	\leftrightarrow	$\underline{\langle LETTER \rangle} \underline{(\langle LETTER \rangle \mid \langle DIGIT \rangle \mid \underline{-})}^*$

These definitions are in fact quite straight forward, we defined keyword *obj*, several operators, and two categories of tokens: integer and identifier. Token definitions that

start with # sign are macros, they are used to form other token definitions, but are not token types by themselves. As we can see from the definition, integer is defined as a sequence of digit, and identifier is defined as a letter followed by an arbitrary sequence of numbers, letters, and underscores.

We should note that token definitions are evaluated from top to bottom, so the substring *obj* will be converted into a $\langle OBJ \rangle$ token, even though it also satisfies the definition of $\langle IDENTIFER \rangle$ token.

When converting the substrings to tokens, the parser will try to include as many characters as possible into a token, until it encounters a character that can not be included into correct tokens according to the token definition. It will then combine all characters before this stop point into a token, and perform a new token check starting at the character where the previous check stopped. So for example, string *intExample* will be converted into a single token and $1 + 2$ will be three tokens 1, + and 2.

As a result, the string in 5.2 will be broken into a sequence of 12 tokens: $\langle OBJ \rangle$, $\langle IDENTIFER \rangle$, $\langle ASSGINMENT \rangle$, $\langle INTEGER \rangle$, $\langle ADDITIVE \rangle$, $\langle INTEGER \rangle$, $\langle MULTIPLICATIVE \rangle$, $\langle LP \rangle$, $\langle INTEGER \rangle$, $\langle ADDITIVE \rangle$, $\langle INTEGER \rangle$, $\langle RP \rangle$.

After converting the source into sequences of tokens, we need to define a parse tree to organize the tokens into a correct tree structure according to its meaning. The parse tree is derived from the language definition. A small portion of the parse tree is shown in Figure 5.14.

In this graph, tokens are represented by circles or eclipses, with tokens that will accept only a same sequence of characters (for example, $\langle OBJ \rangle$ can only be

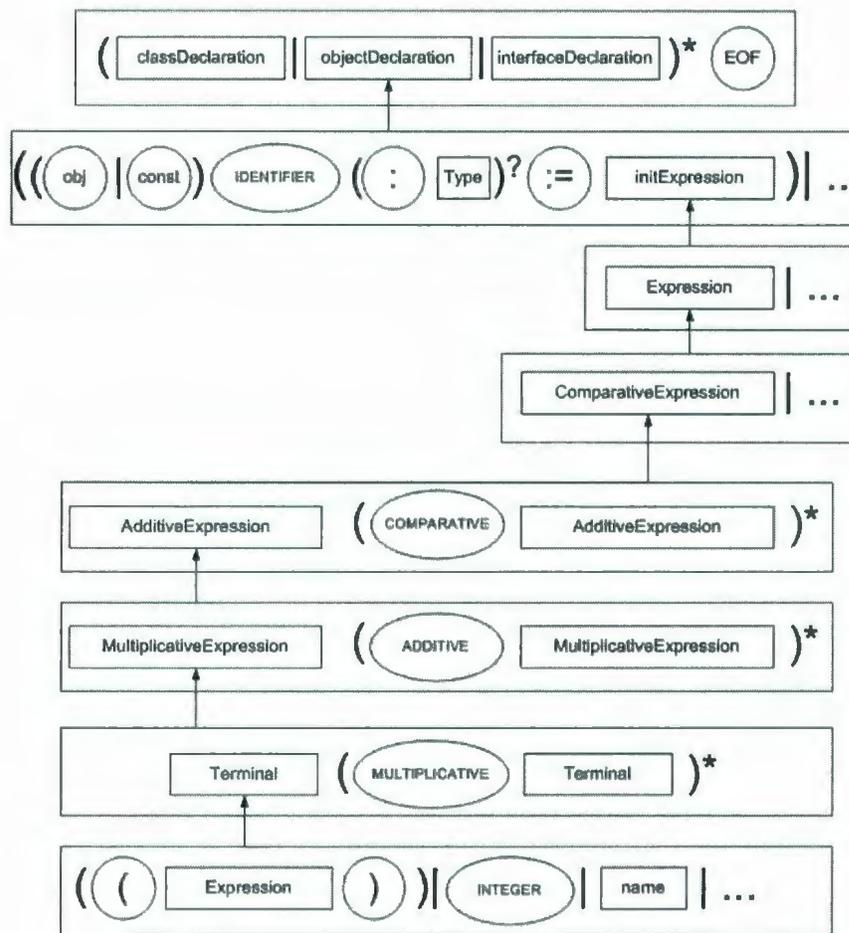


Figure 5.14: An example of parse tree

formed by string “obj”) represented by a circle labeled by that string for simplicity. Rectangles are used to represent branches that can be expanded into subtrees. A subtree may contain many branches, only branches that are involved in parsing (5.2) are included in the graph.

When the parser is trying to organize the tokens into a tree, it will perform a tree traversal. Starting at the root node, when a token is inputted, the parser will walk down into the branch that starts with the token, and consume tokens as specified by each tree node. The parser will return to the parent of a node when the token requirement of that node is satisfied. If the requirement for the root node is satisfied as the last token is consumed, the parsing is successful.

We will not list the detailed tree walking steps for (5.2), just to point out several important issues in this procedure.

- An Abstract Syntax Tree (AST) is generated in the tree walking procedure, when the parser returns from a parsing tree node, it will create a node in AST according to the parsing tree node, and attach this AST node to the AST node generated by the parent node in parsing tree.
- The priority of operators is determined by the depth of the node in the tree. Nodes close to the tree leaf will be evaluated earlier in the next step in compilation, so will have a higher priority. For example, *multiplicative expression* is the branch node of *additive expression*, hence the operator ($*$, $/$, $\%$) will have higher priority than additive operators.
- Parenthesis can be used to change the priority of expressions, as any expression enclosed by parenthesis will become a *terminal* and be brought back to the leaf

of the parse tree.

When the parsing is finished, we can guarantee that the syntax of the source code is correct. After this, we will check the Abstract Syntax Tree against the language semantics to produce typed AST.

5.4.2 Representing types

As discussed earlier, the HARPO/L compiler front-end uses a number of different classes to represent types, all of them extend from a root class *Type*. The symbol table is a tree of hash tables of (*String*, *Type*) pair.

The *Type* class is a base class of all types with a number of methods providing basic services, as shown in Figure 5.15:

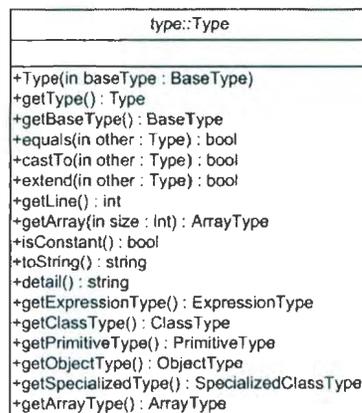


Figure 5.15: Class diagram for class *Type*

- Constructor *Type(BaseType)* can only be used to define *Type* object with for type error or type that can not be decided. All other types should be defined

as a specific type extending *Type*.

- *GetType()* are used to get the real type of the type object, as discussed earlier. So for an *ExpressionType*, calling *GetType()* method will cause the compiler to evaluate the expression and return the resulted type of the expression. Otherwise the type called will return itself.
- *equals(Type)*, *castTo(Type)* and *extend(Type)* method are used to compare between types. For example, an expression can only be assigned to a type that it can be casted to; generic parameters can only accept generic arguments with a type that extends it; an accept statement is only valid if there is a method in the class with all types of parameters equals to its own parameters.
- *getLine()* returns the line number in source code that this particular variable is defined. It can be used to sort the order of declarations and error messages for nicer output.
- *isConstant()* returns whether this type is compile time constant. Only *ExpressionType* and *PrimitiveType* can be constant. Expression for values such as array bounds and constructor arguments must evaluate to a type that is compile time constant.
- *toString()* and *detail()* provides two different ways of outputting a *Type* object. *toString()* provides the basic information about this type and *detail()* lists details such as initial expression of this type.
- *getBaseType()* returns an enumerate type corresponding to the actual type of this object. Enumerator *BaseType* will be discussed later.

- Get specific type methods such as *getPrimitiveType()* or *getObjectType()* is used to cast a *Type* object into a object of another type. It will return a specific type if its initiate expression can be eventually evaluated to that type or *null* if otherwise, irrespective of the actual type of the called type originally.

There are a number of different specific types that extend this *Type* class, each with a set of additional functionality according to that type. As a result, when we look up the table for a name and get a *Type* object, we would like to know to which specific type it belongs, and then convert the *type* object to the correct class.

An enumeration class *BaseType* is used to represent different types:

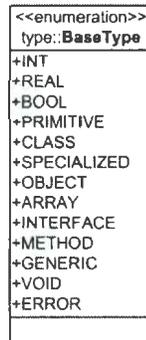


Figure 5.16: Class diagram for class *BaseType*

- *INT*, *REAL*, *BOOL* and *PRIMITIVE* are the set of primitives. Primitives are the only types that can be assigned with a value. They are represented by class *PrimitiveType*.
- *CLASS* and *INTERFACE* are represented by class *ClassType*, they are the same except *ClassType* with base type *INTERFACE* do not have fields for

constructor parameters and threads.

- *SPECIALIZED* are used for *SpecializedClassType*, it is the class type with generic argument filled in.
- *OBJECT* are used for *ObjectType*, representing instantiated objects.
- *ARRAY* are used for *ArrayType*, as discussed earlier.
- *METHOD* are used for *MethodType*, a *MethodType* can only be contained in the field table of a class. A method is also a type because it shares the same name space with other types. Once a method is declared, no fields with the same name can be defined in this scope.
- *GENERIC* are used for *GParamType*, which are used for generic parameters. It also shares a same name space with other fields of the class, hence prevents any field with the same name from being defined. This type will not be seen by any field reference because any correct instantiation of a generic class should have a field with same name in the field table of the corresponding *Specialized-ClassType*, which will shade this type from the descendant tables.
- *ERROR* and *VOID* is used only for types with errors in type checking.

5.4.3 Expressions

Among all specific types, we would like to discuss *ExpressionType* in detail as it is significantly different from other specific types, as shown in Figure 5.17.

In addition to all the standard methods of *Type*, expression type has several methods and fields of its own.

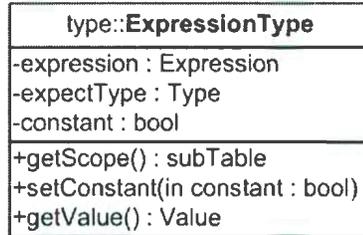


Figure 5.17: Class diagram for *ExpressionType*

The use of the three fields is demonstrated by the following example:

```
const intExample : int32 := 20
```

- The keyword **const** affects the constant field of the *ExpressionType*. The meaning of an *ExpressionType* type being constant is different than an expression being constant. An expression can be constant if all operands of it are constant. However, an *ExpressionType* can only be constant if explicitly declared. Hence, code line

```
obj intExample : int32 := 20
```

declares a object with the same name but not constant.

- The *int32* part goes into the *expectType* field. So this *ExpressionType* will evaluate the type specified by this field instead of the one returned by the initiation expression, as long as the type from initiation expression can be casted to this *expectType*.
- The initiation expression 20 will be recorded by the *Expression* field, the functionality of *Expression* class will be discussed later.

The three access methods of the *ExpressionType* are based on the expression it contains, so they will be discussed as we discuss class *Expression*.

The *Expression* class is used to represent the expressions in the program, as shown in Figure 5.18.

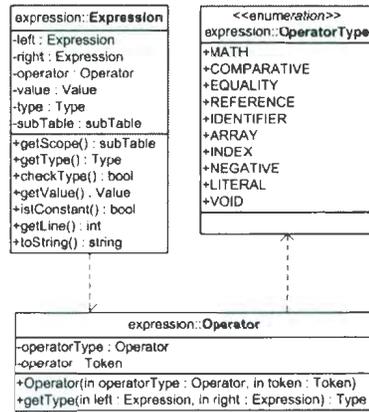


Figure 5.18: Class diagram for *Expression*, *Operator* and *OperatorType*

An *Expression* will consist of a *Type* or *Value* object if it represents the leaf node of the expression tree, or two *Expression* objects if it is the branch node of the tree. Either way, an *Expression* object will also contain an *Operator* object which specifies the operation taken in this expression. Note that the operators are categorized by resulted types instead of priorities in this step. For example, although “+” and “*” operators have different priorities and are treated differently in the parsing step, they will result in a same type for any expression pair hence are both categorized as *MATH* operators. On the other hand, although “>” and “==” have same priority, they are not same in type checking (for example, *boolean == boolean* is correct but *boolean > boolean* does not make sense), so they are categorized as *COMPARATIVE*

and *EQUALITY* operators respectively.

In addition to the methods that are same with *Type* classes and are used to serve the same named method of *ExpressionType*, *Expression* also contains several unique methods and fields for evaluating expressions.

- *getValue()* will return a *Value* object that represents the result value of the expression, only constant expression can have values.
- *SubTable* field is used to record the scope the expression is defined in, as expressions that contain identifiers will have different meaning in different scope.
- *getScope()* method is used mainly to deal with cyclic dependence problem.
- *checkType()* checks if the semantics of the expression is correct, it will return *false* and type the expression as *ERROR* if the expression is not valid.

5.5 Working examples

A number of working examples will be shown in this section. The first example will be the sample code shown at the end of Chapter 2, namely the following block of code:

```
(class FIFO {type T extends primitive}
    constructor(in capacity : int)

    public proc deposit(in value : T)
```

```

public proc fetch(out value : T)

private obj a : T(capacity)
private obj front := 0
private obj size := 0

(thread
  (wh true
    (accept
      deposit( in value : T ) when size < capacity
      a[ (front + size) % capacity ] := value
      size := size + 1
    |
      fetch( out value : T ) when size > 0
      value := a[front]
      front := (front + 1) % capacity
      size := size - 1
    accept)
  wh)
thread)
class)
obj producer := new FIFO{int32}(40)

```

The above code block creates a FIFO object using generic parameter (*int32*) and constructor parameter (40). The execution result is shown as follows:

Detailed field list:

FIFO : FIFO = FIFO generic <: primitive construct (in int8: capacity)

producer : (FIFOint32 obj) = (<(FIFOint32 obj)> new FIFOint32(<int8> 40))

Field dump for object (FIFOint32 obj):

new FIFOint32(<int8> 40)

{

private T : int32 = int32

private const capacity : const int8 = (<const int8> 40)

public deposit(in int32: value)

public fetch(in int32: value)

private a : int32[40] = (<int8[40]> (<int8> 0)(<const int8> capacity))

private front : int8 = (<int8> 0)

private size : int8 = (<int8> 0)

(thread

(wh (<bool> true)

(accept

deposit(in int32: value) when (<bool> (<int8> size)<(<const int8> capacity))

(<int32> (<int32[40]> a)[(<int8> (<int8> (<int8> front)+(<int8> size))%(<const int8> capacity))]) := (<int32> value)

```

(<int8> size) := (<int8> (<int8> size)+(<int8> 1))
|
fetch(out int32: value) when (<bool> (<int8> size)>(<int8> 0))
(< int32 > value) := (<int32> (<int32[40]> a)[(<int8> front)])
(<int8> front) := (<int8> (<int8> (<int8> front)+(<int8> 1))%(<const int8> ca-
pacity))
(<int8> size) := (<int8> (<int8> size)-(<int8> 1))
accept)
wh)
thread)
}

```

The output is just a plain string representation of the information and structure of the typed AST obtained by the compiler front-end. Each section enclosed by parenthesis is a node expression of the typed AST, while the type enclosed by <> is the type of the node. Combining with operators, node expressions can form high level expressions. This composition continues until a statement is formed.

From this example we can see the compiler correctly types the generic argument T to *int32* and makes constructor argument a constant number of 40 as intended, and also correctly types other variables based on this information (for example, $a : T(\textit{capacity})$ in the source code is correctly typed as *int32[40]* in the typed AST for object *FIFO{int32}[40]*).

The following example shows functionality of symbol table as well as error report-

ing of the compiler front-end:

```
obj int1 := 5
```

```
obj int1 := 10
```

```
(class testClass1 constructor()
```

```
    public obj real1 := 10.6
```

```
    public obj int1: int32 := 20
```

```
    private obj int2 := 10
```

```
    public const int3 := 15
```

```
    private obj int4 := int3
```

```
    (thread
```

```
        int3 := 10
```

```
        int4 := 10
```

```
    )
```

```
)
```

```
obj object1 := new testClass1()
```

```
obj real1 := object1.real1
```

```
obj error1 := object1.int2
```

```
obj error2 := object1.int7
```

```
obj int2 := int1
```

```
obj int3 := object1.int1
```

obj cyclic1 := cyclic2

obj cyclic2 := cyclic3

obj cyclic3 := cyclic1

The output for this example is:

Detailed field list:

int1 : int8 = (<int8> 5) testClass1 : testClass1 = testClass1

object1 : (testClass1 obj) = (<(testClass1 obj)> new testClass1)

real1 : real16 = (<real16> (<(testClass1 obj)> object1).real1)

error1 : error = (<error> (<(testClass1 obj)> object1).int2)

error2 : error = (<error> (<(testClass1 obj)> object1).int7)

int2 : int8 = (<int8> int1)

int3 : int32 = (<int32> (<(testClass1 obj)> object1).int1)

cyclic1 : error = (<error> cyclic2)

cyclic2 : error = (<error> cyclic3)

cyclic3 : error = (<error> cyclic1)

Field dump for object (testClass1 obj):

new testClass1

{

public real1 : real16 = (<real16> 10.6)

public int1 : int32 = (<int8> 20)

private int2 : int8 = (<int8> 10)

```

public const int3 : const int8 = (<const int8> 15)
private int4 : const int8 = (<const int8> int3)

(thread
<error> (<const int8> int3) := (<int8> 10)
(<const int8> int4) := (<int8> 10)
thread)
}

```

Line 2: Multiple declaration of identifier int1 in same scope.

Line 11: Can not assign value to constant variable (<const int8> int3)

Line 18: Can not access private field int2 of (<(testClass1 obj)> object1)

Line 19: Error getting field, int7 is not a field of (<(testClass1 obj)> object1)

Line 23: Cyclic dependency for identifier cyclic2

Line 24: Cyclic dependency for identifier cyclic3

Line 25: Cyclic dependency for identifier cyclic1

This example shows how symbol table and scope rule works for the compiler, it also shows that the compiler correctly identifies program errors such as overlap declaration, assign to constant variable, and cyclic dependence.

Although a number of other sample programs have been used to test the correctness of the program, a formal and thorough test is still needed to find and remove potential errors within the program and improve the quality of the compiler front-end.

Chapter 6

Conclusion and future work

6.1 Thesis summary

In this thesis, we have addressed a number of technical issues involved in designing a language, HARPO/L, that can be compiled into CGRA configurations and executed in hardware, and developed a front end for the language compiler.

Firstly, we did a brief discussion on the difference between using software and hardware methods in solving problems, and then introduced reconfigurable architectures, especially CGRA, as a third solution to combine some of the advantages of software and hardware.

Then we introduced the overall structure of the HARPO/L project, which contains compiler front-end, software back-end and hardware back-end. In this thesis, we have mainly concentrated on the compiler front-end.

We have discussed the language design in the second chapter. HAROP/L is similar to common high-level programming languages except the following characteristics:

- HARPO/L allows explicit declaration of parallel execution, which allows the output hardware configuration to benefit more from the parallel nature of hardware.
- HARPO/L uses a generic system and implicit type inference to make the reusing of code segments easier.
- The method calling in HARPO/L is different from typical high-level programming languages to reflect the nature of hardware.
- HARPO/L has explicit declaration of atomic block to allow easier parallel programming.

In chapter 3, we have developed a Colored Petri Net representation of the HARPO/L, that serves as a formal mathematical representation of the language, to allow various analysis. The CPN representation uses places to represent different states of the system, and uses the motion of control tokens to represent the flow of execution in the different threads of the program.

Chapter 4 introduced two operations, fission and fusion, to help simplify the implementation of atomic block, and the code optimization in parallel environment. We utilized trace theory to develop a method to determine if a particular fission is safe under certain context, or to identify the section of code that makes the fission not safe.

Chapter 5 discussed issues involved in developing the HARPO/L compiler front end, and we have addressed a number of issues that makes the implementation of it different than typical compiler front end.

6.2 Thesis Contribution

HARPO/L is a language that is designed to compile into CGRA configuration. The purpose of this thesis is to develop a front end for the compiler, and to solve various problems arising in the development. The main contribution of this thesis is listed as follows:

- We developed a Colored Petri Net representation for the HARPO/L language. Source code written in HARPO/L can then be converted to this formal mathematical representation. Given many analysis tools available for CPN, we can formally examine the behavior and property, such as safety and liveness, for the source code.
- We developed a method to determine the safeness of the fission operation. Because the result is determined by other threads running in parallel with the thread under analysis, the problem can grow overly complicated, even for computer analysis, when the length or number of other threads grows. Our method is based on analyzing the interleaving of shared variable access operations using trace theory. It simplifies the problem into a number of small and simple analyses, to greatly reduce the complexity of the analysis.
- We coded the compiler front end using Java. This front end takes HARPO/L source code as input, performs syntax analysis and type checking for the code, and produce a typed abstract syntax tree as output.

6.3 Open issues for future work

Future work to continue the research done in this thesis could include:

- **Colored Petri Net representation for intermediate representation**

A CPN representation for intermediate representation can also be developed, in comparing the CPN representation for source code and intermediate representation, one can analyze if the intermediate representation derived from source code preserves its various properties.

- **A thorough test of the compiler front end**

Although the compiler front end works correctly against a number of test cases we have done so far, it may still contain unknown problems. A thorough test should be performed to discover and remove potential problems and improve the quality of the front end.

- **Software and hardware back end**

Back ends can be developed to convert the result obtained by the front end into software code or hardware configurations, to complete the compiler development.

Appendix A

Language Design for CGRA project. Design 5 [Draft].

Theodore S Norvell

Electrical and Computer Engineering

Memorial University

Meta notation

$N \rightarrow E$	Nonterminal N can be an E
(E)	Grouping
E^*	Zero or more
E^*F	Zero or more separated by F s
E^+	One or more
E^+F	One or more separated by F s
$E?$	Zero or one
$[E]$	Zero or one
$E F$	Choice

A.1 Classes and Objects

A.1.1 Programs

A program is a set of classes, interfaces, and objects.

$$Program \rightarrow \underline{(ClassDecl | IntDecl | ObjectDecl | ConstDecl | ;)^*}$$

A.1.2 Types

Types come in several categories.

- Primitive types: Primitive types represent sets of value. As such they have no mutators. However objects of primitive types may be assigned to, to change their values. Primitive types represent such things as numbers. They include

– int8, int16, int32, int64, int

- real16, real32, real64, real
- bool
- Classes: Classes represent sets of objects. As such they support methods that may change the object's state.
- f s. Interfaces are like classes, but without the implementation.
- Arrays: Arrays may be arrays of primitives or arrays of objects.
- Generic types. Generic types are not really types at all, but rather functions from some domain to types. In order to be used, generic types must be instantiated.

Types are either names of classes, array types or specializations of generic types

$$Type \rightarrow Name \mid Name \ GArgs \mid Type[Bounds]$$

Arrays are 1 dimensional and indexed from 0 so the bounds are simply one number

$$Bounds \rightarrow ConstIntExp$$

A.1.3 Objects

Objects are named instances of types.

$$ObjectDecl \rightarrow \mathbf{obj} \ Name \ [: Type] := InitExp$$

The Type may not be generic.

Initialization of an object can be an expression or an array initialization

$$\begin{aligned}
InitExp &\rightarrow Exp \mid ArrayInit \mid \mathbf{new} Type(CArg^+) \\
&\mid \left(\mathbf{if} Exp \mathbf{then} InitExp \left(\mathbf{else} \mathbf{if} Exp InitExp \right)^* \mathbf{else} InitExp \left[\mathbf{if} \right] \right) \\
ArrayInit &\rightarrow \left(\mathbf{for} Name : Bounds \mathbf{do} InitExp \left[\mathbf{for} \right] \right) \\
CArg &\rightarrow Exp
\end{aligned}$$

- If the object to be initialized is of a primitive type (such as **int32** or **real64**), the *initExp* should be a compile-time constant expression of a type assignable to the type of the object.
- If the object to be initialized is an array, then the *InitExp* should be an *ArrayInitExp*.
- If the object to be initialized is an object of non-primitive type, then the *InitExp* should be of the form **new** *Type*(*Args*) where the *Type* is a non-generic class type.
- Constructor arguments must either represent objects or compile time values, depending on whether the corresponding parameter is **obj** or **in**.
- In any case, the *InitExp* can be an if-else structure in which the expression is a compile-time constant assignable to **bool**.
- The *InitExp* must have a type that is a subtype of the *Type*.

A.1.4 Constants

A constant is simply a named constant expression

$$ConstDecl \rightarrow \mathbf{const} \textit{Name} \text{ [: } \textit{Type} \text{] } := \textit{ConstExp}$$

The type, if present must be primitive. Constant expressions are always primitive.

A.1.5 Classes and interfaces

Each class declaration defines a family of types. Classes may be generic or nongeneric.

A generic class has one or more generic parameters

$$ClassDecl \rightarrow \left(\mathbf{class} \textit{Name} \textit{GParams}^? \text{ (} \mathbf{implements} \textit{Type}^+ \text{)}^? \mathbf{constructor}(\textit{CPar}^+) \text{ (} \textit{ClassMem} \right)$$

- The *Name* is the name of the class.
- The *GParams* is only present for generic classes, which will be presented in a later section.
- The *Types* are the interfaces that the class implements.

An interface defines a type. Interfaces may be generic or nongeneric. A generic interfaces has one or more generic parameters

$$IntDecl \rightarrow \left(\mathbf{interface} \textit{Name} \textit{GParams}^? \text{ (} \mathbf{extends} \textit{Type}^+ \text{)}^? \text{ (} \textit{IntMember} \text{)}^* \text{ [} \mathbf{interface} \text{ [} \textit{Name} \text{]} \text{]} \right)$$

- The *Name* is the name of the class.

- The *GParams* will be presented in a later section.
- The *Types* are the interfaces that the interface extends.

Constructor parameters represent objects to which this object is connected.

$$CPar \rightarrow \mathbf{obj} \ Name : Type \mid \mathbf{in} \ Name : Type$$

- Object parameters represent named connections to other objects. So for example if we have

```
(class B constructor( obj x : A ) ... )
obj a := (for i : 10 do new A() )
obj b := (for i : 10 do new B(a0) )
```

Then object `b[0]` knows object `a[0]` by the name of `x`.

- In parameters are compile time constants and the corresponding argument must be such.

A.1.6 Class Members

Class members can be fields, methods, and threads. [Nested classes and interfaces are a possibility for the future.]

$$ClassMember \rightarrow Field \mid Method \mid Thread \mid ConstDecl \mid ;$$

Fields are objects that are within objects. Field declarations therefore define the

part/whole hierarchy.

$$Field \rightarrow Access \mathbf{obj} \ Name[: Type] := InitExp$$
$$Access \rightarrow \mathbf{private} \mid \mathbf{public}$$

Method declarations declare the method, but not its implementation. The implementation of each must be embedded within a thread.

$$Method \rightarrow Access \mathbf{proc} \ Name((Direction [Name :] Type]*)]$$
$$Direction \rightarrow \mathbf{in} \mid \mathbf{out}$$

The types of parameters must be primitive.

Recommended order of declarations is

- public methods and fields, followed by
- private methods and fields, followed by
- threads.

There is no ‘declaration before use rule’. Name lookup works from inside out.

A.1.7 Interface Members

Interfaces members can be fields and methods. [Nested classes and interfaces are a possibility for the future.]

$$IntMember \rightarrow Field \mid Method \mid Const Decl \mid ;$$

A.2 Threads

Threads are blocks executed in response to object creation.

$$Thread \rightarrow (\mathbf{thread} \textit{Block} \textit{[_thread]_})$$

Each object contains within it zero or more threads. Coordination between the threads within the same object are the responsibility of the programmer. All concurrency within an object arises from the existence of multiple threads in its class. Thus you can write a monitor (essentially) by having only one thread in a class.

A.2.1 Statements and Blocks

A block is simply a sequence of statements and semicolons

$$Block \rightarrow (\textit{Statement} \textit{| ;} \textit{;})^*$$

Statements as follow

- Assignment statements

$$\textit{Statement} \rightarrow \textit{ObjectIds} := \textit{Expressions}$$

$$\textit{ObjectIds} \rightarrow \textit{ObjectId} \textit{ (, ObjectId)^*}$$

$$\textit{Expressions} \rightarrow \textit{Expression} \textit{ (, Expression)^*}$$

$$\textit{ObjectId} \rightarrow \textit{Name} \textit{ | ObjectId[Expression] | ObjectId.Name}$$

The type of the `ObjectId` must admit assignment, which means it should be a primitive type, like `int32` or `real64`.

- Local variable declaration

$$Statement \rightarrow \mathbf{obj} \ Name[: Type] := \ InitExp \ Block$$

Same restrictions as fields. The type may be omitted, in which case it is inferred from the initialization expression. The block part contains as many statements as possible. The scope of a local variable name is the block that follows it.

- Constant Declarations

$$Statement \rightarrow \mathbf{ConstDecl} \ Block$$

The block part contains as many statements as possible. The scope of a local constant name is the block that follows it.

- Method call statements

$$Statement \rightarrow \mathbf{ObjectId.Name}(Args)$$

$$| \ Name(Args)$$

- Sequential control flow

$$Statement \rightarrow \left(\mathbf{if} \ Expression \ \mathbf{then} \ Block \ (\mathbf{else} \ \mathbf{if} \ Expression \ Block)^* (\mathbf{else} \ Block)^? \ [\mathbf{if}] \right)$$

$$| \left(\mathbf{wh} \ Expression \ \mathbf{do} \ Block \ [\mathbf{wh}] \right)$$

$$| \left(\mathbf{for} \ Name : Bounds \ \mathbf{do} \ Block \ [\mathbf{for}] \right)$$

- Parallelism

$$Statement \rightarrow \left(\mathbf{co} \ Block \ (|| \ Block)^* \ [\mathbf{co}] \right)$$

$$| \left(\mathbf{co} \ Name : Bounds \ \mathbf{do} \ Block \ [\mathbf{co}] \right)$$

In the second case, the *Bounds* must be compile-time constant.

- Method implementation.

$$Statement \rightarrow (\mathbf{accept} \textit{MethodImp} _ (| \textit{MethodImp} _)^* _ [\mathbf{accept}] _)$$

$$\textit{MethodImp} \rightarrow \textit{Name} (_ (\textit{Direction} \textit{Name} : \textit{Type} _)^* _) _ [\textit{Guard}] _ \textit{Block}_0 _ [\mathbf{then} _ \textit{Block}_1] _)$$

$$\textit{Guard} \rightarrow \mathbf{when} _ \textit{Expression}$$

– Restrictions

- * The directions and types must match the declaration.
- * The guard expression must be boolean.
- * Each method may only be implemented once per class

– Possible restrictions:

- * The guard may not refer to any parameters.
- * The guard may refer only to the in parameters.

- Semantics: A thread that reaches an accept statement must wait until there is a call to one of the methods it implements and the corresponding guard is true. Once there is at least one method the accept can execute, one is selected. Input parameters are passed in, \textit{Block}_0 is executed and finally the output parameters are copied back to the calling thread. If there is a \textit{Block}_1 it is executed next.

- Sequential consistency

$$Statement \rightarrow (\mathbf{atomic} _ \textit{Block} _ [\mathbf{atomic}] _)$$

The block is executed as-if atomically. That is, any two atomic statements within the same object can not execute at the same time unless they can not interfere with each other.

A.3 Expressions

[[To Be Completed]]

A.4 Genericity

Classes and interfaces can be parameterized by “generic parameters”. The effect is a little like that of Java’s generic classes or C++’s template classes. Classes and interfaces may be parameterized, in general, by other classes and interfaces, values of primitive types, for example integers, and objects.

Programs using generics can be expanded to programs that do not use generics at all. For example a program

```
(class K ... class)
(class G{ type T } ...T... class)
obj g : G{K} := ...
```

Expands to

```
(class K ... class)
```

```
obj k : K
(class G0 ...T... class)
obj g : G0 := ...
```

Generic parameters may be one of the following

- Nongeneric Types
- Nongeneric Classes

$$GParams \rightarrow \{GParam^+\}$$
$$GParam \rightarrow \text{type Name} [\text{extends Type}]$$
$$GArgs \rightarrow \{Type^+\}$$

A.5 Examples

```
(class FIFO {type T extends primitive}
  constructor(in capacity : int)

  public proc deposit(in value : T)
  public proc fetch(out value : T)
```

```

private obj a : T(capacity)
private obj front := 0
private obj size := 0

(thread
  (wh true
    (accept
      deposit( in value : T ) when size < capacity
        a[ (front + size) % capacity ] := value
        size := size + 1
      |
      fetch( out value : T ) when size > 0
        value := a[front]
        front := (front + 1) % capacity
        size := size - 1
    accept)
  wh)
thread)
class)

```

A.6 Lexical issues

Appendix B

The Static Semantics of HARPO/L

Theodore S Norvell

Electrical and Computer Engineering

Memorial University

B.1 Abstract Syntax

We present the abstract syntax of the language as a phrase structured (context-free grammar).

B.2 Types

B.2.1 Typing relation

Each well-formed phrase of the language is associated with some phrase type. A context is a mapping from identifiers to phrase types. If E is a phrase of the abstract

syntax, t is a phrase type, and Γ is a context, we write

$$\Gamma \vdash E : t$$

to mean that phrase E has type t in context Γ .

For example

$$\Gamma \vdash 1 = 2 : \text{bool}$$

This typing relation is specified by a set of inference rules written

$$\frac{\text{assumptions}}{\text{conclusion}}$$

The domain of the context is always a finite set of identifiers.

The typing relation is intended to define a partial function from contexts and phrases to types

Types and objects

Objects	o	$::=$	$\text{obj}_{rw}(t)$
Types	t, u, v	$::=$	$p \mid \text{array}(t) \mid \alpha \text{ boundedby } k \mid k$
Prim. types	p, q	$::=$	$\text{bool} \mid \text{int8} \mid \text{int16} \mid \text{int32} \mid \text{float16} \mid \text{float32} \mid \text{float64}$
Class and interface types	k	$::=$	$c \langle \bar{a} \rangle$
Type variables	α, β		
Generic arguments	a, b	$::=$	t
Read/Write Mode	rw	$::=$	$r \mid w$
Values	V		
Class identifiers	c, d		

Context

Context $\Gamma ::= x \mapsto o, \Gamma \mid x \mapsto m, \Gamma \mid x \mapsto c, \Gamma \mid x \mapsto (\alpha \text{ bounded by } k), \Gamma \mid \varepsilon$

Methods $m ::= [TBD]$

Class environment. A class environment is a partial function from class identifiers to symbol table entries for classes and interfaces. A class or interface symbol table entry records the declarations of the class, the set of interfaces it extends (empty for classes) and the set of interfaces it implements (empty for interfaces).

Class Environment $\Theta ::= c \mapsto cid, \Theta \mid \varepsilon$

Class and interface declarations $cid ::= \lambda \bar{g} \cdot \text{clint}_{ci}(\Delta, \bar{t}, \bar{u})$

Class or interface $ci ::= \text{class} \mid \text{interface}$

Generic parameter $g ::= x <: t$

Member Declarations $\Delta ::= TBC$

B.3 Building a class environment

We can analyse each class and interface in two passes. The first pass builds a class environment. The second pass does type checking and inference.

In the first pass, we record information about each class and interface in the class environment (Θ). Since this is done before type checking, all that can be done is to record information in a raw form. For each class declaration

(class x implements $\bar{i} D$)

we add an entry

$c \mapsto \lambda \varepsilon \cdot \text{clint}_{\text{class}}(\Delta, \varepsilon, \bar{u})$

to the class environment, where c is the fully qualified name for the class, Δ is derived from D , and \bar{u} is derived from \bar{i} . Similarly for each interface declaration

(interface x extends \bar{i} D)

we add an entry

$$c \mapsto \lambda \varepsilon \cdot \text{clint}_{\text{class}}(\Delta, \bar{t}, \varepsilon)$$

to the class environment, where c is the fully qualified name for the interface, Δ is derived from D , and \bar{t} is derived from \bar{i} .

When there are generic parameters,

type $x_i <: E_i$

we create new type variables α_i and add constraints $\alpha_i <: t_i$ to between the λ and the \cdot . Each t_i is derived from each E_i by replacing identifiers representing classes with the corresponding class identifier, replacing braces with angle brackets, replacing each x_i with the corresponding α_i and so on. [To do: Formalize this.]

Deriving Δ from the sequence of declarations D is done by a similar process. The type expressions used in field declarations, method declarations, and constructor arguments are turned into types t using a superficial analysis. [To do: Formalize this.]

After the first pass is completed for the whole program, we can do full type checking on the whole program.

B.4 Types of expressions

B.4.1 Identifiers are looked up in the context

The type of an identifier can be looked up in the context. This is the only rule for identifiers, so an identifier not in the current context results in a type error.

$$\frac{E \text{ is an identifier} \quad E \in \text{dom}(\Gamma)}{\Gamma \vdash E : \Gamma(E)} \quad (\text{LOOKUP})$$

B.4.2 Constants

For constants of the language we have

$$\frac{E \text{ is an integer constant in } \{-128, \dots, +127\}}{\Gamma \vdash E : \text{obj}_r(\text{int8})}$$

$$\frac{E \text{ is an integer constant in } \{-2^{15}, \dots, +2^{15} - 1\}}{\Gamma \vdash E : \text{obj}_r(\text{int16})}$$

$$\frac{E \text{ is an integer constant in } \{-2^{31}, \dots, +2^{31} - 1\}}{\Gamma \vdash E : \text{obj}_r(\text{int32})}$$

[TBD: Similar for float]

B.4.3 Arithmetic expressions

Generally, unary expressions leave the type alone, while binary expressions require the operands to have the same type and produce the same result type. When the operand types are different, there must be a widening conversion from one to the other.

Subtypes are given by the following rules: [[Does this make sense?]]

$$\frac{}{\overline{\text{int8} <: \text{int16}}} \quad \frac{}{\overline{\text{int16} <: \text{int32}}}$$

$$\frac{}{\overline{\text{float16} <: \text{float32}}} \quad \frac{}{\overline{\text{float32} <: \text{float64}}}$$

Furthermore, subtyping is transitive and reflexive

$$\frac{t <: u \quad u <: v}{t <: v}$$

$$\frac{}{\overline{t <: t}}$$

All primitive types are subtypes of the built-in interface `primitive`.

$$\frac{}{\overline{p <: \text{primitive} \langle \rangle}}$$

The following two rules illustrate the typing rules for the binary arithmetic operations on integers. The rules show that either operand may be widened, but not both.

$$\frac{\Gamma \vdash E : \text{obj}(p) \quad \Gamma \vdash F : \text{obj}(q) \quad p <: q \quad p, q \in \{\text{int8}, \text{int16}, \text{int32}\} \quad \oplus \in \{+, -, *, \text{div}, \text{mod}\}}{\Gamma \vdash E \oplus F : \text{obj}_r(q)}$$

$$\frac{\Gamma \vdash E : \text{obj}(p) \quad \Gamma \vdash F : \text{obj}(q) \quad q <: p \quad p, q \in \{\text{int8}, \text{int16}, \text{int32}\} \quad \oplus \in \{+, -, *, \text{div}, \text{mod}\}}{\Gamma \vdash E \oplus F : \text{obj}_r(p)}$$

[Arithmetic expressions to be completed.]

B.4.4 Arrays

Arrays can be indexed by integers

$$\frac{\Gamma \vdash E : \text{obj}_x(\text{array}(t)) \quad \Gamma \vdash F : \text{obj}(p) \quad p \in \{\text{int8}, \text{int16}, \text{int32}\}}{\Gamma \vdash E[F] : \text{obj}_x(t)}$$

B.4.5 Inheritance

Classes can implement interfaces, while interfaces can extend other interfaces. In the future we may allow classes to extend classes, so these rules are written with that in mind.

Extension and implementation induce a subtype relation on classes and interfaces as follows

- Inheritance by extension

$$\frac{\begin{array}{l} \Theta(c) = \lambda \bar{g} \cdot \text{clint}_{ci}(\Delta, \bar{t}, \bar{u}) \\ \exists t \in \bar{t} \cdot d \langle \bar{b} \rangle = t[\bar{g} := \bar{a}] \end{array}}{c \langle \bar{a} \rangle <: d \langle \bar{b} \rangle}$$

- Inheritance by implementation

$$\frac{\begin{array}{l} \Theta(c) = \lambda \bar{g} \cdot \text{clint}_{ci}(\Delta, \bar{t}, \bar{u}) \\ \exists u \in \bar{u} \cdot d \langle \bar{b} \rangle \in u[\bar{g} := \bar{a}] \end{array}}{c \langle \bar{a} \rangle <: d \langle \bar{b} \rangle}$$

Furthermore, a type variable is a subtype of its bound

$$\overline{(\alpha \text{ bounded by } k) <: k}$$

As noted earlier, subtyping is reflexive and transitive.

B.4.6 Fields and methods

A field can be found in an object that implements an interface or class that declares the field. The same rule serves for method lookup. Fields and methods may also

be inherited. Rules on consistency of inheritance (see section [[TBD]]) ensure that a field or method can only be inherited from one supertype and that there is no conflict between the declarations of a type and any of its supertypes.

$$\begin{array}{c}
\Gamma \vdash E : \text{obj}_{rw}(t) \\
t <: x \langle \bar{a} \rangle \\
\Theta(x) = \lambda \bar{g} \cdot \text{clint}_{ci}(\Delta, \bar{u}, \bar{v}) \\
\Delta(i) = (\text{public}, om) \\
\hline
\Gamma \vdash E.i : om[\bar{g} := \bar{a}]
\end{array}$$

B.4.7 Initialization Expressions

A new object can be created from a concrete class

$$\begin{array}{c}
\Gamma \vdash E : c \langle \bar{a} \rangle \\
\text{[Matching constructor arguments is To Be Done.]} \\
\hline
\Gamma \vdash \text{new } E(F_0, F_1, \dots, F_{n-1}) : \text{obj}_w(c \langle \bar{a} \rangle)
\end{array}$$

A new array can be created using a for loop.

$$\frac{\Gamma \vdash E : \text{obj}(t) \quad t <: \text{int} \langle \rangle \quad \Gamma_{t \leftarrow \text{obj}_r(q)} \vdash F : \text{obj}(t)}{\Gamma \vdash (\text{for } i : E \text{ do } F) : \text{obj}_w(\text{array}(t))}$$

It is required that E be a compile time constant, evaluable after generic specialization.

This requirement is not captured formally by this rule.

A choice of initializations is given by an ‘if’ expression

$$\frac{\Gamma \vdash E : \text{obj}(\text{bool}) \quad \Gamma \vdash F : \text{obj}(t) \quad \Gamma \vdash G : \text{obj}(t)}{\Gamma \vdash (\text{if } E \text{ then } F \text{ else } G) : \text{obj}_w(t)}$$

Other initializations are simply expressions and are typed the same as other expressions.

B.5 Type checking types

Some of the phrases in a program represent types.

B.5.1 Primitives

Each primitive type is typed to itself

$$\frac{p \in \{\text{bool}, \text{int8}, \text{int16}, \text{int32}, \text{float16}, \text{float32}, \text{float64}\}}{\Gamma \vdash p : p}$$

B.5.2 Class and interfaces

In the abstract syntax, class names are followed by 0 or more generic arguments in braces. (In the concrete syntax, the braces are omitted in the 0 argument case.)

Calculating the type of a phrase $x \{E_0, E_1, \dots, E_{n-1}\}$ is done in several steps

- Look up identifier x in the context. It should map to a class identifier, c .
- Look up that class identifier in the class environment. This gives a lambda expression, which should have n generic parameters.
- Calculate the type of each phrase E_i giving a type a_i .
- Check that each argument type a_i matches the corresponding generic argument.

- The resulting class type is $c \langle a_0, a_1, \dots, a_{n-1} \rangle$.

$$\begin{aligned} \Gamma(x) &= c \\ \Theta(c) &= \lambda \bar{\alpha} <: \bar{t} \cdot \text{clint}_{c_i}(\Delta, \bar{u}, \bar{v}) \\ \Gamma \vdash E_i &: a_i, \text{ for all } i \\ a_i <: t_i[\bar{\alpha} := \bar{a}], &\text{ for all } i \\ \hline \Gamma \vdash x \{ \bar{E} \} &: c \langle \bar{a} \rangle \end{aligned}$$

B.5.3 Array types

Phrases representing array types include a bound. This bound must be a compile time constant calculable after generic expansion. Our rule here does not capture that requirement, as it can only be determined at or after specialization

$$\frac{\Gamma \vdash E : t \quad \Gamma \vdash F : \text{obj}(u) \quad u <: \text{int} \langle \rangle}{\Gamma \vdash E[F] : \text{array}(t)}$$

B.5.4 Generic parameters

Inside a generic class or interface the parameters' identifiers will be bound—in the context—to generic parameters of the form

$$\alpha \text{ boundedby } k$$

B.6 Type checking of commands

For statements, I'll use judgements of the form

$$\Gamma \vdash E$$

where E is a command, to mean that E is well typed. We can think of this as an abbreviation for $\Gamma \vdash E : \text{comm}$, where comm is the type of commands.

B.6.1 Assignments

Assignments are permitted only for primitive variables. Thus the rule is

$$\frac{\Gamma \vdash E : \text{obj}_w(t) \quad \Gamma \vdash F : \text{obj}(u) \quad u <: t \quad t <: \text{primitive} \langle \rangle}{\Gamma \vdash E := F}$$

B.6.2 Local variable declaration

Local variables may be of any object type

$$\frac{\Gamma \vdash E : t \quad \Gamma \vdash F : \text{obj}(u) \quad u <: t \quad \Gamma_{i \leftarrow \text{obj}_w(t)} \vdash S}{\Gamma \vdash \text{obj } i : E := F S}$$

For local variables, the type, if omitted, is inferred from the type of the expression.

$$\frac{\Gamma \vdash F : \text{obj}(t) \quad \Gamma_{i \leftarrow \text{obj}_w(t)} \vdash S}{\Gamma \vdash \text{obj } i := F S}$$

B.6.3 Blocks

A block is a sequence of 0 or more statements.

$$\frac{\Gamma \vdash S_i, \text{ for all } i \in \{0, 1, \dots, n-1\}}{\Gamma \vdash S_0 S_1 \dots S_{n-1}}$$

B.6.4 Method calls

TBD

B.6.5 Sequential control flow

$$\frac{\Gamma \vdash E : \text{obj}(\text{bool}) \quad \Gamma \vdash S \quad \Gamma \vdash T}{\Gamma \vdash (\text{if } E \text{ then } S \text{ else } T)}$$

$$\frac{\Gamma \vdash E : \text{obj}(\text{bool}) \quad \Gamma \vdash S}{\Gamma \vdash (\text{wh } E \text{ do } S)}$$

$$\frac{\Gamma \vdash E : \text{obj}(t) \quad t <: \text{int} \langle \rangle \quad \Gamma_{i \leftarrow \text{obj}_r(t)} \vdash S}{\Gamma \vdash (\text{for } i : E \text{ do } S)}$$

B.6.6 Parallelism

$$\frac{\Gamma \vdash S_i, \text{ for all } i \in \{0, 1, \dots, n-1\}}{\Gamma \vdash (\text{co } S_0 \parallel S_1 \parallel \dots \parallel S_{n-1})}$$

$$\frac{\Gamma \vdash E : \text{obj}(t) \quad t <: \text{int} \langle \rangle \quad \Gamma_{i \leftarrow \text{obj}_r(t)} \vdash S}{\Gamma \vdash (\text{co } i : E \text{ do } S)}$$

B.6.7 Method Implementation

TBD

B.6.8 Atomicity

$$\frac{\Gamma \vdash S}{\Gamma \vdash (\text{atomic } S)}$$

B.7 Type Checking Declarations

B.7.1 Class declarations

B.7.2 Interface declarations

B.7.3 Global object and field declarations

B.7.4 Method declarations

Bibliography

- [1] K. Compton and S. Hauck. Reconfigurable computing: a survey of systems and software. *ACM Comput. Surv.*, 34(2):171–210, 2002.
- [2] S. A. Edwards. The challenges of hardware synthesis from C-like languages. In *DATE '05: Proceedings of the Conference on Design, Automation and Test in Europe*, pages 66–67, Washington, DC, USA, 2005. IEEE Computer Society.
- [3] S. Guccione, D. Levi, and P. Sundararajan. Jbits: A Java-based interface for reconfigurable computing. *X Inc*, 1999.
- [4] R. Hartenstein. Coarse grain reconfigurable architecture (embedded tutorial). In *ASP-DAC '01: Proceedings of the 2001 conference on Asia South Pacific design automation*, pages 564–570, New York, NY, USA, 2001. ACM.
- [5] K. Jensen. An introduction to the theoretical aspects of coloured petri nets. In *A Decade of Concurrency, Reflections and Perspectives, REX School/Symposium*, pages 230–272, London, UK, 1994. Springer-Verlag.
- [6] K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 1, Basic Concepts*. Springer-Verlag, 1997.

- [7] T. Kambe, A. Yamada, K. Nishida, K. Okada, M. Ohnishi, A. Kay, P. Boca, V. Zammit, and T. Nomura. A c-based synthesis system, bach, and its application (invited talk). In *ASP-DAC '01: Proceedings of the 2001 conference on Asia South Pacific design automation*, pages 151–155, New York, NY, USA, 2001. ACM.
- [8] T. S. Norvell. Language design for CGRA project. design 5. [unpublished draft]. 2008.
- [9] T. S. Norvell. The static semantics of HARPO/L [unpublished draft]. 2008.
- [10] M. Odersky and P. Wadler. Pizza into Java: translating theory into practice. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 146–159, New York, NY, USA, 1997. ACM.
- [11] J. L. Peterson. Petri Nets. *ACM Comput. Surv.*, 9(3):223–252, 1977.
- [12] J. Rose, A. El Gamal, and A. Sangiovanni-Vincentelli. Architecture of Field-Programmable Gate Arrays. *PROCEEDINGS- IEEE*, Volume: 81, Issue: 7:1013–1029, 1993.
- [13] J. L. A. van de Snepscheut. Trace Theory and VLSI design. *PhD Thesis, Department of Computer Science, Eindhoven University of Technology, Eindhoven, The Netherlands*, 1983.

- [14] K. Wakabayashi. C-based synthesis experiences with a behavior synthesizer, “Cyber”. In *DATE '99: Proceedings of the conference on Design, automation and test in Europe*, pages 390–393, New York, NY, USA, 1999. ACM.
- [15] D. Zhang. Intermediate representation for parallel languages on CGRAs. Master’s thesis, Memorial University of Newfoundland, December 2007.



