UNIVERSITY OF
CAMBRIDGE

Formalising, improving, and reusing the Java Module System

Rok Strniša
St. John's College

This dissertation is submitted for the degree of Doctor of Philosophy

Abstract

Java has no module system. Its *packages* only subdivide the class namespace, allowing only a very limited form of component-level information hiding. A Java Community Process has started designing the Java Module System, a module system for Java 7, the next version of Java. The extensive draft of the design is written only in natural language documents, which inevitably contain many ambiguities.

We design and formalise LJAM, a core of the module system. Where the informal documents are complete, we follow them closely; elsewhere, we make reasonable choices. We define the syntax, the type system, and the operational semantics of this core, defining these rigorously in the Isabelle/HOL automated proof assistant. We highlight the underlying design decisions, and discuss several alternatives and their benefits.

Through analysis of our formalisation, we identify two major deficiencies of the module system: (a) its class resolution is unintuitive, insufficiently expressive, and fragile against incremental interface evolution; and (b) only a single instance of each module is permitted, which forces sharing of data and types, and so makes it difficult to reason about module invariants. We propose modest changes to the module language, and to the semantics of the class resolution, which together allow the module system to handle more scenarios in a clean and predictable manner. To develop confidence, both theoretical and practical, in our proposals, we (a) formalise in Ott the improved module system, iJAM, (b) prove in Isabelle/HOL mechanised type soundness results, and (c) give a proof-of-concept implementation in Java that closely follows the formalisation.

Both of the formalisations, LJAM and iJAM, are based on Lightweight Java (LJ), our minimal imperative fragment of Java. LJ has been a good base language, allowing a high reuse of the definitions and proof scripts, which made it possible to carry out this development relatively quickly, on the timescale of the language evolution process.

Finally, we develop a module system for Thorn, an emerging, Java-like language aimed at distributed environments. We find that local aliasing and module-prefixed type references remove the need for boundary renaming, and that in the presence of multiple module instances, care is required to avoid ambiguities at de-serialisation.

We conclude with a high-level overview of the interactions between the desired properties and the language features considered, and discuss possible future directions.

Declaration

This thesis:

- is the result of my own work and includes nothing which is the outcome of work done in collaboration except where specifically indicated in the text;
- is not substantially the same as any that I have submitted for a degree or diploma or other qualification at any other university; and
- does not exceed the prescribed limit of 60,000 words.

Rok Strniša
May 2010

Acknowledgements

First, I would like to thank my family for all their moral and financial support throughout my studies. Without them, achieving a Ph.D. at University of Cambridge would have been much harder, if not impossible.

Peter Sewell and Matthew Parkinson, my Ph.D. supervisors, have provided me with all the attention, knowledge and advice a Ph.D. student could ask for. They helped me craft the big picture, while often also giving key insights and ideas for solving specific problems I faced.

I thank Jan Vitek, Doug Lea, Alex Buckley and Stanley Ho for introducing me to the topic of module systems for Java, for providing me with many documents related to the topic, and for extensive discussion.

The Thorn team, whom I have had the pleasure of collaborating with, has been most helpful with the design of the draft module system for the language. At the time, the team included John Field, Jan Vitek, Tobias Wrigstad, Johan Östlund, Bard Bloom, Nate Nystrom, and Gregor Richards.

I also thank Sriram Srinivasan, Silvia Breu, Tom Ridge, Alisdair Wren, Viktor Vafeiadis, Nobuko Yoshida, Simon Peyton Jones, John Billings, Sam Staton, and Jat Singh for useful comments on this work.

I would like to thank my examiners, Gavin Bierman and Sophia Drossopoulou, for thorough analysis of my work, and for many valuable comments and suggestions.

Finally, I acknowledge funding from two EPSRC grants: GR/T11715/01 and DTA-RG44132.

Contents

List of Figures

List of symbols

General presentation rules for symbols

Syntax	Description
$\overline{X}$	$X_1..X_n$ for some n.
$\overline{X_k}^k$	$X_1..X_k..X_n$ for some n.
X^{c}	Not yet annotated (compile-time) code for later annotated (runtime) X.
X_{opt}	Result of a lookup function that can return either X or $\mathbf{null}$.
$X^{\perp}_{opt}$	Like X_{opt}, except that the function can also abort (by returning $\perp$).

Terminals, non-terminals, and meta-variables

Syntax	Description	Language[1]	Page(s)
Γ	type environment	L, J, I	68
θ	variable mapping	L, J, I	67
π	method type, short for '$\overline{\tau} \rightarrow \tau$'	L, J, I	73
τ	***see*** $Type$	L, J, I	——
ϕ	repository cache	J, I	92, 123
a	administrator action	J, I	92
ali	name alias	T	142
am	access modifier	J, I	90
amn	abstract module name	I	122
$\mathbf{bootstrap\_r}$	bootstrap repository's name	J, I	92
br	boundary renaming	I	122
C	class name	L, J, I	59
cl	***see*** C	L, J, I	——
cld	class definition	L, J, I, T	59, 90
cn	class name	T	142
$config$	configuration	L, J, I	61

continues on next page

[1] Abbreviation key: [L ↦ LJ ; J ↦ LJAM ; I ↦ iJAM ; T ↦ Thorn]

continued from previous page

$\mathbf{core\_m}$	name of the core module definition	J, I	90
$ctxcld$	short for '$(ctx,\ cld)$'	L, J, I	62
ctx	context	L, J, I	60, 89
dcl	name of derived class	L, J, I	59
def	definition	T	146
def_or_mc	definition or module construct	T	146
$entity$	entity	T	146
$Exception$	exception	L, J, I	61
f	***see*** $Field$	L, J, I	——
fd	field declaration	L, J, I, T	59
$Field$	field name	L, J, I	59
file	source file	T	146
fqn	fully-qualified name (of a class)	L, J, I	59, 90
ga	module-level, generic arguments	T	144
H	heap	L, J, I	61
id	non-fully-qualified name	T	142
imp	import statement	I, T	122, 146
imp_dep	import dependency	I	123
inc	optional include of import	T	145
$\mathbf{include}$	annotation to include import	T	145
j	index variable	L, J, I	59
k	***see*** j	L, J, I, T	——
L	variable state	L, J, I	61
l	***see*** j	L, J, I	——
m	module name (cannot be $\mathbf{core\_m}$)	J, I	90
mc	module construct	T	141
md^{c}	module definition	J, I	92, 123
md	module instance	J, I	92, 100, 123
mem	membership declaration in a module file	T	141
$Method$	method name	L, J, I	59
$meth$	***see*** $Method$	L, J, I, T	59, 142
$meth\_body$	method body	L, J, I	59
$meth\_def$	method definition	L, J, I, T	59
$meth\_sig$	method signature	L, J, I	59
mf	module file	J, I	90, 121
MH	module hierarchy	J, I	92, 123
mhv	module hierarchy value	J, I	122

continues on next page

continued from previous page

mi	module instance identifier	J, I	90
$mibr$	associated boundary renaming	I	123
mn	module name	J, I, T	90, 142
m_ali	module name aliasing	T	143
m_loc	module location override	T	141
$name$	possibly fully-qualified name	T	142
nn	natural number	L, J, I	63
NPE	null-pointer exception	L, J, I	61
null	invalid pointer value	L, J, I	61
`Object`	top class	L, J, I	67
oid	***see*** $Pointer$	L, J, I	——
own	parameter to create a separate instance	T	143
P	program	L, J, I	59, 92
pd	package declaration	J, I	90
pn	package name	J, I	90
$Pointer$	object identifier	L, J, I	61
private	access modifier for module members	T	143
public	access modifier for module members	J, I, T	90, 143
RC	repository context	J, I	92
rep	import's replication parameter	T	143
$repl$	replication modifier	I	122
replicating	replication policy annotation	I	122
rn	repository name (can be **bootstrap_r**)	J, I	92
r	repository name	J, I	91
R	repository	J, I	92
req	replication parameter	T	143
s	statement	L, J, I	59
singleton	replication policy annotation	I	122
SRC	compile-time class definitions	J, I	90
this	reference to currently executing object	L, J, I	60
$TVar$	term variable	L, J, I	59
$Type$	type	L, J, I	68
URI	uniform resource identifier (URI)	T	141
URL	uniform resource locator (URL)	T	141
v	***see*** Val	L, J, I	——
va	module's value annotation	T	144
Val	value	L, J, I	61

continues on next page

continued from previous page

value	indicates a value module	T	144
var	***see*** Var	L, J, I	——
Var	term variable	L, J, I	59
vd	variable declaration	L, J, I	59
vis	visibility declaration for a name	T	143
vn	variable name	T	142
w	***see*** Val	L, J, I	——
x	***see*** $TVar$	L, J, I	——
y	***see*** $TVar$	L, J, I	——

Judgements and functions

Syntax/Description	Language[2]	Page(s)
$(MH,\ \overline{mi},\ nn) \in \mathbf{reachable}$ there are nn module instances reachable from $\overline{mi}$ in MH	J	96
$(P,\ ctx,\ cl,\ nn) \in \mathbf{path\_length}$ the length of the inheritance path for cl is nn	L, J, I	63
$(P,\ mi,\ P') \in \mathbf{wf\_P\_change}$ well-formed program change (proof related)	J, I	108
$\vdash_{ctx} meth\_def^{\mathsf{c}} \rightsquigarrow meth\_def$ context insertion for a method definition	J, I	175
$\vdash_{ctx} s^{\mathsf{c}} \rightsquigarrow s$ context insertion for a statement	J, I	175
$\vdash_{mi} cld^{\mathsf{c}} \rightsquigarrow cld$ context insertion for a class definition	J, I	175
$\vdash_{mi} md^{\mathsf{c}} \rightsquigarrow md$ module definition translation (instantiation)	J, I	175
$\vdash\ P$ well-formed program	L, J, I	73, 104
$\Gamma \vdash\ config$ well-formed configuration	L, J, I	77
$\theta \vdash\ s \rightsquigarrow\ s'$ variable translation for a statement	L, J, I	68
$config \longrightarrow\ config'$ reduction of a statement	L, J, I	65
$config \xrightarrow{a} config'$ reduction of an administrative action	J, I	99
$config \xrightarrow{\overline{ia}} config'$ reduction of internal actions	J, I	100, 126

[2]Abbreviation key: [L ↦ LJ ; J ↦ LJAM ; I ↦ iJAM]

continues on next page

continued from previous page

$P \vdash \tau$ valid type	L, J, I	69
$P \vdash \tau \prec \tau'$ subtyping	L, J, I	69, 101
$P \vdash H$ well-formed heap	L, J, I	77
$P \vdash R$ well-formed repository	J, I	104
$P \vdash cld$ well-formed class	L	74
$P \vdash_\tau meth\_def$ well-formed method in τ	L, J, I	75
$P \vdash_{ctx} (dcl, cl, \overline{fd}, \overline{meth\_def})$ well-formed class in ctx (generic rule)	L, J, I	74
$P \vdash_{mi} cld$ well-formed class in mi	J, I	105
$P \vdash_{mi} md$ well-formed module instance	J, I	105, 128
$P, \Gamma \vdash s$ well-formed statement	L, J, I	76
$P, \Gamma, H \vdash L$ well-formed variable state	L, J, I	77
$P, H \vdash v_{opt} \prec \tau_{opt}$ well-formed value	L, J, I	78
$MH \vdash \phi$ well-formed repository cache	J, I	104, 128
$MH \vdash RC$ well-formed repository context	J, I	104
$RC \vdash MH$ well-formed module hierarchy	J, I	105, 128
$\mathbf{acyclic\_clds}\ P$ the class inheritance hierarchy in P is acyclic	L, J, I	63, 97
$\mathbf{acyclic\_clds}_{mi} P$ the class inheritance hierarchy in P is acyclic (starting at mi)	J, I	97
$\mathbf{acyclic\_mh}\ MH$ a module hierarchy is acyclic	J, I	96
$\mathbf{class\_fields}\ (cld) = \overline{fd}$ extract field declarations from a class	L, J, I	64
$\mathbf{class\_methods}\ (cld) = \overline{meth\_def}$ extract method definitions from a class	L, J, I	72
$\mathbf{class\_name}\ (cld) = dcl$ extract the class name from a class	L, J, I	69
$\mathbf{distinct\_fqns}\ (\overline{cld})$ fully-qualified names are distinct	J, I	95

continues on next page

continued from previous page

$\mathbf{distinct\_names}(P)$ names of type definitions are distinct	L	74
$\mathbf{fields}\,(P,\,\tau) = \overline{f}_{\,opt}$ fields lookup in type τ	L, J, I	64
$\mathbf{fields\_in\_path}\,(\overline{ctxcld}) = \overline{f}$ fields lookup in a class path	L, J, I	64
$\mathbf{find\_cld}\,(P,\,ctx,\,fqn) = ctxcld_{opt}$ class lookup	L, J, I	62, 93, 123
$\mathbf{find\_cld\_in\_core}\,(P,\,fqn) = ctxcld_{opt}$ class lookup in the core library module	J, I	93
$\mathbf{find\_cld\_in\_imports}\,(MH,\,\overline{mibr},\,fqn) = ctxcld_{opt}$ class lookup in the imports	I	124
$\mathbf{find\_cld\_in\_imports}(MH, \overline{mis}, fqn) = ctxcld_{opt}$ class lookup in the imports	J	94
$\mathbf{find\_cld\_in\_module}\,(\overline{cld},\,fqn) = cld_{opt}$ class lookup in an import	J, I	94
$\mathbf{find\_cld\_in\_self}\,(\overline{cld},\,pn,\,fqn) = cld_{opt}$ class lookup in the same module	J, I	94
$\mathbf{find\_md}\,(RC,\,rn,\,mn) = rnmd^{\mathsf{c}}_{opt}$ module definition lookup	J, I	98
$\mathbf{find\_md\_in\_mds}\,(\overline{md^{\mathsf{c}}},\,mn) = md^{\mathsf{c}}_{opt}$ module definition lookup in a list	J, I	97
$\mathbf{find\_md\_rec}\,(RC,\,rn_1,\,mn,\,nn) = rnmd^{\mathsf{c}}_{opt}$ module definition lookup (recursive part)	J, I	97
$\mathbf{find\_meth\_def}\,(P,\,\tau,\,meth) = ctxmeth\_def_{opt}$ method definition lookup in type τ	L, J, I	64
$\mathbf{find\_meth\_def\_in\_list}\,(\overline{meth\_def},\,meth) = meth\_def_{opt}$ method definition lookup in a list	L, J, I	64
$\mathbf{find\_meth\_def\_in\_path}\,(\overline{ctxcld},\,meth) = ctxmeth\_def_{opt}$ method definition lookup in a path	L, J, I	64
$\mathbf{find\_path}\,(P,\,ctx,\,cl) = \overline{ctxcld}_{opt}$ class path lookup with a class name	L, J, I	63
$\mathbf{find\_path}\,(P,\,\tau) = \overline{ctxcld}_{opt}$ class path lookup with a type	L, J, I	63
$\mathbf{find\_path\_rec}\,(P,\,ctx,\,cl,\,\overline{ctxcld}) = \overline{ctxcld}_{opt}$ class path lookup (recursive part)	L, J, I	62
$\mathbf{find\_type}\,(P,\,ctx,\,cl) = \tau_{opt}$ type lookup	L, J, I	62
$\mathbf{ftype}\,(P,\,\tau,\,f) = \tau'$ field type lookup in a type	L, J, I	72
$\mathbf{ftype\_in\_fds}\,(P,\,ctx,\,\overline{fd},\,f) = \tau^{\perp}_{opt}$ field type lookup in a list	L, J, I	73
$\mathbf{ftype\_in\_path}\,(P,\,\overline{ctxcld},\,f) = \tau_{opt}$ field type lookup in a path	L, J, I	72

continues on next page

continued from previous page

$\mathbf{full\_name}\,(cld) = fqn$ extract the full name of a class	J, I	102
$\mathbf{imp\_br}\,(imp) = br$ extract boundary renaming from an import statement	I	127
$\mathbf{imp\_dep\_of}\,(md^{\mathsf{c}},\ mi,\ imp) = imp\_dep$ generate import dependency from md^{c}, mi and imp	I	126
$\mathbf{imp\_name}\,(imp) = m$ extract the module name from an import statement	I	126
$\mathbf{lift\_opts}(\overline{\tau_{opt}}) = \overline{\tau}_{opt}$ lift a list of option types	L, J, I	73
$\mathbf{md\_name}\,(md^{\mathsf{c}}) = mn$ extract the module name from a module definition	J, I	99
$\mathbf{mds\_rm}\,(\overline{md^{\mathsf{c}}}_1,\ md^{\mathsf{c}}) = \overline{md^{\mathsf{c}}}_2$ remove a module definition from a list	J, I	99
$\mathbf{method\_name}\,(meth\_def) = meth$ extract the method name from a method definition	L, J, I	74
$\mathbf{methods}\,(P,\ \tau) = \overline{meth}$ method names lookup in τ	L, J, I	72
$\mathbf{methods\_in\_path}\,(\overline{cld}) = \overline{meth}$ method names lookup in class path	L, J, I	72
$\mathbf{mtype}\,(P,\ \tau,\ meth) = \pi$ method type lookup	L, J, I	73
$\mathbf{no\_core\_renaming}\ P$ there is no renaming of core classes in P	I	130
$\mathbf{no\_core\_renaming\_in\_mibrs}\,(P,\ \overline{mibr})$ there is no renaming of core classes in $\overline{mibr}$	I	130
$\mathbf{package\_name}\,(cld) = pn$ extract the package name from a class	J, I	93
$\mathbf{R\_body}\,(R) = (\overline{md^{\mathsf{c}}},\ \phi)$ extract repository's contents	J, I	99
$\mathbf{R\_name}\,(R) = rn$ extract repository's name	J, I	104
$\mathbf{R\_update}\,(R,\ \overline{md^{\mathsf{c}}},\ \phi) = R'$ update a repository with given contents	J, I	99
$\mathbf{superclass\_name}\,(cld) = cl$ extract the name of the superclass from a class	L, J, I	62

1

Introduction

Division of labour increases both the quantity and the quality of goods produced or services provided. Within a modern large corporation, layers of management are required to select, organise, and lead highly specialised individuals in order to achieve a goal.

Similarly in computer software, large applications are typically built in several parts, each performing a highly specific function. Programming languages used for writing these applications normally provide a system that helps with the production, distribution, linking, and execution of these parts. Such a system is commonly known as a *module system*, while a single part is usually referred to as a *module*.

The modern understanding of a software module appeared with the concept of *information hiding* introduced by Parnas in 1972 [10, p. 398]. The main idea is that modules should not be based on a flowchart, i.e. a chart describing the control flow of a software program, but rather in a way that hides from the other modules the design decisions that are likely to change. With information hiding, a modular design can minimise redundancy, and maximise opportunities for reuse [40].

Initially, the idea of information hiding was applied only on the source level, i.e. for isolating and hiding bits of code that are likely to change. The concept of *abstract data types*, introduced by Guttag in 1975 [10, p. 444], applies this idea to runtime objects. The term 'abstract data type' refers to a class (or a group) of objects defined by a representation independent specification. That is, one is concerned about what can be done with an object (implementing some abstract data type), not how the various operations are implemented.

Most general-purpose programming languages today are either *imperative* (e.g. C [44], Java [19], C# [33]) or *functional* (e.g. SML [34], OCaml [24], Haskell [41]). The latter treat computation as evaluation of mathematical functions, and tend to avoid state and mutable data, i.e. most functions are guaranteed to produce the same result when called twice with the same arguments. This does not hold for the former, since they describe computation with a sequence of statements that manipulate the program state.

Many of the current functional languages, such as SML and OCaml, use a variation of the *ML module system* [30]. In these languages, a module's interface is normally defined explicitly through a *module signature*, which can abstract away from the details of the module's implementation. Then, we can safely replace (without modifying any of the *client modules*, i.e. modules that use this module) a module with any other module that *implements* the same interface. Module interfaces allow *separate compilation*, i.e. each module is compiled separately, after which the resulting program fragments are *linked* together. Module interfaces can also be *abstract*; that is, they can abstract away from the representation of the data being exchanged, promoting even looser coupling among modules. Furthermore, ML-style module systems support *functors*, functions that can generate modules, which provide developers with further opportunities for code reuse [22].

Object-oriented languages [5] are a sub-group of the imperative languages that are based around *objects*, runtime embodiments of state and related functionality. In these languages, *classes* are the main encapsulation mechanism; a class closely corresponds to the idea of an abstract data type, since it describes all the *instances* (objects) of this class. The key features are *encapsulation* (hiding parts of the implementation through the use of accessibility constraints on both the state and the functionality of a class), class *inheritance* (defining a child class in terms of its parent), and *polymorphism* (an object behaving according to the definition of its class even when used as an instance of an ancestor class). These languages normally support *incremental, cut-off compilation* [22, §8.3], i.e. compilation where dependencies need to be present, but where the already-compiled ones do not need to be re-compiled. This compilation scheme is well-suited for the common cyclic dependencies among classes; conversely, most ML module systems do not permit *recursive modules* [13], which simplifies the type system and allows a linear initialisation order consistent with dependencies among the modules. Classes are normally compiled into separate files, which are linked dynamically (by the runtime environment) — while more flexible than the ML's link-then-execute approach, dynamic linking requires post-linking checks [28] (that can fail at runtime) to ensure that classes are used correctly.

Even though a class can define *inner classes*, i.e. classes defined within a class, this approach does not scale well: large groups of top-level classes are often required to implement a single high-level component. Most object-oriented languages lack a high-level module system that provides a way to encapsulate, package, distribute, deploy, link, and execute such components as versionable units.

In the following section (§1.1), we show how Java [19] (version 6), currently the most popular general-purpose programming language [47], fails to satisfy some of the basic properties of a high-level module system, and describe what its designers are proposing for its next release. We then state our thesis (§ 1.3), outline our contributions and the structure of this document (§1.4), and overview the related work (Chapter 2).

1.1 The Java Module System

Currently, the Java programming language [19] has no (high-level) module system. Its *packages* are a filesystem-based mechanism, which enables only the subdivision of type namespaces, and a very limited form of component-level information hiding.

More specifically, a package can hide a class from classes in other packages. However, a package cannot semantically *contain* another package, and any package can import any other; therefore, any class can access any other non-hidden (public) class. There is also no built-in support for package-level generics, versioning, type renaming, or multiple (package) instances — all these concepts are described in §1.2.

Two Java Community Processes, JSR-277 [51] and JSR-294 [52], are developing the Java Module System [54] (JMS[1]), a module system for Java 7, the next version of Java [53]. It is aimed at bringing component-level information hiding (§1.1.2) to the language, while simultaneously removing the need for complex systems of classloaders when avoiding JAR hell (§ 1.1.3). It tries to do this in a way that (1) is easy to understand, (2) makes its features available automatically, (3) is backward and forward bytecode-compatible[2] with the existing Java programs (on the new Java Virtual Machines), and (4) does not modify the syntax of the underlying language.

We performed a careful analysis of the draft documents, i.e. the two Java Service Requests (JSRs) [51, 52], which together exceed two hundred pages of natural language specification. More specifically, we formalise (i) the syntax for module files, (ii) the semantics of the administrator actions (installing, un-installing, and initialising module definitions), and (iii) class resolution, which searches for class definitions across package, module, and repository boundaries. However, our formalisation excludes versions, custom import policies (custom code that initialises module definitions, and links together the resulting module instances), and support for legacy files.

In this section, we introduce the main ideas behind the module system, as we see them, while Chapter 4 presents the syntax of its core, the associated operational semantics, and more.

[1] There is currently no official abbreviation for "Java Module System." JAM, used in our previous work, now refers to a test implementation of the system. In this document, we use JMS to refer to the specification.

[2] A newer system is *backward bytecode-compatible* when it can deal with bytecode for an older system. An older system is *forward bytecode-compatible* when it can deal with bytecode for a newer system.

1.1.1 The WebCalendar example

Suppose we want to run a servlet that provides calendar services on our web server, in existing Java. Our code (the `webcalendar` package) uses two third-party software packages: the XSLT library (the `xslt` package), and the servlet engine (the `engine` package). Both of these require an XML parser (the `xml.parser` package): the first to create HTML from XML calendar data, and the second for its configuration files. Furthermore, the servlet engine uses a cache (the `engine.cache` package) to avoid overhead. Figure 1.1 shows the dependency relation between the Java packages in our system.

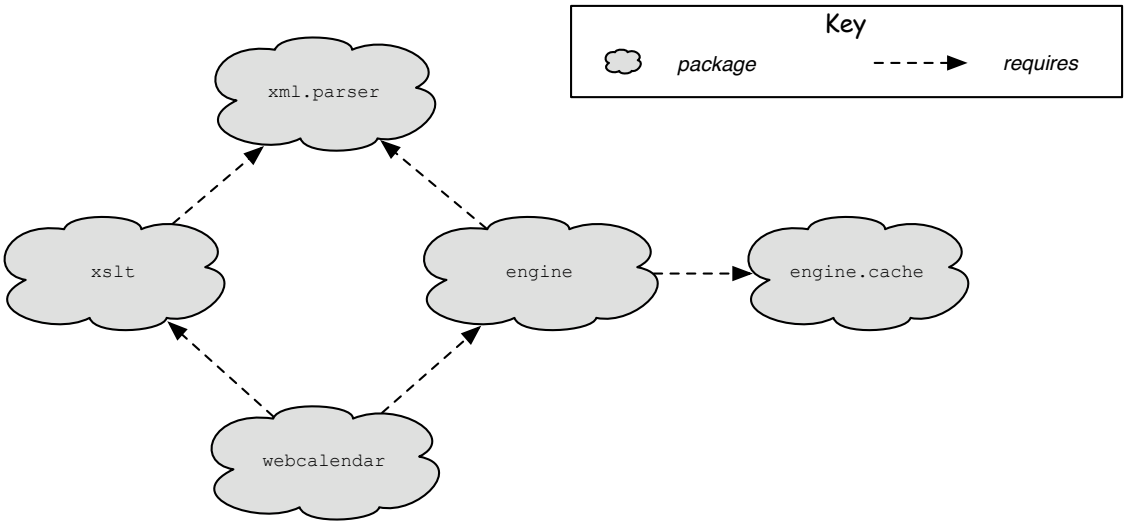


Figure 1.1: Java packages in our example, and dependencies among them

Currently, all classes in our system have access to the public classes in any Java package, including `engine.cache`. Since the cache is logically part of the servlet engine, we would like to make `engine.cache`'s public interface (its public classes) visible only to the engine, i.e. to classes in package `engine`, and not also to the rest of the system.

A way of tackling this is to make `engine.cache`'s public interface package-private instead, and then combine the contents of both `engine` and `engine.cache` into a single package; however, the approach loses structure, and makes package-private interfaces of the original packages visible to all classes in the new, combined package.

1.1.2 Component-level information hiding

A better approach is to put both packages (`engine` and `engine.cache`) in a bigger structure, which acts (almost) like a black box, so that, e.g., `webcalendar`, which is outside this structure, cannot see its contents. This kind of a bigger structure is known as a *module definition*; the term *superpackage* is also used. We refer to the module definition holding the two packages as *ServletEngine*.

The module definitions can also selectively leak parts of the interfaces of their contents — we refer to this as *selective exporting*. For example, suppose there is a public class `Servlet` in the `engine` package (within *ServletEngine*), and another public class `CalendarServlet` in the `webcalendar` package (outside *ServletEngine*). If we want to make `Servlet` visible to `CalendarServlet`, *ServletEngine* has to explicitly *export* `Servlet`, i.e. to explicitly reveal the bound name `engine.Servlet`.

Since module definitions are an abstraction layer above Java packages, we have to put the `webcalendar` package inside its own module definition, *WebCalendar*, which then *imports* the *ServletEngine* module definition, making `engine.Servlet` visible to the whole `webcalendar` package. See Fig. 1.2 for the software design of our example, now with module definitions. In JMS, a module definition cannot contain another; this allows, for example, that both *XSLT* and *ServletEngine* import the same *XMLParser*.

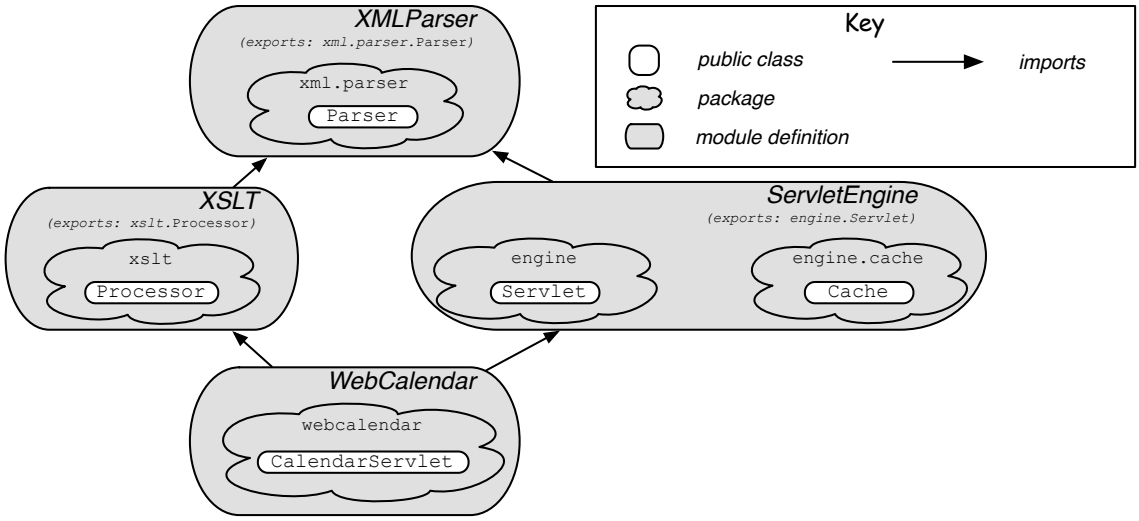


Figure 1.2: The example with module definitions

In general, importing a module definition makes its exported classes visible to the members of the importer, while its non-exported classes, e.g. `Cache` in *ServletEngine*, remain invisible to the importers. Classes made available by the imports can be re-exported — we refer to this feature as *optional re-exporting*.

The programmer defines a module definition through a module file. Each module file specifies the module's name, member Java packages, imported modules, exported classes, and various other properties, e.g. a version. Figure 1.3 shows the example's source code.

1.1.3 Dealing with JAR hell

In Java, a class is loaded into the runtime by a *classloader*. Normally, it is loaded from a class file on the local file system, but it can also be loaded from elsewhere, or generated by

```
                        module XMLParser {
                          member xml.parser;
                          export xml.parser.Parser; }

module XSLT {                          module ServletEngine {
  member xslt;                           member engine; member engine.cache;
  import XMLParser;                      import XMLParser;
  export xslt.Processor; }                export engine.Servlet; }

                        module WebCalendar {
                          member webcalendar;
                          import ServletEngine; import XSLT; }
```

Figure 1.3: The module files defining the example's module definitions

reflection, and then possibly post-processed/checked before use [29]. A classloader maintains a map from fully-qualified names of classes[3] to their definitions, which means that these names have to be distinct for a particular classloader. Classes loaded or generated by different classloaders have incompatible types — this makes sense, since the definitions of the classes can be different even if their names match, and since they do not share their static state. By default, classes are loaded by the *system classloader*.

We can easily have multiple versions of a class available to the system; however, a classloader can load just one of these. When dependencies between classes are non-trivial, one part of the system might require the classloader to load a certain group of classes, while another part requires a different group of classes that have identical names. When a classloader is told to load a class whose name matches the name of an already loaded class, it simply ignores the command — in large programs, this normally leads to program failures at seemingly random points during execution. Since groups of classes are normally distributed in the compressed format called JAR (Java ARchive), this phenomenon is known as *JAR hell*.

In our example, this can happen if *XSLT* and *ServletEngine* require different versions of *XMLParser*. This means that the system classloader would need to load two versions of all `xml.parser` classes, which is not possible with a single classloader due to the above-mentioned restrictions. Undesirable behaviour occurs either for *XSLT* or *ServletEngine*, depending on which version of *XMLParser* is loaded first.

As mentioned above, each Java classloader creates its own class namespace. Therefore, we can hold two different versions of a class through two custom classloaders in the same Java runtime. If the same class is loaded by two different classloaders, two distinct types will be created, each with its own copy of the static data.

A classloader can contain arbitrary code for generating or loading classes at runtime; it can even delegate its resolution to another classloader. Such systems of classloaders

[3] A fully-qualified class name includes the name of a Java package that a class belongs to.

are incredibly expressive, but also quickly become hard to understand, use, and debug — the most common argument against using classloaders directly. For this reason, all module systems for Java try to encapsulate classloader expressivity behind a user-friendly interface, and the Java Module System is no exception.

A *module instance* is a runtime instance of a module definition. Each module instance creates its own (1) types for entities defined within the corresponding module definition, and (2) copy of any static data within those entities. It is a classloader in disguise.[4]

Therefore, JMS allows same-named classes in the system as long as they are within different module instances. Figure 1.4 shows a solution to the above-described JAR hell problem: there is a module instance for each of the two versions of the *XMLParser* module definition. The *XSLT* module instance is linked to a different module instance of *XMLParser* than *ServletEngine*. Since the linking of module instances coordinates the classloading procedure, *XSLT*'s classes cannot access classes of *ServletEngine*'s *XMLParser*, and vice versa. The details of the module system, e.g. module initialisation and the class resolution algorithm, are described in Chapter 4.

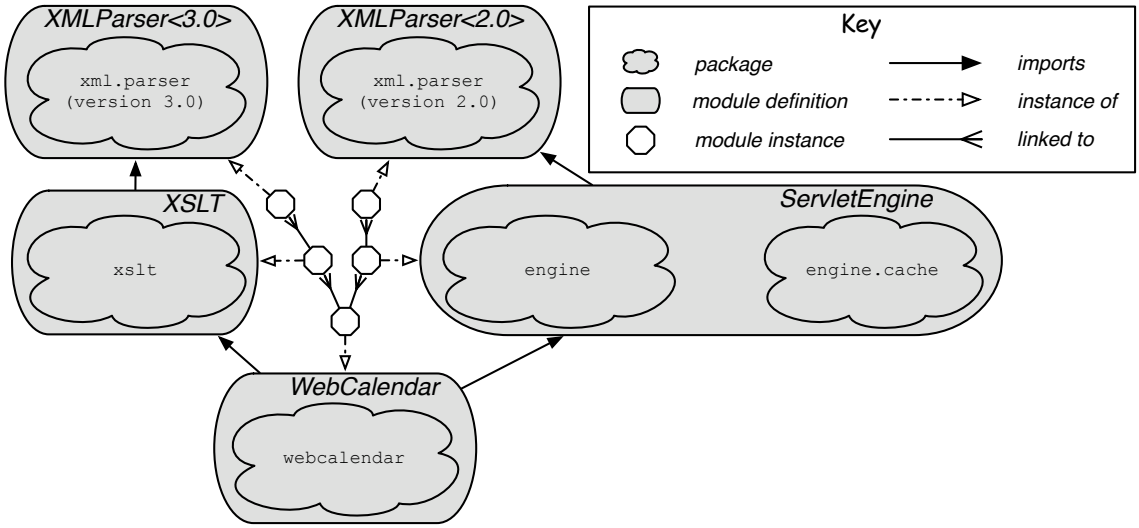


Figure 1.4: The solution to the example JAR hell problem

1.1.4 Using the module system

The module system does not change the underlying language: a developer writes the standard Java source files as before — this makes the source files forward and backward bytecode compatible. He also writes module files (as in §1.1.2), which define the module membership, the imports, the exports, and various other properties.

[4]Note that developers can still define their own classloaders — control over such custom classloading is done through Java's Security API. This is discussed further in Chapter 7.

A module file is compiled into a module definition: compilation replaces each package name in the module file with the corresponding class definitions. Module definitions are then (distributed and) installed in a Java runtime environment. Specifically, they are installed into a *repository*, a runtime concept, through which an administrator of the system can search for, install, uninstall, and initialise module definitions. There can be many repositories at runtime to further control the dependency and isolation between different module definitions. Repositories are organised into hierarchies with the *bootstrap repository* as its root. The bootstrap repository always contains the *core platform* module definition, which in turn contains all the classes within the core Java platform.

As in the case of JAR files, developers can specify the *main class* of module definitions, i.e. the class whose `main` method will get executed when a module definition is "executed." In fact, JAR files are forward-compatible with module definitions, which are not backward compatible (although JAR files can be embedded within module definitions).

When the user "executes" a module definition, e.g. *WebCalendar*, the module system creates and links module instances of *WebCalendar* and of all module definitions it (recursively) depends on, i.e. *XMLParser*, *XSLT*, and *ServletEngine*. Then, the execution begins in the main class of the module instance originally executed.

A module definition can hold various properties, e.g. a version. An importer can request a module definition that satisfies certain property constraints. The module system already supports this for the usual versioning scheme; however, it also allows custom property annotations on module definitions, and custom import code. With the latter, the developer can write Java code that is responsible for selecting and initialising imported module definitions, and linking together the resulting module instances. While more expressive, custom import code is not guaranteed to terminate.

1.1.5 A short summary of JMS's features

We have seen through examples that the Java Module System (JMS) can bring two long-sought-for features to Java: better component-level information hiding, and versioning.

As described in the previous subsections, a module system administrator installs module definitions into repositories. Depending on the relation between repositories, module definitions in one repository will either all be accessible or all be inaccessible to module definitions in another repository. Also, if all module definitions were installed in a single repository, every module definition could import any other.

Java packages already support selective exporting. In addition to this, JMS modules support optional re-exporting. In this thesis, we show that optional re-exporting has little value if the module visibility relation is transitive (explained in § 1.2) as in JMS, and develop ways to restrict this property for both module instances and module definitions to achieve better information hiding.

JAR hell (§1.1.3) is a serious problem for large Java applications. JMS introduces versioning, which will, together with a smart use of classloaders underneath, provide an easy solution to most classloading problems.

It is not hard to draw an analogy between concepts in the underlying Java language and those in the Java Module System. For example, module files can be seen as Java source files at the module level, module definitions as class files, and module instances as objects. The main difference is that the modules operate on a higher level than classes, on a level more suitable for packaging, distribution, and versioning. Finally, JMS modules instances are not first-class members: once they are linked, they play no role except in the (deterministic) semantics of classloading.

1.2 Desirable properties of a module system

There are *many* desirable properties of a module system; however, the two most important are *expressivity* and *scalability*. The former states how many types of scenarios it can handle, while the latter specifies how easy it is to manage a large and/or growing system.

Scalability depends on how *intuitive* the semantics of a module system is, how *reusable* and *robust* its modules are, and whether or not it supports *separate compilation* — all of these sub-properties/features depend on how *localised* the influence of a single module is, i.e. how far the names and policies of a single module necessarily propagate (recursively) to its imports and its clients. *Versioning* is another important step towards robustness, whereas operations of module *composition*, the ability to specify a module in *multiple files*, and the ability to *share runtime state* among multiple programs, all improve reusability. Furthermore, *module instantiation policies*, a good *name resolution algorithm*, and various *information hiding* techniques contribute to a better localisation of a module's influence. The name resolution algorithm is influenced by the presence of *module-prefixed type references* and *renaming* (includes renaming *of exports by exporters*, *of exports by importers*, and *of imports*). Finally, information hiding is based on *selective exporting*, and improved by *optional re-exporting* and *non-transitive module visibility*.

Expressivity heavily depends on *parametricity* and the name resolution algorithm. It is improved by various, specific language features, such as *recursive modules*, *first-class modules*, and *sub-modules*. Parametricity is often provided through *functors/generics*, which depend on module interfaces, which in turn depend on module interfaces.

As described above, these properties and the related language features are heavily interconnected. In Fig. 1.5, we try to illustrate these connections in a directed graph, where the general meaning of an arrow is "contributes to," while the boldness of an arrow represents our opinion of the relative importance.

In this thesis, we focus on localising the influence of a single module through inform-

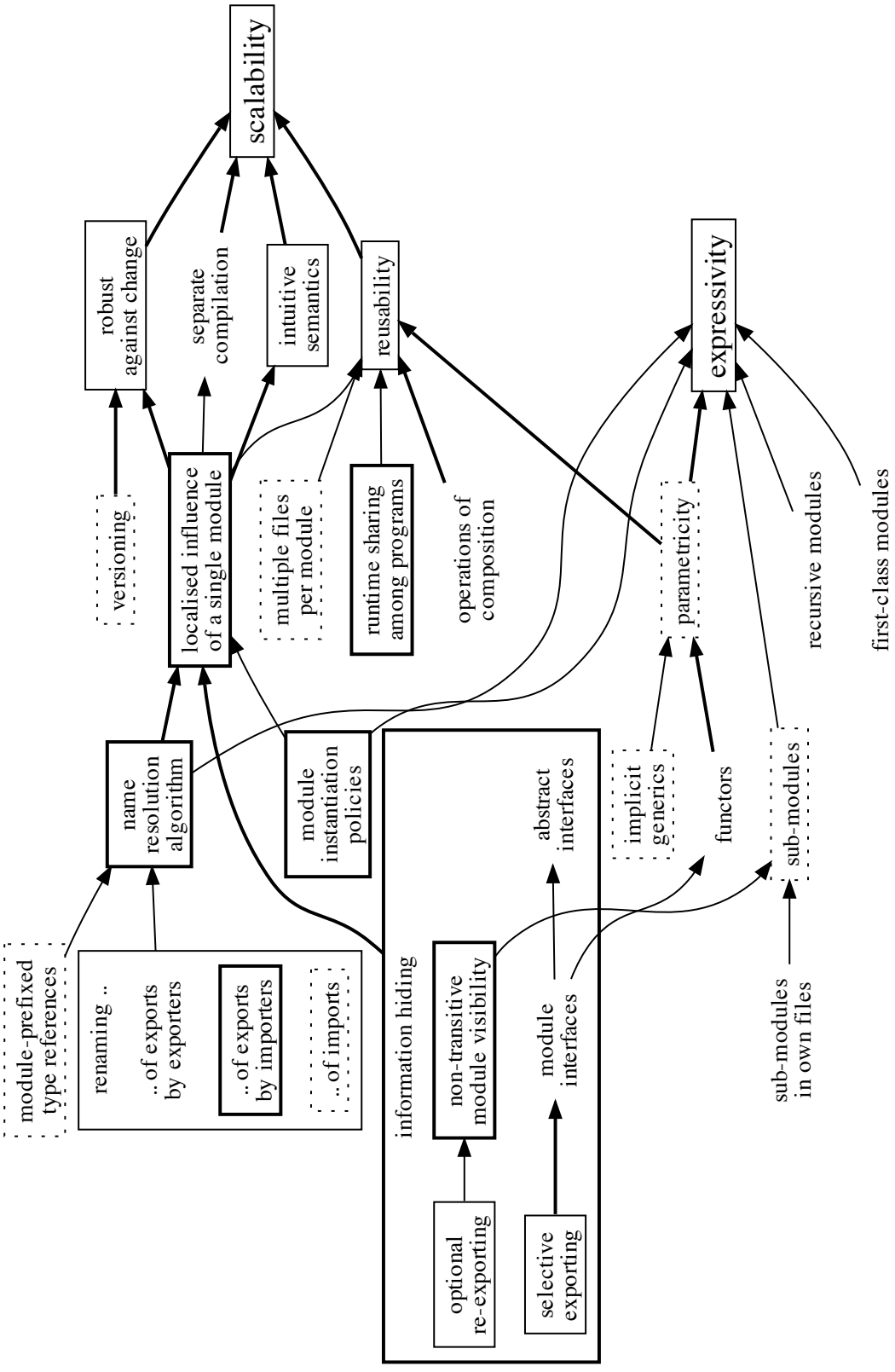


Figure 1.5: The connections among the desired properties/features of a module system. *(The boldness of a box represents how involved this thesis is with a particular property/feature, while the boldness of an arrow represents our opinion of its relative contribution.)*

ation hiding techniques, name resolution algorithms, and module instantiation policies. We then show how this leads to more robust modules and a more intuitive semantics. We also research the expressivity of the different name resolution algorithms in the presence of module instantiation policies. Finally, we briefly discuss the importance of module-prefixed type references and sub-modules. In Fig. 1.5, the boldness of a border represents how involved this thesis is with a particular property/feature.

Following are the brief descriptions of the non-trivial properties/features mentioned above that also appear in Fig. 1.5 and are referenced throughout this thesis:

first-class modules Modules that can be treated as values in the underlying language.

parametricity The module system supports functions that produce modules.

localised influence of a module This property expresses how far the names and policies of a single module necessarily propagate (recursively) to its imports and its clients.

module-prefixed type references Type references can refer to specific modules.

multiple files per module A module can be defined in several files.

module instantiation policies The developer can specify when new instances of a module definition are created, and when they are shared.

name resolution algorithm Defines how a type's definition is found given a type name.

non-transitive module visibility If module *A* can see module *B*, and module *B* can see module *C*, *A* does not necessarily see *C*.

operations of composition Well-defined and normally type-safe procedures of combining parts of a module (or whole modules) into larger parts (or whole modules).

optional re-exporting A module can export (non-hidden) members of imported modules.

recursive modules Modules that permit cycles in the module import graph.

renaming of exports by exporters The internal and the external names for a type of a module can differ.

renaming of exports by importers An imported type can be bound under a different name within the client module.

renaming of imports The code within the client module can refer to an imported module under a different name.

runtime sharing among programs A type's static data can be shared among programs.

selective exporting A module can hide any of its members.

separate compilation Ability to compile a module definition in isolation, or with only the public interfaces of its imports available.

sub-modules A module can semantically contain another module.

sub-modules defined in own files A module and its sub-module can be defined in different files.

versioning Modules can be versioned, and imported according to their versions.

1.3 Thesis

Although the Java Module System is a great improvement over Java packages, there is much left to be desired. We claim that:

- the lack of explicit module interfaces together with the parent-then-self name resolution (discussed in Chapter 5) leads to poor support for localisation of a module's influence and for code reuse; and
- the absence of module-prefixed references (due to backward compatibility) results in fragile and inexpressive semantics for the end user.

Once a software product, e.g. programming language or a software module, is released, its clients usually become heavily dependent on its definition. If such a definition contains errors, the best fix likely breaks much of the clients' code. Since this is highly undesirable, the developers of such a product usually try to find a non-optimal, but a backward-compatible, solution, or simply let clients find a work-around.

A well-designed module system of a programming language can relatively easily limit the dependencies of clients on a software module, but it cannot eliminate them in general. Capturing a part of a programming language definition within a module, on the other hand, is substantially more difficult, since parts of a language often rely on each other's subtle details. It is well known that software development for both applications and programming languages is (at least at the moment) significantly less costly than software maintenance [17]. Therefore, it is only logical to put more effort into preventing errors of a software product *before* its release.

Recently, tools for formalising and verifying programming language definitions have improved substantially: from software tools that help you write the definition [46], to programs that help you state and verify the proofs of language properties [1–4]. Most modern programming languages and their specific features, however, are still defined in natural language documents with hardly any or no formalisation. We believe that a rigorous formalisation of a programming language (and its add-ons):

- gives a valuable insight into the details of the semantics;
- promotes a precise discussion of the definition;
- allows important properties to be proven about the language;
- can find subtle errors and potential problems early (before release); and
- is cost-effective, and can be done on the same timescale as the industrial design and standardisation process.

Finally, we believe that formalisations of language definitions and their proof scripts can be modularised, and the 'language modules' reused.

1.4 Contribution

The draft documents describing the design of the Java Module System [51, 52] are written solely in natural language, which means they inevitably contain many ambiguities. We formalise the core of this module system: whenever possible, we follow the informal description closely; for the remainder, we try to capture what we believe is the intended semantics. We call this formalisation the Lightweight Java Module System (LJAM).

We identify and discuss two key deficiencies of the module system: (a) its class resolution is unintuitive, insufficiently expressive, and fragile against incremental interface evolution; and (b) a highly inflexible module instantiation that makes it difficult to reason about module invariants. We then develop and precisely define clean solutions that (i) make class resolution intuitive and flexible (through class renaming and an adapted resolution algorithm), and (ii) allow developers to control the sharing of module instances (through custom policies). The solutions are modelled on top of LJAM, producing the Improved Java Module System (iJAM). As a proof of concept, we also implement a module system on top of Java, which can follow the semantics of either LJAM or iJAM.

LJAM and iJAM share a common base: Lightweight Java (LJ), a minimal imperative core of Java. Initially, we designed LJ as a base language for the two formalisations. Through this process, LJ has been abstracted to the point where we think it can be used for experimentation. In fact, LJ has already been used by others to formalise "features" in LFJ [14].

All our formalisations are expressed rigorously: with the help of Ott [46], we define their syntax, type system and operational semantics, producing (from a single source) both human-readable typeset rules and machine-processed mathematics for the Isabelle/HOL proof assistant [3]. We also mechanically prove type soundness for all our formalisms.

Finally, we develop a draft module system for Thorn, an emerging Java-like language for distributed computing. Here, we examine the effects of (i) module-prefixed type references on namespace localisation, and (ii) multiple module instances on de-serialisation.

More specifically, our contributions are:

- the design and formalisation of LJ (Chapter 3);
- a lightweight introduction to the Java Module System (§1.1);
- a detailed formalisation of the module system, producing LJAM (Chapter 4);
- identification of its two major deficiencies (Chapter 5);
- precise and clean solutions to them (Chapter 5);
- formalisation of these solutions, producing iJAM (Chapter 6);
- Isabelle/HOL type soundness proofs for LJ (§3.6), LJAM (§4.6), and iJAM (§6.5);
- an implementation that can model both LJAM and iJAM (Chapter 7); and
- a draft design of a module system for Thorn (Chapter 8).

While the following chapter is devoted to related work (Chapter 2), the rest of the thesis is structured as follows:

Chapter 3 When reasoning about language features, it is a good idea to limit the underlying language to avoid dealing with unnecessary details. However, a key concept of the language – state – even though present in the first formal model of Java [16], is absent from the most widely used formal model [23]. This chapter presents LJ, a minimal *imperative* core of Java, which we build upon in the later chapters.
Note: *If the reader is eager to start with modules, we suggest at least skimming through LJ's syntax (Fig. 3.1 and Fig. 3.2), and LJ's definition for a type (*§3.4.1*), and then skipping to the start of Chapter 4.*

Chapter 4 The draft of the Java Module System's design is written in natural language documents, which inevitably contain many ambiguities. The chapter describes our formalisation of the core of the module system on top of LJ. The formalisation, named the Lightweight Java Module System, creates an essential basis for precise discussion of the key design decisions.

Chapter 5 Through analysis of LJAM, we identify two major deficiencies of the module system: (a) its class resolution is unintuitive, insufficiently expressive, and fragile against incremental interface evolution; and (b) only a single instance of each module is permitted, which forces sharing of data and types, and so makes it difficult to reason about module invariants. We propose modest changes to the module language, and to the semantics of the class resolution, which together allow the module system to handle more scenarios in a clean and predictable manner.

Chapter 6 To develop theoretical confidence in our informal proposals for improvement of the module system, we formalise them in an extension of LJAM, named iJAM, and mechanically show that type soundness is preserved.

Chapter 7 To show that our proposals can be put into practice, we give a proof-of-concept implementation that follows most of iJAM's semantics with modules that can contain *any* Java code. The implementation is also able to enter a compatibility mode, where it follows the LJAM's semantics, instead.

Chapter 8 We develop a module system for Thorn, where we find that module-prefixed type references remove the need for boundary renaming, and that in the presence of multiple module instances, ambiguities can arise during de-serialisation.

Chapter 9 Finally, we give a high-level overview of the interactions between the desired properties and the language features considered, and discuss possible future work.

1.5 Collaboration

The work on LJ (Chapter 3) and LJAM (Chapter 4) was published at OOPSLA'07 [50]:

> The Java Module System: Core Design and Semantic Definition
> Rok Strniša, Peter Sewell, and Matthew J. Parkinson.

The work on iJAM (Chapter 5 and Chapter 6) and the implementation (Chapter 7) appeared at FTfJP'08 (a satellite workshop of ECOOP'08) [48]:

> Fixing the Java Module System, in Theory and in Practice
> Rok Strniša.

The work on Thorn (Chapter 8) was accepted to OOPSLA'09 [8]:

> Thorn—Robust, Concurrent, Extensible Scripting on the JVM
> Bard Bloom, John Field, Nathaniel Nystrom, Johan Östlund, Gregor Richards,
> Rok Strniša, Jan Vitek and Tobias Wrigstad.

1.6 Preliminaries

All syntax definitions, semantic rules, and language terms in this document are checked by Ott [46]; the tool also generates their LaTeX. Keywords are printed in **bold**, meta-variables in *italic*, and constants in `typewriter` font. The over-bars indicate lists, e.g. $\overline{cl}$ is a list of cl's. Non-trivial terms and propositions are surrounded with single quotation marks or brackets. There is also a list of symbols (page 15), and an index (end of the document).

Unless specified otherwise, all lemmas and theorems in this document are taken directly from the Isabelle/HOL [3] scripts — variables not bound explicitly (within a lemma or a theorem) are universally bound implicitly. Similarly, the proofs in the document are natural language versions of the corresponding, mechanically-verified proofs written in Isabelle/HOL. In the appendix, there is a segment of the Isabelle/HOL script.

All our semantic rules are defined as inductive relations in Ott. Some of these relations, e.g. the class resolution, are functions, and are easier to deal with in a theorem prover if expressed as such. For those, we wrote Isabelle/HOL functions, and proved equivalence (in Isabelle/HOL) between the Ott-generated relations and the corresponding functions. Since the original relations are quite verbose, we put those in the appendix. In the main text, we present the more compact Isabelle/HOL functions using an OCaml-like syntax [24], while using Ott to generate LaTeX for language terms — this makes the functions compact and readable, and their presentation consistent with syntax definitions and semantics rules.

We highlight larger changes in semantic rules, lemmas, and theorems from one language to the next. Added formalisms are displayed normally (not highlighted), while unchanged ones are mentioned and skipped.

1.6.1 A brief introduction to Ott

Ott [46] is a tool for writing definitions of programming languages and calculi. It takes as input a definition of a language syntax and semantics, in a concise and readable ASCII notation that is close to what one would write in informal mathematics. It generates LaTeX to build a typeset version of the definition. Furthermore, it can output Coq [1], HOL [2], and/or Isabelle [3] versions of the definition; in this thesis, we generate and use Isabelle definitions only.

In Ott, we can define structures with BNF meta-syntax. That is, each production of a meta-term defines a data constructor for that meta-type. Ott then generates a data type in Isabelle/HOL for each such meta-type.

n	::=		natural number — Ott-generated data type
	|	0	data constuctor ‘zero’
	|	$\mathbf{Succ}\ n$	data constructor ‘successor’

While the BNF notation is normally used only for constructive definitions, e.g. n, we can use BNF in Ott to define an *abstract meta-type*, e.g. nn below. The underlying representation of an abstract meta-type (in our formalisations) is an Isabelle/HOL type, while its interface are the corresponding *meta-productions* (indicated by the letter M). The corresponding Isabelle/HOL type, e.g. nat, is always displayed in brackets within the main comment of each such meta-term, e.g. nn. The meta-productions correspond to Isabelle/HOL functions,[5] which return a meta-term of the appropriate meta-type — note that, for this reason, we cannot specify non-meta-productions for an abstract meta-type.

nn	::=			natural number — Isabelle/HOL type (nat)
	|	0	M	Isabelle/HOL literal ‘0’
	|	1	M	Isabelle/HOL literal ‘1’
	|	$nn + nn'$	M	uses Isabelle/HOL’s ‘+’

However, meta-productions can be specified for a non-abstract meta-type, e.g. n.

n	::=			natural number — Ott-generated data type
	|	0		data constuctor ‘zero’
	|	$\mathbf{Succ}\ n$		data constructor ‘successor’
	|	$n + n'$	M	uses our Isabelle/HOL function for addition of n’s

In our formalisations, we use nn for representing natural numbers. Note that we only need to write meta-productions for the natural numbers that we use within our semantic rules. Another important rule to note here is that, in Ott, the name of a meta-variable,

[5] The Isabelle/HOL functions that implement the meta-productions are not shown in Ott’s LaTeX output.

e.g. nn, must be identical to the name of its meta-type; however, a meta-variable can include a number, an index variable, and/or prime symbols, e.g. nn_2''.

As noted before, $\overline{cl}$ in Ott stands for a list of cl. Furthermore, it is possible to refer to 'some element within the list,' and indirectly to 'the length of the list.' For example, $\overline{cl_k}^{\,k}$ is short for $\overline{cl_k}^{\,k\in 1..n}$, or $cl_1..cl_k..cl_n$, or $cl_1..cl_n$, for some (unspecified) length n. The k in the subscript implies that cl_k is referring to the k-th element in the list, while the k in the superscript (next to the overline) states that this list ranges over all elements, i.e. where $k = 1..n$. The index variables, e.g. k, are bound per semantic rule, which means that specifying both $\overline{cl_k}^{\,k}$ and $\overline{var_k}^{\,k}$ within the same semantic rule implies that $\overline{cl_k}^{\,k}$ and $\overline{var_k}^{\,k}$ are of the same (unspecified) length, and that $\overline{cl_k\ var_k}^{\,k}$ is an element-wise pairing of the two lists. Using the same philosophy, $\overline{\overline{cl_k\ var_j}^{\,k}}^{\,j}$ is a shorthand for $\overline{cl_k\ var_1}^{\,k}..\overline{cl_k\ var_j}^{\,k}..\overline{cl_k\ var_m}^{\,k}$ for some m. Note that an index variable also specifies the identity of a list, which implies that $\overline{cl_k}^{\,k}$ is not automatically unifiable with $\overline{cl_j}^{\,j}$ or $\overline{cl}$. Finally, the two dots in $cl_1..cl_n$ imply that the list could be empty, i.e. $n \in \mathbb{N}$, while the three dots in $cl_1...cl_n$ imply that the list has at least one element, i.e. $n \in \mathbb{N}^+$.

A good analogy to Ott's list notation (described above) is the sum notation in mathematics. The table below displays a couple of comparisons.

	Full expression	**Standard**	**Short**
Math	$x_1 + .. + x_i + .. + x_m$	$\sum_{i=1}^{m} x_i$	$\sum_{i} x_i$
Ott	$x_1..x_i..x_m$	$\overline{x_i}^{\,i\in 1..m}$	$\overline{x_i}^{\,i}$
Math	$\sum_{i=1}^{m} y_1 x_i + .. + \sum_{i=1}^{m} y_j x_i + .. + \sum_{i=1}^{m} y_n x_i$	$\sum_{j=1}^{n}\sum_{i=1}^{m} y_j x_i$	$\sum_{j}\sum_{i} y_j x_i$
Ott	$\overline{y_1\ x_i}^{\,i\in 1..m}..\overline{y_j\ x_i}^{\,i\in 1..m}..\overline{y_n\ x_i}^{\,i\in 1..m}$	$\overline{\overline{y_j\ x_i}^{\,i\in 1..m}}^{\,j\in 1..n}$	$\overline{\overline{y_j\ x_i}^{\,i}}^{\,j}$

1.6.2 A brief introduction to Isabelle/HOL

Isabelle [3] is a generic proof assistant. It allows mathematical formulae to be expressed in a formal language and provides tools for proving those formulae in a logical calculus.

Isabelle/HOL is a specialisation of Isabelle for Higher Order Logic (HOL), providing a higher-order logic theorem proving environment for large applications. The user of Isabelle/HOL writes *theories*, which are collections of declarations, definitions, and proofs. Both declarations and definitions highly resemble declarations and definitions in OCaml, respectively; however, each function defined within Isabelle must provably terminate.

Proofs or lemmas within Isabelle/HOL have the following form:

```
proof/lemma name: "proposition"
  proof steps
done
```

When compiling a proof, Isabelle/HOL verifies that *proof steps* reduce *proposition* to `True`. The *proof steps* can involve built-in axioms, various definitions, already compiled lemmas and theorems, and tactics (algorithms for applying axioms, definitions, lemmas and theorems). Note that the variables not bound explicitly within the *proposition* are universally bound implicitly.

As an example, we include in Appendix B our Isabelle/HOL script that proves the progress property of LJ (Chapter 3). The rest of the proof script can be found online as indicated in the following chapters.

2

Related work

This thesis is concerned with the design and the formalisation of programming languages and their features, focusing on module systems for object-oriented languages. While there are currently only a few other mechanically-verified language formalisms, module system designs are numerous. We overview both of these categories in the following sections.

2.1 Verified language formalisms

No programming language used widely in the industry, e.g. Java [19] or C# [33], has a complete formalisation. SML [34] is currently the only general-purpose language widely used in academia that also has a complete formalisation, and has been mechanically proven sound [27]. Type soundness has also been shown for large subsets of Java [25, 35, 55], and OCaml [39]. There has also been substantial work on formalising Scheme [31], C [37], and C++ templates [56].

There are a few academic formalisations trying to capture the core of Java. The most popular of these is Featherweight Java (FJ) [23]. As part of this thesis, we develop our own simplified model of Java, Lightweight Java (LJ), and briefly compare it to FJ (Chapter 3).

During the course of this research, LJ has already been used by others to develop Lightweight Feature Java (LFJ) [14]. Type-soundness of LFJ was proven by mechanically showing that any well-formed LFJ program can be translated to a well-formed LJ program.

2.2 Module systems

A lot of research has been done in this field, and there are *many* existing module systems, both commercial and academic. For this reason, we chose to overview only a few module systems that are most related to the topic of this thesis, and that together cover a wide variety of technologies. Furthermore, our analyses focus mainly on the most relevant properties/features (§1.2). We start with a short overview of JMS, continue with the analyses, and conclude with a table showing what module system has which property/feature.

2.2.1 A short overview of JMS

The Java Module System (JMS) is a prototype module system that is planned to be integrated into Java 7. In JMS, developers write module files, which are then compiled together with the appropriate classes into module definitions. The administrator installs these module definitions into repositories — the relation among the repositories determines what a given module definition can import. The initialisation of an installed module definition creates its module instance, obtains the module instances of its imports, and links them together. At this point, the Java code within a module instance is executable. Note that once a module instance has been linked to its imports, the links cannot be modified, i.e. initialisation sets up fixed classloaders that perform deterministic name resolution during execution. The underlying Java code is not "aware" of the module system.

JMS supports selective exporting, optional re-exporting, and its module visibility is transitive (see §1.1.5). It features versions, allows modules to be defined in multiple files (class and module file), and enables sharing of static data among multiple programs — as described in Chapter 5, static data sharing among programs is enforced. However, JMS does not support any form of renaming, module instantiation policies, sub-modules, functors, or module-prefixed type references (due to backward-compatibility). Chapter 4 contains a more involved overview of the system.

2.2.2 OSGi

OSGi [38] is currently the most widespread module framework for Java, and the largest competitor to JMS. Its *bundles* (module definitions) are, in fact, JAR (Java archive) files.

Bundles specify individual classes or packages as members, packages or bundles as imports, and packages as exports. The *import package directive*, OSGi's preferred way to import, specifies *what* package the client wants, not *who* has the package he wants.

OSGi promotes *service-oriented programming*, i.e. the "publish-find-bind" model for services, which are provided by the bundles registered within the global registry. A bundle is registered together with a key-value map that stores custom properties; this map can be used by custom constraints when looking up 'a service.'

Furthermore, event-based handling is used for bundles and services — each bundle is responsible for its own publication, discovery, binding, and adapting to changes in runtime. For example, if a service stops while another service is using it, the latter service breaks if its event handler does not explicitly re-bind to another suitable service.

As an example, we define a simple spellchecker as an OSGi service — this is an adapted version of the example by Richard Hall [21]. First, we define the interface:

```
1  package spellchecker.service;
2  public interface SpellcheckerService {
3    public boolean checkWord(String word);
4  }
```

We compile it, and put it in a JAR file, `spellchecker.jar`, together with a manifest:

```
1  Manifest-Version: 1.0
2  Bundle-ManifestVersion: 2
3  Bundle-Version: 1.0.0
4  Bundle-SymbolicName: spellchecker
5  Import-Package: org.osgi.framework
6  Export-Package: spellchecker.service
```

Next, consider implementing an English variant of the service. We also specify the code executed at start and stop points of this bundle by implementing `BundleActivator`.

```
1  package spellchecker;
2  import java.util.*;
3  import org.osgi.framework.*;
4  import spellchecker.service.SpellcheckerService;
5
6  public class Activator implements BundleActivator {
7    public void start(BundleContext context) {
8      Properties props = new Properties();
9      props.put("Language", "English");
10     context.registerService(SpellcheckerService.class.getName(),
11                             new SpellcheckerImpl(), props);
12   }
13   public void stop(BundleContext context) {}
14   private static class SpellcheckerImpl
15       implements SpellcheckerService {
16     List<String> dictionary =
17       Arrays.asList(new String[] {"hello", "goodbye"});
18     public boolean checkWord(String word) {
19       return dictionary.contains(word.toLowerCase());
20     }
21   }
22 }
```

Again, we compile the code, and put it in a JAR file, `spell_en.jar`, along with a manifest file (the first three lines are the same as above):

```
4   Bundle-SymbolicName: spell_en
5   Bundle-Activator: spellchecker.Activator
6   Import-Package: org.osgi.framework, spellchecker.service
```

By making small changes to the English service, we also create a French service, and store it within a different JAR file, `spell_fr.jar`.

Finally, we create a client to these services:

```
4  package client;
5  import java.io.*;
6  import org.osgi.framework.*;
7  import spellchecker.service.SpellcheckerService;
8
9  public class Activator implements BundleActivator {
10   public void start(BundleContext context)throws Exception {
11     ServiceReference[] refs = context
12        .getServiceReferences(SpellcheckerService.class.getName(),
13                              "(Language=*)");
14     if (refs != null)
15       try {
16         BufferedReader in =
17           new BufferedReader(new InputStreamReader(System.in));
18         while (true) {
19           System.out.print("Enter word: ");
20           String word = in.readLine();
21           if (word.length() == 0) break;
22           SpellcheckerService spell =
23              (SpellcheckerService) context.getService(refs[0]);
24           if (spell.checkWord(word))
25             System.out.println("Correct.");
26           else
27             System.out.println("Incorrect.");
28           context.ungetService(refs[0]);
29         }
30       } catch (IOException ex) { }
31     else
32       System.out.println("Couldn't find a spellchecker service.");
33   }
34   public void stop(BundleContext context) {}
35 }
```

We then store this client in another JAR file, `client.jar`, together with a manifest file similar to that of the service.

At this point, we can install, start, stop, and uninstall the bundles:

```
1   $ java -jar osgi.jar -console
2   osgi> install file:/programs/osgi/spellchecker.jar
3   Bundle id is 1
4   osgi> install file:/programs/osgi/spell_en.jar
5   Bundle id is 2
6   osgi> install file:/programs/osgi/spell_fr.jar
7   Bundle id is 3
8   osgi> install file:/programs/osgi/client.jar
9   Bundle id is 4
10  osgi> start 1 2 3 4
11  Enter word: hello
12  Correct.
13  Enter word: bonjour
14  Incorrect.
15  Enter word:
16  osgi> stop 4 3 2 1
17  osgi> uninstall 1
18  osgi> start 3 2 4
19  Enter word: hello
20  Incorrect.
21  Enter word: bonjour
22  Correct.
23  Enter word:
24  osgi> close
25  $
```

First, we installed (lines 2–9) and started (line 10) all the bundles. When the client bundle was started, it found the English spellchecker service (lines 11–15). Then, we stopped all the bundles (line 16), and uninstalled the service specification bundle (line 17). Finally, we re-started the spellchecker services in a different order (line 18), and now the client found the French spellchecker (lines 19–23), instead.

There are two interesting points to note here. First, the order in which the bundles were started determines the preference order for their services. Second, an uninstalled bundle remains in the framework for any bundles already bound to it.

Next, we investigate OSGi's name resolution algorithm. Consider three bundles: `a`, `b`, and `c`. All three bundles include a class `p.T` (located in package `p`), which holds a static integer `n` with the initial value of zero. Whenever `a`, `b`, or `c` is started, the value of `p.T.n` is increased by `1`, `10`, and `100`, respectively, before being printed out.

If a and b export package p, while c imports p, we get the following interaction:

```
1  osgi> install file:/programs/osgi/a.jar
2  Bundle id is 1
3  osgi> install file:/programs/osgi/b.jar
4  Bundle id is 2
5  osgi> install file:/programs/osgi/c.jar
6  Bundle id is 3
7  osgi> start 1 2 3
8  bundle a:  p.T.n=1
9  bundle b:  p.T.n=10
10 bundle c:  p.T.n=101
11 osgi> uninstall 3 2 1
12 osgi> install file:/programs/osgi/b.jar
13 Bundle id is 4
14 osgi> install file:/programs/osgi/a.jar
15 Bundle id is 5
16 osgi> install file:/programs/osgi/c.jar
17 Bundle id is 6
18 osgi> start 5 4 6
19 bundle a:  p.T.n=1
20 bundle b:  p.T.n=10
21 bundle c:  p.T.n=110
```

From lines 8–9 above, we see that bundles a and b refer to different instances of package p (otherwise, the lines 9 and 20 would read 11). From line 10, we observe that bundle a shares its instance of p with bundle c. If we re-install the bundles in a different order (lines 11–17), but start them in the same order as before (line 18), bundle b now shares its instance of p (line 21) with c, instead.

To better understand the underlying semantics, we modify the above example a little. Suppose that b now not only exports p, but also imports p. The above interaction produces exactly the same output, except for lines 9 and 10, where the printed values of p.T.n are 11 and 111, respectively. In this case, the framework combined b's import of p to the a's export of p. Interestingly, when the installation order is changed, the framework does not create this link, giving the same output as above. From this, it appears that the bundle installation order determines the *primary exporters of packages*. Therefore, if a bundle is not the primary exporter for a package it contains, the bundle will share the package with its primary exporter only if it also imports the package.

A bundle can also require another bundle (with the Require-Bundle directive in a manifest file); however, importing a package is the OSGi's preferred way of importing. For example, if in the first scenario (a and b export p; c imports p) c also *required* b, then c would still use a's p when a is installed first. However, if both b and c require bundles a and b, respectively, instead of importing package p, the final value printed for both

installation orders is `111`. This implies that the `Require-Bundle` directive, contrary to the `Import-Package` directive, is independent of the installation order.

To summarise, OSGi supports selective exporting, optional re-exporting (only possible when using `Require-Module`), but its module visibility is transitive. Furthermore, the framework supports the `Import-Package` directive, which is included mainly to relieve bundle developers from worrying about what bundles to import; JMS left out this directive, probably because (1) it is not clear how to obtain deterministic and intuitive semantics when ambiguities arise, and (2) clients become highly dependent on their imports to export whole packages. Like JMS, OSGi uses the parent-then-self name resolution scheme, and does not support module instantiation policies, renaming, or functors.

2.2.3 .NET

A .NET assembly [15] is an encapsulation of a collection of language definitions compiled down to .NET bytecode (*Common Intermediate Language*, CIL), stored either as an executable (.exe), or as a DLL library (.dll). An assembly can import other assemblies, and implicitly export types (by declaring them public). .NET also has namespaces, which are used to provide some name structuring and prevent name clashes.

An assembly can be composed from multiple *code modules*. These code modules are not semantic barriers; however, they do provide a way to write an assembly in multiple user languages, optimise downloading of an application by putting seldom-used types in a code module that is downloaded on-demand, and can easily combine code from multiple developers without creating multiple assemblies.

Assembly repositories are file-system-based. The .NET runtime searches for the imported assemblies first within the application's directory, and then within the system-wide repository, *Global Assembly Cache*. To avoid name conflicts of same-named assemblies that have different properties, e.g. version, the Global Assembly Cache is a single .NET folder implemented using nested file-system directories.

For example, we can write some source code into a file, `Hi.cs`.

```
1 namespace HelloNamespace {
2   public class HelloClass {
3     public string hi() {return "hello, world";}
4   }
5 }
```

In a second file, `Client.cs`, we refer to the first.

```
1 using HelloNamespace;
2 class MyClient {
3    public static void Main() {
4       HelloClass myHelloClass = new HelloClass();
5       System.Console.WriteLine(myHelloClass.hi());
6    }
7 }
```

Now, we can either (a) compile them both directly into an executable, `hello.exe`:

```
$ csc /out:hello.exe Client.cs Hi.cs
```

Or, we can (b) create code modules, and link them into an executable:

```
$ csc /t:module Hi.cs
$ csc /addmodule:Hi.netmodule /t:module Client.cs
$ al /main:MyClient.Main /t:exe /out:hello.exe \
       Hi.netmodule Client.netmodule
```

Finally, we can (c) create a DLL library, and an executable that refers to it:

```
$ csc /t:library /out:Hi.dll Hi.cs
$ csc /r:Hi.dll /t:exe /out:hello.exe Client.cs
```

Option (a) is quick and simple, but requires re-compilation of all source files; option (b) shows that incremental compilation of an assembly is possible through code modules; and, option (c) compiles against a DLL library that must later be found in the application's directory. A DLL library needs to be signed with a developer-generated key before it can be installed into the Global Assembly Cache.

Suppose `Client.cs` was linked also with a copy of `Hi.dll`, `HiCopy.dll`. Then, the reference to `HelloClass` within `Client.cs` would be ambiguous. To disambiguate the reference to the class within `Hi.dll` without modifying the imports, we can create an *external alias*. First, we replace line 1 in `Client.cs` with:

```
1 extern alias Hello;
2 using HelloClass = Hello::HelloNamespace.HelloClass;
```

Then, we provide the compiler with the appropriate binding (`/r:Hello=Hi.dll`):

```
$ csc /t:library /out:Hi.dll Hi.cs
$ csc /t:library /out:HiCopy.dll Hi.cs
$ csc /r:Hello=Hi.dll /r:HiCopy /t:exe /out:hello.exe Client.cs
```

To summarise, .NET assemblies provide selective exporting (only public classes are visible to client assemblies), but do not allow optional re-exporting, or renaming (only class and namespace aliasing), and its assembly visibility is transitive. The bytecode can

contain only explicit type references:[1] if there are multiple ways of resolving a class reference, an error is thrown at compile-time — such an ambiguity can always be resolved with external aliases described above. The fact that all references must be resolved at compile-time implies that JMS name resolution problems (§5.1) do not apply to .NET; however, this also makes an assembly less versatile than a JMS module definition, since an assembly can be linked only to the assemblies (that have the same key signature as those) it was compiled against. Finally, there is no support for module instantiation policy: each assembly always creates an instance of every assembly it is linked against, while assembly sharing between multiple applications is not supported.

2.2.4 OCaml

Although functional languages are substantially different to object-oriented languages, we feel that comparing JMS to a functional language module system is essential. We chose to compare against OCaml [24]'s module system, since OCaml is currently one of the most popular functional programming languages.

The following example source files demonstrate how OCaml deals with information hiding, renaming, and name resolution. First, we define a string variable, `msg`, in a sub-module, `M`, and a function, `hello`, that prints the value of `msg`. We define both of these in a source file, `A.ml`, that implicitly defines an OCaml module, `A`.

```
module M = struct let msg = "hello, world" end
let hello () = print_endline M.msg
```

To define what `A` does *not* export, we write its interface in a separate file, `A.mli`. The file below implies that `A` does not export its sub-module `M`.

```
var hello : unit -> unit
```

Another module, `B` (defined in `B.ml`), accesses `A`'s exported function `hello` with a module-prefixed name reference. Note that trying to access `A`'s sub-module `M` gives a compile-time error, and that `A`'s `hello` is, in fact, re-exported as `hi`.

```
(* module N = A.M *) (* Compile-time error: A.M hidden *)
let hi = A.hello
```

The following module, `C`, re-exports all names exported by `B` using the OCaml's **`include`** directive.

```
include B
```

The last module, `D`, first declares a variable, `hi`, with value `42`, then puts `C`'s exported names in the current namespace with OCaml's `open` directive, and finally uses `A`'s `hello`

[1] There exists a proposal that lifts this restriction [11].

through the above re-exporting. Note that (1) module renaming is supported, (2) a name binding overrides previous bindings to that name, (3) names are not automatically re-exported, and (4) we can always use A's hello directly; however, non-transitive module visibility for top-level modules, e.g. A, can be achieved through **include** statements, e.g. by including A in another module as a sub-module and restricting its visibility.

```
let hi = 42
module M = C
open M
let _ = hi ()
(* Error: A, B, and C do not automatically re-export hello. *)
(* let _ = hello () *)
```

The following commands compile and run the above modules.

```
$ ocamlc -o hello A.mli A.ml B.ml C.ml D.ml
$ ./hello
hello, world
```

To summarise, OCaml features selective exporting, optional re-exporting, renaming of exports by exporters (through aliases and interfaces), renaming of exports by importers (through aliases), module renaming, non-transitive module visibility, and module-prefixed type references, while a name always resolves to the last corresponding binding. The concept of module instantiation and module instances is not a natural concept for functional programming languages, since these languages tend to avoid state; for this reason, OCaml does not support module instantiation policies.

2.2.5 Jiazzi

Jiazzi [32] is an experimental component framework for Java. It is based on *units* [18], a module system originally designed for Scheme [31]. It does not modify the syntax of the underlying Java, and does, therefore, not support module-prefixed type references.

There are two types of units: *atoms* and *compounds*. The specification of a unit defines (in a text file) which Java packages are imported, and which exported. Every imported or exported package must have a *package signature* ascribed to it. A package signature is a developer-written text file describing a sub-set of the public definitions of a Java package.

For example, suppose we define classes C and D in package p for an atom unit a.

```
package p;
public class C {
  public void hello() {
    System.out.println("a.p.C: " + new D().getMsg());
  }
}
```

```
package p;
public class D {
  public String getMsg() {return "(msg from atom-private a.p.D)";}
}
```

Furthermore, suppose we now define a package signature, p_s, for p, which only exports class C's constructor and hello method:

```
signature p_s = {class C extends Object {C(); void hello();}}
```

Finally, we state in unit a's definition file that a exports p_s's of the Java package p:

```
atom a {export p : p_s;}
```

We have to perform three steps to compile a unit in Jiazzi. First, we generate stub files based on the signatures of the imported packages — this step is unnecessary for a, since it does not import any packages. Second, we compile the Java source files that a unit contains against the generated stub files; Jiazzi requires that we put all files for a unit in its own sub-directory. Third, we perform the linking step, which essentially puts all member class files and any inner units into a JAR file (for a unit a, the file is named a.jar).

```
$ java -jar jiazzi.jar . -stub a
$ javac -sourcepath a -d a -cp a/stubs.jar a/p/C.java
$ javac -sourcepath a -d a -cp a/stubs.jar a/p/D.java
$ java -jar jiazzi.jar . -link a
```

Next, we create a unit, b, which is identical to unit a, except that the strings in classes C and D are "$b.p.C$" and "$b.p.D$" instead of "$a.p.C$" and "$a.p.D$", respectively.

Now, we create a unit, c, that imports and uses a's exports. It contains a single class, r.Main, that creates an instance of p.C, and calls its hello method:

```
package r;
public class Main {
  public static void main(String[] args) {new p.C().hello();}
}
```

The signature of the package r simply makes the Main.main method visible:

```
signature r_s = {class Main extends Object {
  public static void main(String[]); }
}
```

The unit file for the atom c specifies that c imports the Java package p (ascribed with p_s), and exports the Java package r (ascribed with r_s):

```
atom c {import p : p_s; export r : r_s;}
```

We obtain c.jar with the above procedure: generate stubs, compile classes, and link.

Now, we define a compound unit, d, that composes the atom units a and c:

```
1 compound d {
2   export r : r_s;
3 } {
4   link unit a : a, c : c;
5   link package a@p to c@p, c@r to r;
6 }
```

Line 2 above states that d, like c, exports the package r (ascribed with r_s). Line 4 specifies that this compound unit contains units named a and b of same-named unit types (defined above). Finally, line 5 links a's export of p to c's import of p, and c's export of r to this compound's export of r.

Since the compound unit d does not contain any classes, and does not import any packages, we can skip stub generation and source compilation steps. Assuming the package signatures, the unit files, and the previously-generated JAR files are in the same directory, we can perform the linking step, then execute the main method of the r.Main class (available through d's export of r).

```
$ java -jar jiazzi.jar . -link d
$ java -cp d.jar r.Main
a.p.C: (msg from atom-private a.p.D)
```

Replacing all references to a in d with references to b gives the following output:

```
$ java -jar jiazzi.jar . -link d
$ java -cp d.jar r.Main
b.p.C: (msg from atom-private b.p.D)
```

The above example show that, once the framework has been set up, it is relatively easy to link packages, atoms, and other compounds as desired.

Note that any reference to the class p.D from c's code results in a compilation error: p's package signature p_s does not include D, therefore no stub file for that class was generated. If c also contained a class p.C, it would be ignored, since all references to package p are re-directed to atom a. Therefore, since package bindings must be specified explicitly, the name resolution always has a single possible target definition.

Our last example defines a component, e, that encapsulates the component d:

```
compound e {
  export r : r_s;
} {
  link unit d : d;
  link package d@r to r;
}
```

This example show that re-exporting is possible. However, to successfully link `e`, we not only require the definition of `d`, but also the definitions of `a` (or `b`) and `c`. This implies that the module visibility relation is transitive.

To summarise, Jiazzi supports selective exporting of classes through its package signatures, optional re-exporting of packages through its units, and sub-modules, since compound units can encapsulate atoms and other compounds. However, its module visibility is transitive, there are no unit instantiation policies (at runtime, units are never shared), and there is no support for renaming or versioning. Jiazzi is largely based on its package signatures, which can be parametrised to easily adapt to multiple Java packages (not shown); there is also some support for package-level generics (not shown). When creating a linking unit, i.e. a sub-module, within a compound, we can name it freely, and use that name within the same unit file; however, the underlying Java code cannot refer to any unit, only packages. There is also no support for unit-level generics.

2.2.6 General overview

The table in Fig. 2.1 (page 54) shows which of the analysed module systems has which property or features — these properties and features correspond to a subset of those mentioned in Fig. 1.5 (page 32); the table also includes Java classes and Java packages for comparison. As mentioned at the beginning of this section, we only focused on the properties and features most relevant to this thesis. A larger table that includes the module systems developed in this thesis is shown in Chapter 9.

Apart from the examples shown above, we created a diamond import example for all of the analysed module systems — the source code for these can be found online:

`http://www.cl.cam.ac.uk/~rs456/thesis/diamond.zip`

For example, module *D* imports modules *B* and *C*, both of which import module *A* and manipulate its static state. Within a single application, *B* and *C* share their instance of *A* in all analysed module systems. OSGi also shares its instances across applications; however, OSGi's sharing can sometimes be, as the above examples show, unpredictable. As we shall observe later (Chapter 5), OSGi and JMS has similar sharing principles.

Design space discussion

We have now shown what subset of a specific set of module-level features a few selected module systems have. However, we have so far not explained why some features are left unsupported. This subsection gives a broad discussion about the design space of the selected module-level properties and features.

PROPERTY	JAVA CLASSES	JAVA PACKAGES	JMS MODULES	OSGI BUNDLES	.NET ASSEMBLIES	OCAML MODULES	JIAZZI UNITS
	MODULE(-LIKE) FEATURE						
parametricity	✓	✗	✗	✗	✗	✓	✓
module-prefixed type references	✓	✓	✗	✗	✗	✓	✗
module instantiation policies	✓	✗	✗	✗	✗	✗	✗
multiple files per module	✗	✓	✓	✓	✓	✓	✓
non-transitive module visibility	✗	✗	✗	✗	✗	✓	✗
optional re-exporting	✗	✗	✓	✓	✗	✓	✓
renaming of exports by exporters	✗	✗	✗	✗	✗	✓	✗
renaming of exports by importers	✗	✗	✗	✗	✓	✓	✗
renaming of imports	✗	✗	✗	✗	✓	✓	✗
runtime sharing among programs	✗	✗	✓	✓	✗	✗	✗
selective exporting	✓	✓	✓	✓	✓	✓	✓
sub-modules	✓	✗	✗	✗	✗	✓	✓
sub-modules defined in own files	✗	✗	✗	✗	✗	✓	✓
versioning	✗	✗	✓	✓	✓	✗	✗

Figure 2.1: Overview of properties/features for a few related module systems

A module system often comes late into the design of a programming language; sometimes too late to easily add support for **module-prefixed type references** in a backward-compatible way (e.g. JMS) — while increasing expressivity at the user level, they do not seem to place constraints on other language features. In Chapter 8, we show how module-prefixed type references can be used while localising the influence of a single module.

The possibility of **runtime sharing of modules among programs** allows a runtime system to use a lower memory footprint when executing multiple programs that use the same modules. The price of this feature is the added complexity that comes with the module instance management system. While module **versions** are a useful, low-cost feature for every module system, they are practically required for any module system with support for runtime sharing in order to effectively select and store module definitions.

Module instantiation policies give developers the ability to control module instance sharing and separation, and contribute both to localisation of a single module, and to expressivity in general; however, they can introduce some complexity for the module system. This is discussed further in Chapter 5, where we show how we implement them for JMS.

The added benefit of module instantiation policies is that they allow for **non-transitive**

module visibility, a vital part for strong information hiding (Chapter 5) — this has been achieved for iJAM (Chapter 6). The property can also be obtained through module inclusion (OCaml; Thorn, Chapter 8).

Parametricity can be implemented for Java packages (e.g. Jiazzi), JMS module definitions or OSGi bundles — Thorn (Chapter 8) defines similar module structures to JMS and OSGi, and defines lightweight parametricity. Although it leads to greater reusability on the source level (if configuration costs are not too high), sharing of parametric modules at runtime is restricted to those with equivalent arguments, only. Furthermore, parametricity can easily introduce a short circuit into the initialisation procedure; either the arguments must be restricted to prevent this, or there must be support for developers to cope with such initialisation exceptions.

A great feature for improving expressivity of module composition is **optional re-exporting**, especially when **sub-modules** are involved (Chapter 8); however, it also introduces the question of whether to explicitly bind all type references at compile time (for execution speed purposes), or whether to resolve them dynamically (for greater versatility of modules). Many forms of **renaming** (that improve robustness and, thus, scalability) also share this dilemma.

In general, a module-level feature is often connected to or required by another. Each feature has its implementation, module system complexity, and runtime cost. Whether to include a certain feature heavily depends on the style and type of programs written in a specific programming language. In Chapter 8, however, we show how a module system can define all of the mentioned features, yet remain lightweight and non-intrusive.

3

Lightweight Java (LJ)

When designing or reasoning about a language feature or a language analysis, researchers try to limit the underlying language to avoid dealing with unnecessary details. For example, object-oriented generics were formalised on top of Featherweight Java (FJ) [23], a substantially simplified model of the Java programming language [19].

Many researchers have used FJ as their base language. However, FJ is not always suitable, since it is purely functional — it does not model state; there are only expressions, which are evaluated completely locally. Therefore, FJ is a poor choice for language analyses or language features that rely on state, e.g. separation logic [43] or mixins [9].

In this chapter, we present Lightweight Java (LJ) [49], a minimal *imperative* core of Java. We chose a minimal set of features that still gives a Java-like feel to the language, i.e. fields, methods, single inheritance, dynamic method dispatch, and method overriding. We did not include type casts, local variables, field hiding, interfaces, method overloading, or any of the more advanced language features mainly due to their apparent orthogonality to the module system; however, we later realised that, by including type casts and static data, we could formally verify properties regarding class cast exceptions (or their lack of) and module state independence — this extension remains future work.

LJ's semantics uses a program heap, and a variable state, but does not model a frame stack — method calls are effectively flattened as they are executed, which simplifies the semantics. In spite of this, LJ is a proper subset of Java, i.e. every LJ program is a valid Java program, while its observable semantics exactly corresponds to Java's semantics.

LJ is largely a simplification of Middleweight Java (MJ) [6]. In addition to the above,

MJ models a stack, type casts, and supports expressions (not just statements).

LJ is defined rigorously. It is designed in Ott [46], a tool for writing definitions of programming languages and calculi; all LaTeX presenting the syntax, the semantic rules, and various in-text language terms, is generated by Ott. From LJ's Ott code, the tool also generates the language definition in Isabelle/HOL [3], a tool for writing computer-verified maths. Based on this definition, we mechanically prove type soundness in Isabelle/HOL, which gives us high confidence in the correctness of the results.

Initially, we designed LJ as a base language for modelling the Java Module System (Chapter 4), and its improvement (Chapter 6) — in both, we achieved a high level of reuse in both the definitions and proof scripts. Through this process, LJ has been abstracted to the point where we think it can be used for experimenting with other language features. In fact, LJ has already been used by others to formalise "features" in LFJ [14].

The following sections show an example LJ program (§3.1), LJ's syntax (§3.2), operational semantics (§3.3), type system (§3.4), type checking (§3.5), and a detailed proof of type soundness (§3.6).

The full Ott definition, the complete Isabelle/HOL proof of type soundness, and various other LJ documents can be found at the following address:

`http://www.cl.cam.ac.uk/research/pls/javasem/lj/`

3.1 Example program

Here are two Lightweight Java class definitions, which show the use of class fields, class methods, class inheritance, method overriding, subtyping, and dynamic method dispatch.

```
class A {                  // class definition
    A f;                   // class field
    A m(B var) { this.f = var; return var; }    // subtyping
}

class B extends A {        // class inheritance
    A m(B var) { this.f = var; return this; }  // overriding
}

// A a, result; B b;
a = new B();               // subtyping
b = new B();
result = a.m(b);           // dynamic method dispatch (calls B::m)
```

Due to method overriding, the method call on the last line calls `B`'s method `m`. Therefore, when the execution stops, both `result` and `a` point to the same heap location.

3.2 Syntax

This section presents the *user syntax*, i.e. the abstract syntax of what the user writes. The syntax is presented in the top-down, depth-first fashion. In our presentation, keywords are **bold**, meta-variables are *italic*, and constants are in `typewriter` font. The over-bars indicate lists, e.g. $\overline{cld}$ is a list of class definitions, and $\overline{s_k}^{\,k}$ stands for $s_1..s_k..s_n$ for some n.

Meta-variables dcl, f, $meth$ and var range over Java identifiers. Meta-variable dcl is used for *derived class* names (cannot be `Object`), f (of meta-type $Field$) for field names, $meth$ (of meta-type $Method$) for method names, and var (of meta-type Var) for parameter names; j, k and l are used as index variables. The user syntax is shown in Fig. 3.1.

P	::=			program (cld list)
	|	$\overline{cld}$	M	def.
cld	::=			class definition
	|	$\mathbf{class}\ dcl\ \mathbf{extends}\ cl\ \{\ \overline{fd}\ \overline{meth\_def}\ \}$		def.
$C,\ cl$	::=			class name
	|	`Object`		top class
	|	fqn		fully qualified name
fqn	::=			fully-qualified name
	|	dcl		def.
fd	::=			field declaration
	|	$cl\ f\ ;$		def.
$meth\_def$	::=			method definition
	|	$meth\_sig\ \{\ meth\_body\ \}$		def.
$meth\_sig$	::=			method signature
	|	$cl\ meth\ (\overline{vd})$		def.
vd	::=			variable declaration
	|	$cl\ var$		def.
$meth\_body$	::=			method body
	|	$s_1\ ..\ s_k\ \mathbf{return}\ y\ ;$		def.
s	::=			statement
	|	$\{\ \overline{s_k}^{\,k}\ \}$		block
	|	$var = x\ ;$		variable assignment
	|	$var = x\ .f\ ;$		field read
	|	$x\ .f = y\ ;$		field write
	|	$\mathbf{if}\ (x == y)\ s\ \mathbf{else}\ s'$		conditional branch
	|	$var = \mathbf{new}_{ctx}\, cl();$		object creation
	|	$var = x\ .\ meth\ (\overline{y})\ ;$		method call
$TVar,\ x,\ y$	::=			term variable
	|	var		normal variable
	|	$\mathbf{this}$		ref. to current object

Figure 3.1: LJ user syntax

A class definition, cld, defines a class with a class name, dcl (a superclass name cl), a list of field declarations, $\overline{fd}$ (pairs of cl and f), and a list of method definitions, $\overline{meth\_def}$. A method definition, $meth\_def$, is composed of a method signature, $meth\_sig$, and a method body, $meth\_body$. The former is defined with a return class reference, cl, method's name, $meth$, and a list of parameter definitions, $\overline{vd}$ (pairs of cl and var), while the latter is a list of statements, $\overline{s_k}^k$, followed by a return statement, '$\mathbf{return}\ y;$'.

We use cl to refer to either `Object`, or a fully-qualified name fqn. Since LJ does not define Java packages, fqn is defined simply as dcl (derived class); however, in our two extensions of LJ (Chapter 4 and Chapter 6), fqn is a tuple of a package name and dcl, since the two languages *do* define Java packages. Abstracting the concept of fully-qualified names in this way allows a greater reuse of the language definitions and proofs.

The sub-language of statements, s, is simple, and Java-like. We avoid the complexity of arithmetic normal forms and compound expressions, since they do not increase expressivity. For example, if we modelled compound expressions, we would also have to formalise partially-evaluated expressions, evaluation order, and congruence rules, all of which would make the operational semantics much less clear than it is now.

As in Java, the keyword $\mathbf{this}$ is used as a special variable that always points to the object of which code is currently executing. To prevent the programmer from breaking this invariant, assignment to $\mathbf{this}$ is not allowed, which we enforce syntactically. A term variable x is either a variable, var, or $\mathbf{this}$ — x is used when either of the two can appear.

The ctx in '$var = \mathbf{new}_{ctx}\, cl();$' is actually *not* part of the user syntax. It is there to support different forms of runtime loading of class definitions. In LJ, ctx is *always* empty, since class loading in LJ is trivial; however, languages with more complex class loading can use ctx as a compile-time (or load-time) annotation, which guides the class resolution (classloaders) at runtime — examples of this are shown later in §4.3.1 and §6.2.1.

3.3 Operational semantics

We precisely define the execution of LJ programs with small-step operational semantics. The judgement $config \longrightarrow config'$ stands for "a configuration, $config$, reduces to another configuration, $config'$, in one step." In the following subsections, we define configurations (§ 3.3.1), show some lookup functions that operate on the program state (§ 3.3.2), and describe the statement reduction rules (§3.3.3), which use the lookup functions.

3.3.1 Configuration ($config$)

A configuration, $config$, is a tuple consisting of a program, P, a variable state, L, a heap, H, and a list of statements left to execute, $\overline{s}$. A program, P, is a list of class definitions, $\overline{cld}$;

a variable state, L, is a partial map from term variables, x (or y), to their values, v (meta-type: $TVar \rightharpoonup Val$); and a heap, H, is a partial map from object identifiers, oid (non-null pointers), to class types, τ, and field-to-value maps, f-to-v (meta-type: $Pointer \rightharpoonup (Type \times (Field \rightharpoonup Val))$). As explained in §3.4.1, a type, τ, is either `Object` or the name of a derived class, dcl. Values are object identifiers, oid, or **null**. LJ only models one type of exception: the null pointer exception, **NPE**. The *inner syntax*, i.e. the abstract syntax used for representing a program's state, of these concepts is shown in Fig. 3.2.

config	::=			configuration
	|	$(P,\ L,\ H,\ \overline{s_k}^{\,k})$		normal configuration
	|	$(P,\ L,\ H,\ Exception)$		exception occurred
L	::=			variable state ($x \rightharpoonup v$)
	|	$[\,]$	M	empty variable state
	|	$L\,[x \mapsto v]$	M	L with $x \mapsto v$
	|	$L\,[x_1 \mapsto v_1\,..\,x_k \mapsto v_k]$	M	L with many mappings
$Val,\ v,\ w$	::=			value
	|	**null**		null value
	|	oid		object identifier
H	::=			heap ($oid \rightharpoonup (\tau \times (f \rightharpoonup v))$)
	|	$[\,]$	M	empty heap
	|	$H\,[oid \mapsto (\tau,\ f_1 \mapsto v_1\,..\,f_k \mapsto v_k)]$	M	H with new oid of type τ
	|	$H\,[(oid,\ f) \mapsto v]$	M	H with $(oid,f) \mapsto v$
$Exception$	::=			exception
	|	**NPE**		null-pointer exception

Figure 3.2: Abstract syntax used for representing an LJ program's state

The productions of L and H in Fig. 3.2 are *meta productions*, i.e. they describe the syntax that represent functions, which change the inner state of the corresponding structures.

3.3.2 Lookup functions

This subsection shows the lookup functions (directly and indirectly) used within the statement reduction rules. The definitions of these functions are fairly straightforward, which is why we recommend a quick scan-through only, on a first reading; the only non-trivial function here is the one for finding the class inheritance path (page 62). All the functions are presented using the rules described in §1.6.

In our language definitions, the lookup functions hide the concrete types of the configuration structures (Figure 3.2, page 61). In our extensions of LJ, the statement sublanguage remains unchanged, while the program structure becomes substantially more complicated; however, by adapting only the corresponding lookup functions, we are able to reuse the statement reduction rules in their entirety.

Finding a class definition

To find a class definition within a specific program, P, we use a simple linear search through the list of program's class definitions. If an appropriate class definition is found, the function below returns it along with the context in which it was found (in LJ, this is simply the starting context).

$\mathbf{find\_cld}\,(P, ctx, fqn) \;:\; (P \times ctx \times fqn) \to (ctx, cld)_{opt} \;=$
  match P **with** $[] \to$ `None` | $(\mathbf{class}\; dcl\; \mathbf{extends}\; cl\; \{\, \overline{fd}\; \overline{meth\_def}\, \} :: \overline{cld}) \to$
  if $dcl = fqn$ **then** `Some` (ctx, cld) **else** $\mathbf{find\_cld}\,(\overline{cld}, ctx, fqn)$

Finding a type

As mentioned later in §3.4.4, every class definition defines a type. This is why the function that finds a type corresponding to a particular class reference relies on $\mathbf{find\_cld}$: the type is extracted from the context and the class definition found.

$\mathbf{find\_type}\,(P, ctx, cl) \;:\; (P \times ctx \times cl) \to \tau_{opt} \;=$
  match cl **with** `Object` $\to$ `None` | $dcl \to$ **match** $\mathbf{find\_cld}\,(P, ctx, cl)$ **with**
      `None` $\to$ `None` | `Some` $(ctx', cld) \to$ `Some` $ctx'.dcl$

Finding an inheritance path

A class is fully defined by its *inheritance path*, i.e. the class and all of its ancestors (through the superclass relation). Knowing the inheritance path for a particular class reference is vital in many parts of the LJ semantics — here we describe how this is done.

We split the function into two sub-functions: $\mathbf{find\_path\_rec}$ and $\mathbf{find\_path}$. The former includes a recursive function call and implements most of the semantics, while the latter provides a nice interface.

$\mathbf{find\_path\_rec}\,(P, ctx, cl, \overline{ctxcld}) \;:\; (P \times ctx \times cl \times \overline{ctxcld}) \to \overline{ctxcld}_{opt} \;=$
  match cl **with** `Object` $\to$ `Some` $\overline{ctxcld}$ | $fqn \to$
  if $\neg(\mathbf{acyclic\_clds}\,P)$ **then** `None` **else match** $\mathbf{find\_cld}\,(P, ctx, fqn)$ **with**
    `None` $\to$ `None` | `Some` $(ctx', cld) \to$
      $\mathbf{find\_path\_rec}\,(P, ctx', \mathbf{superclass\_name}\,(cld), \overline{ctxcld}\;@\;[(ctx', cld)])$

Explanation. The function tries to find the inheritance path that corresponds to the given class name, cl (in the given ctx and program P). The fourth parameter to the function, $\overline{ctxcld}$, is the function's accumulator used to store intermediate results between function's recursive calls. If the definitions for all the classes in the inheritance path are found, the function returns the path; otherwise, the function fails by returning None. The function $\mathbf{superclass\_name}$ simply extracts the name of the superclass from the given class definition; $ctxcld$ is a shorthand for '$(ctx,\ cld)$' (for some fresh meta-variables ctx and cld); $\mathbf{acyclic\_clds}$ is described below.

Proof of Termination. When defining Isabelle/HOL functions with non-primitive recursion, one also has to prove their termination. Specifically, one has to provide a measure of the arguments, which provably decreases with each recursive call [36, §3.5.2].

For the above function, this measure is: *the length of the inheritance path from the current argument class*. This $\mathbf{path\_length}$ is defined as follows.

$$\frac{}{(P,\ ctx,\ \texttt{Object},\ 0) \in \mathbf{path\_length}}\ \text{PL\_OBJ}$$

$$\frac{\begin{array}{l}\mathbf{find\_cld}\,(P,\ ctx,\ fqn) = (ctx',\ cld)\\ \mathbf{superclass\_name}\,(cld) = cl\\ (P,\ ctx',\ cl,\ nn) \in \mathbf{path\_length}\end{array}}{(P,\ ctx,\ fqn,\ nn\ +\ 1) \in \mathbf{path\_length}}\ \text{PL\_FQN}$$

Since $\mathbf{path\_length}$ is a relation, and not a function, we have to use the definite descriptor [36, §5.10.1], ι, to obtain the length, nn, that corresponds to the class reference, cl (in ctx, P). The value that gets smaller with each recursive call is:

$$\iota\ nn.(P,\ ctx,\ cl,\ nn) \in \mathbf{path\_length}\ .$$

To ensure that there exists a finite nn for each recursive call, we place an acyclicity constraint on the inheritance relation among the class definitions in the program. The $\mathbf{acyclic\_clds}$ relation is defined as follows.

$$\frac{\forall ctx\ fqn\ .\ \left(\begin{array}{c}(\exists ctx'\ cld\ .\ \mathbf{find\_cld}\,(P,\ ctx,\ fqn) = (ctx',\ cld)) \longrightarrow\\ \exists nn\ .\ (P,\ ctx,\ fqn,\ nn) \in \mathbf{path\_length}\end{array}\right)}{\mathbf{acyclic\_clds}\ P}\ \text{AC\_DEF}$$

Explanation. If we can find a class definition for some ctx and fqn (in LJ, there is no class definition for `Object`), then the corresponding inheritance path has a *finite* length.

Since Isabelle/HOL does not support dependent types, we simulate them by testing for acyclicity at the beginning of the function — if a program does not satisfy the property, the function fails. As shown later in §3.5, a well-formed program is acyclic by definition.

Given the acyclicity constraint, it is relatively easy to show that the $\mathbf{path\_length}$ measure decreases with each recursive call. □

$\mathbf{find\_path}\,(P, ctx, cl)\ :\ (P \times ctx \times cl) \to \overline{ctxcld}_{opt} =$
  $\mathbf{find\_path\_rec}\,(P, ctx,\ cl, [])$

Explanation. This function is a friendly interface to $\mathbf{find\_path\_rec}$: it starts the search with an empty accumulator.

Looking up the inheritance path for a type

Often, we already hold a type, and want to find the corresponding inheritance path. This function does just that. It uses the $\mathbf{find\_path}$ function.

$\mathbf{find\_path}\,(P, \tau)\ :\ (P \times \tau) \to \overline{ctxcld}_{opt}\ =$
  match `ty` **with** `Object` $\to$ `Some []` | $ctx.dcl \to \mathbf{find\_path}\,(P, ctx, dcl)$

Getting field names of a class

Knowing all field names for a class is required when creating a field-value map for the newly created objects. We obtain the field names through the inheritance path, and scan through each class definition for field definitions. The function **class_fields** simply extracts the field definitions from the given class definition.

$\mathbf{fields}\,(P,\tau)\ :\ (P\times\tau)\to\overline{f}_{opt}\ =$
  match $\mathbf{find\_path}\,(P,\tau)$ **with**
    None $\to$ None | Some $\overline{ctxcld}\to$ Some $(\mathbf{fields\_in\_path}\,(\overline{ctxcld}))$

$\mathbf{fields\_in\_path}\,(\overline{ctxcld})\ :\ \overline{ctxcld}\to\overline{f}\ =$
  match $\overline{ctxcld}$ **with** $[]\to[]$ | $(ctx,cld)::\overline{ctxcld}'\to$
    $(\mathbf{map}\,(\lambda(cl\,f\,;)\to f)\ (\mathbf{class\_fields}\,(cld)))\ @\ \mathbf{fields\_in\_path}\,(\overline{ctxcld}')$

Finding method definitions

Method definition lookups appear both in the well-formedness rules, and in the reduction rules. To look up a method definition for some type, τ, we first find the type's inheritance path, and then perform a linear search through it.

$\mathbf{find\_meth\_def}\,(P,\tau,meth)\ :\ (P\times\tau\times meth)\to(ctx\times meth\_def)_{opt}\ =$
  match $\mathbf{find\_path}\,(P,\tau)$ **with** None $\to$ None | Some $\overline{ctxcld}\to$
    Some $(\mathbf{find\_meth\_def\_in\_path}\,(\overline{ctxcld},meth))$

In the second step, method definitions of each class in the inheritance path are searched for a method with the given name. Since the subclasses are searched first, the function respects method overriding.

$\mathbf{find\_meth\_def\_in\_path}\,(\overline{ctxcld},meth)\ :\ (\overline{ctxcld}\times meth)\to(ctx\times meth\_def)_{opt}\ =$
  match $\overline{ctxcld}$ **with** $[]\to$ None | $((ctx,cld)::\overline{ctxcld}')\to$
  match $\mathbf{find\_meth\_def\_in\_list}\,(\mathbf{class\_methods}\,(cld),meth)$ **with**
    Some $meth\_def\to$ Some $(ctx,meth\_def)$
  | None $\to\mathbf{find\_meth\_def\_in\_path}\,(\overline{ctxcld}',meth)$

When searching through method definitions of a particular class, the first method definition with a matching name is returned — as shown later in §3.5.2 (page 74), method names within a class must be distinct, which implies that 'if there is a method with a matching name, it is the only such method within the method definitions of a class.'

$\mathbf{find\_meth\_def\_in\_list}\,(\overline{meth\_def},meth)\ :\ (\overline{meth\_def}\times meth)\to meth\_def_{opt}\ =$
  match $\overline{meth\_def}$ **with** $[]\to[]$ | $(cl\,meth'\,(\overline{vd})\,\{\,meth\_body\,\})\ @\ \overline{meth\_def}')\to$
  if $meth=meth'$ **then** $cl\,meth'\,(\overline{vd})\,\{\,meth\_body\,\}$
  else $\mathbf{find\_meth\_def\_in\_list}\,(\overline{meth\_def}',meth)$

$$\frac{}{(P,\ L,\ H,\ \{\ \overline{s_k}^{\,k}\ \}\ \overline{s'_l}^{\,l})\ \longrightarrow\ (P,\ L,\ H,\ \overline{s_k}^{\,k}\ \overline{s'_l}^{\,l})}\ \text{R\_BLOCK}$$

$$\frac{1.\ L\,(x) = v}{(P,\ L,\ H,\ var = x\,;\ \overline{s_l}^{\,l})\ \longrightarrow\ (P,\ L\,[var\ \mapsto\ v],\ H,\ \overline{s_l}^{\,l})}\ \text{R\_VAR\_ASSIGN}$$

$$\frac{1.\ L\,(x) = \mathbf{null}}{(P,\ L,\ H,\ var = x\,.f\,;\ \overline{s_l}^{\,l})\ \longrightarrow\ (P,\ L,\ H,\ \mathbf{NPE})}\ \text{R\_FIELD\_READ\_NPE}$$

$$\frac{1.\ L\,(x) = oid \qquad 2.\ H\,(oid,\ f) = v}{(P,\ L,\ H,\ var = x\,.f\,;\ \overline{s_l}^{\,l})\ \longrightarrow\ (P,\ L\,[var\ \mapsto\ v],\ H,\ \overline{s_l}^{\,l})}\ \text{R\_FIELD\_READ}$$

$$\frac{1.\ L\,(x) = \mathbf{null}}{(P,\ L,\ H,\ x\,.f = y\,;\ \overline{s_l}^{\,l})\ \longrightarrow\ (P,\ L,\ H,\ \mathbf{NPE})}\ \text{R\_FIELD\_WRITE\_NPE}$$

$$\frac{1.\ L\,(x) = oid \qquad 2.\ L\,(y) = v}{(P,\ L,\ H,\ x\,.f = y\,;\ \overline{s_l}^{\,l})\ \longrightarrow\ (P,\ L,\ H\,[(oid,\ f)\ \mapsto\ v],\ \overline{s_l}^{\,l})}\ \text{R\_FIELD\_WRITE}$$

$$\frac{1.\ L\,(x) = v \qquad 2.\ L\,(y) = w \qquad 3.\ v = w}{(P,\ L,\ H,\ \mathbf{if}\,(x == y)\ s_1\ \mathbf{else}\ s_2\ \overline{s'_l}^{\,l})\ \longrightarrow\ (P,\ L,\ H,\ s_1\ \overline{s'_l}^{\,l})}\ \text{R\_IF\_TRUE}$$

$$\frac{1.\ L\,(x) = v \qquad 2.\ L\,(y) = w \qquad 3.\ v\ \neq\ w}{(P,\ L,\ H,\ \mathbf{if}\,(x == y)\ s_1\ \mathbf{else}\ s_2\ \overline{s'_l}^{\,l})\ \longrightarrow\ (P,\ L,\ H,\ s_2\ \overline{s'_l}^{\,l})}\ \text{R\_IF\_FALSE}$$

$$\frac{\begin{array}{l}1.\ \mathbf{find\_type}\,(P,\ ctx,\ cl) = \tau\ \ 2.\ \mathbf{fields}\,(P,\ \tau) = \overline{f_k}^{\,k}\\ 3.\ oid\ \notin\ \mathbf{dom}\,(H)\quad 4.\ H' = H\,[oid\ \mapsto\ (\tau,\ \overline{f_k \mapsto \mathbf{null}}^{\,k})]\end{array}}{(P,\ L,\ H,\ var = \mathbf{new}_{ctx}\,cl();\ \overline{s_l}^{\,l})\ \longrightarrow\ (P,\ L\,[var\ \mapsto\ oid],\ H',\ \overline{s_l}^{\,l})}\ \text{R\_NEW}$$

$$\frac{1.\ L\,(x) = \mathbf{null}}{(P,\ L,\ H,\ var = x\,.\,meth\,(\overline{y_k}^{\,k})\,;\ \overline{s_l}^{\,l})\ \longrightarrow\ (P,\ L,\ H,\ \mathbf{NPE})}\ \text{R\_MCALL\_NPE}$$

R_MCALL

$$\frac{\begin{array}{l}1.\ L\,(x) = oid \qquad 2.\ H\,(oid) = \tau\\ 3.\ \mathbf{find\_meth\_def}\,(P,\ \tau,\ meth) = (ctx,\ cl\ meth\,(\overline{cl_k\ var_k}^{\,k})\ \{\ \overline{s'_j}^{\,j}\ \mathbf{return}\ y\,;\ \})\\ 4.\ \overline{var'_k}^{\,k} \perp \mathbf{dom}\,(L) \qquad 5.\ \mathbf{distinct}\,(\overline{var'_k}^{\,k}) \qquad 6.\ x' \notin \mathbf{dom}\,(L)\\ 7.\ x' \notin \overline{var'_k}^{\,k} \qquad 8.\ \overline{L\,(y_k) = v_k}^{\,k}\\ 9.\ L' = L\,[\overline{var'_k \mapsto v_k}^{\,k}]\,[x'\ \mapsto\ oid]\\ 10.\ \theta = [\overline{var_k \mapsto var'_k}^{\,k}]\,[\mathbf{this}\ \mapsto\ x'] \qquad 11.\ \overline{\theta\ \vdash\ s'_j\ \rightsquigarrow\ s''_j}^{\,j}\\ 12.\ \theta\,(y) = y'\end{array}}{(P,\ L,\ H,\ var = x\,.\,meth\,(\overline{y_k}^{\,k})\,;\ \overline{s_l}^{\,l})\ \longrightarrow\ (P,\ L',\ H,\ \overline{s''_j}^{\,j}\ var = y'\,;\ \overline{s_l}^{\,l})}$$

Figure 3.3: LJ's small-step operational semantics for statements

3.3.3 Statement reduction ($config \longrightarrow config'$)

The small-step operational semantics of LJ is shown in Fig. 3.3. Except for the object creation statement, '$var = \mathbf{new}_{ctx}\, cl()$;', reduction rule R_NEW, and the method call statement, '$var = x\,.\,meth\,(\overline{y_k}^{\,k})$;', reduction rule R_MCALL, the rules are straightforward.

As in Java, de-referencing an invalid pointer throws a null-pointer exception, **NPE**. This can happen when reading a field (if x in '$var = x\,.f$;' is **null**), when writing to a field (if x in '$x\,.f = y$;' is **null**), or when calling a method (if x in '$var = x\,.\,meth\,(\overline{y_k}^{\,k})$;' is **null**). Since LJ has no exception handling mechanism, these exceptions terminate the program, resulting in the second form of the configuration (as shown in Fig. 3.2). All such exceptions can be avoided by checking the term variable in question: since LJ has only the most basic syntax, where variables cannot be compared directly to values, one has to assign **null** to a term variable, then check the equality between the variable and the term variable in question in an if statement before executing any of the above statements.

The object creation statement creates an entry in the heap, which includes a map that holds values for all fields of the new object. The first two steps of the R_NEW rule resolve (with **find_type**, in context ctx) the class reference, cl, to a type, τ, and locate (with **fields**) the type's fields. These two steps must have already succeeded during the type-checking phase (shown later in §3.5.2, page 76), which is why we know that they will not fail during execution. Alternatively, we could have stored the fields found during the type-checking phase within the compiled version of the object creation statement.

Method calls are effectively flattened, since LJ does not model a frame stack. When a method is called on an object, the definition of that method is looked up within the class definitions that define the object. The body of the method is then inserted into the list of statements to be executed. To make sure that variable binding within the method is logically preserved, parameter names are replaced with fresh variables. Benefits and drawbacks of not having a stack are discussed in the conclusion of this chapter (§3.7).

For example, suppose we execute a statement '`x=x.m(x,y);`', which calls the method '`void m(A y, A x){x=x.f; return y;}`'. Before we can execute the method's body, we need to assign values to parameter names, `x` and `y`. Without fresh names, we would be assigning the values of `x` and `y` to the names `y` and `x` in the global context, respectively — that is, the values of `x` and `y` would be incorrectly swapped even after the method call. Another bad approach would be to translate the method body using the parameter-argument name mapping, e.g. the above method call would be flattened into '`y=y.f; x=x;`' — this is incorrect, since the change to `y` should remain local to the method call. In other words, our evaluation strategy is call-by-value (as in Java), not call-by-name. For this reason, we create fresh parameter variables, bind them to argument values, and execute the method's body, where the original parameter names are replaced with their fresh counterparts.

We explain R_MCALL's premises here one-by-one: (1) obtain value (pointer oid) of x

in L; (2) obtain the type, τ, for oid through the heap, H; (3) find the method definition in τ with name $meth$; (4) obtain fresh names, $\overline{var'_k}^{\,k}$, for method's parameters, $\overline{var_k}^{\,k}$ (here, $\perp$ is defined as an infix proposition, which holds when the two argument sets are disjoint); (5) make sure $\overline{var'_k}^{\,k}$ are distinct from each other; (6) obtain a fresh name, x'; (7) make sure x' is also distinct from any variable in $\overline{var'_k}^{\,k}$; (8) look up values, $\overline{v_k}^{\,k}$, of method arguments, $\overline{y_k}^{\,k}$, in L; (9) extend the variable state, L, to L' by element-wise mapping $\overline{var'_k}^{\,k}$ to $\overline{v_k}^{\,k}$, and x' to oid; (10) create a variable-to-variable mapping, θ, which element-wise maps the method's original parameters, $\overline{var_k}^{\,k}$, to the fresh variables, $\overline{var'_k}^{\,k}$, and $\mathbf{this}$ to x'; (11) translate the variables in the statements of the method, $\overline{s'_j}^{\,j}$, to $\overline{s''_j}^{\,j}$ using θ; and (12) translate the method's return variable, y, to y' using θ. The method call statement, '$var = x\,.\,meth\,(\overline{y_k}^{\,k})\,;$', is then replaced with the translated statements, $\overline{s''_j}^{\,j}$, and with the original var assigned to the translated returned variable, y', obtaining '$\overline{s''_j}^{\,j}\ var = y'\,;$'.[1]

3.3.4 Variable translation ($\theta \vdash s \rightsquigarrow s'$)

As described above, the formalisation of a method call statement reduction requires fresh variables and variable translation: each time a method is called, a fresh set of variables is created (in the variable state L) that corresponds to the parameters of a method called; before the statements of a method are executed, the variables in those statements are renamed to the corresponding fresh variables.

In particular, variable translation is performed in step (11) of R_MCALL (Fig. 3.3, page 65). The structure used to perform this translation is a variable-to-variable map, θ: $x \rightharpoonup x$. The straightforward variable translation rules are shown in Fig. 3.4.

3.4 Type system

An LJ program, which was not checked before execution, could contain errors that would stop the execution, e.g. trying to call an undefined method on an object. Such program errors, known as *type errors*, are not defined in our semantics, since they are prevented by the *type-checking* phase, which occurs before execution.

Type-checking is based on the *type system* of a language, which defines what types are, and how they are related. This section presents LJ's type system.

3.4.1 Type (τ)

`Object` is the supertype of all types, i.e. a value of any type can be used where an `Object` is expected.[2] Each class definition, cld, defines a type. Therefore, a *type*, τ, is either

[1] There is no ';' between $\overline{s''_j}^{\,j}$ and '$var = y'\,;$', since a non-empty $\overline{s''_j}^{\,j}$ already ends with ';' or '}'.

[2] One can also instantiate the `Object` type; however, the resulting object contains no state or functionality.

$$\text{TR\_S\_BLOCK}\quad \frac{1.\ \overline{\theta \vdash s_k \rightsquigarrow s'_k}^{\,k}}{\theta \vdash \{\ \overline{s_k}^{\,k}\ \} \rightsquigarrow \{\ \overline{s'_k}^{\,k}\ \}} \qquad \text{TR\_S\_VAR\_ASSIGN}\quad \frac{1.\ \theta\,(var) = var' \quad 2.\ \theta\,(x) = x'}{\theta \vdash var = x\,; \rightsquigarrow var' = x';}$$

$$\text{TR\_S\_FIELD\_READ}\quad \frac{1.\ \theta\,(var) = var' \quad 2.\ \theta\,(x) = x'}{\theta \vdash var = x\,.f\,; \rightsquigarrow var' = x'\,.f\,;} \qquad \text{TR\_S\_FIELD\_WRITE}\quad \frac{1.\ \theta\,(x) = x' \ \ 2.\ \theta\,(y) = y'}{\theta \vdash x\,.f = y\,; \rightsquigarrow x'\,.f = y';}$$

$$\text{TR\_S\_IF}\quad \frac{1.\ \theta\,(x) = x' \quad 2.\ \theta\,(y) = y' \quad 3.\ \theta \vdash s_1 \rightsquigarrow s'_1 \quad 4.\ \theta \vdash s_2 \rightsquigarrow s'_2}{\theta \vdash \textbf{if}\ (x == y)\ s_1\ \textbf{else}\ s_2 \rightsquigarrow \textbf{if}\ (x' == y')\ s'_1\ \textbf{else}\ s'_2}$$

$$\text{TR\_S\_NEW}\quad \frac{1.\ \theta\,(var) = var'}{\theta \vdash var = \textbf{new}_{ctx}\,cl(); \rightsquigarrow var' = \textbf{new}_{ctx}\,cl();}$$

$$\text{TR\_S\_MCALL}\quad \frac{1.\ \theta\,(var) = var' \quad 2.\ \theta\,(x) = x' \quad 3.\ \overline{\theta\,(y_k) = y'_k}^{\,k}}{\theta \vdash var = x\,.\,meth\,(\overline{y_k}^{\,k})\,; \rightsquigarrow var' = x'\,.\,meth\,(\overline{y'_k}^{\,k})\,;}$$

Figure 3.4: LJ's variable translation within method calls

`Object`, or the name of a defined class, dcl.

The latter is prefixed with ctx so that type identifiers uniquely identify a class definition also in language extensions where ctx is non-empty.

$Type,\ \tau$	::=		type
	|	`Object`	supertype of all types
	|	$ctx.dcl$	class identifier

3.4.2 Type environment (Γ)

A method parameter is declared using two names: a variable, var, and a class name, cl. The variable is the parameter's name, while the class name stands for the parameter's type.

When type-checking the body of a method, we use a *type environment*, Γ, which stores the types of variables in scope. The type environment is, therefore, a partial map from term variables, x (either var or $\mathbf{this}$), to types: $TVar \rightharpoonup Type$.

Γ	::=			type environment ($x \rightharpoonup \tau$)
	|	$[x_1 \mapsto \tau_1\,..\,x_k \mapsto \tau_k]$	M	type mappings
	|	$\Gamma\,[x \mapsto \tau]$	M	Γ with $x \mapsto \tau$

3.4.3 Subtyping ($P \vdash \tau \prec \tau'$)

The *subtyping* relation states when a value of a type can be used when a value of another type, supertype, is expected. Our subtyping relation is both reflexive and transitive.

Most languages define the subtyping relation as a transitive closure of class extension. An example of this is FJ [23]. LJ's subtyping is defined in a non-standard way: a type, τ, is a subtype of τ' *iff* a class definition corresponding to τ is in the inheritance path of τ'.

$$\frac{1.\ \mathbf{find\_path}\,(P,\ \tau) = \overline{ctxcld}}{P \vdash \tau \prec \texttt{Object}}\ \text{STY\_OBJ}$$

$$\frac{\begin{array}{l}1.\ \mathbf{find\_path}\,(P,\ \tau) = \overline{(ctx_k,\ cld_k)}^{\,k} \\ 2.\ \overline{\mathbf{class\_name}\,(cld_k) = dcl_k}^{\,k} \\ 3.\ (ctx',\ dcl') \in \overline{(ctx_k,\ dcl_k)}^{\,k}\end{array}}{P \vdash \tau \prec ctx'.dcl'}\ \text{STY\_DCL}$$

The **find_path** function (page 63) finds the inheritance path, $\overline{ctxcld}$, for a type, τ, while **class_name** extracts the class name, cl, from a class definition, cld.

We derive type reflexivity and transitivity properties from the relation. This definition makes it relatively easy to prove many of our lemmas in Isabelle/HOL, since quite a few of them are based on functions that are defined in terms of **find_path**, too. An example of such a lemma is method type preservation (shown later in §3.5.1, page 73).

3.4.4 Valid type ($P \vdash \tau$)

Although every class definition defines a type, a class definition can have an invalid reference to its superclass. Since the class inheritance relation and the subtyping relation rely on validity of these references, class definitions should not contain such errors.

We define the concept of a *valid type* here in order to simplify the discussion of type reflexivity and transitivity: a type, τ, is valid in a program, P, *iff* there is an inheritance path in P that corresponds to τ:

$$P \vdash \tau \stackrel{\text{def}}{=} \exists \overline{ctxcld}.\ \mathbf{find\_path}\,(P,\ \tau) = \overline{ctxcld}$$

We can now show that if τ is a subtype of τ' in P, then both τ and τ' are valid types.

Lemma 1 (Subtype is a valid type)**.** $P \vdash \tau \prec \tau' \implies P \vdash \tau$

Proof. Trivially follows from the definition of a valid type (above) and the subtype relation (§3.4.3). □

Showing that the supertype is also a valid type is more difficult. We can prove it with the help of the following lemma (this lemma is used throughout our proof scripts):

Lemma 2 (Inheritance sub-path)**.**

$$\begin{array}{l}\mathbf{find\_path}\,(P,\ ctx,\ cl) = \overline{ctxcld} \quad \wedge \quad (ctx',\ cld') \in \overline{ctxcld} \quad \wedge \\ \mathbf{class\_name}\,(cld') = dcl' \quad \wedge \quad \mathbf{find\_path}\,(P,\ ctx',\ dcl') = \overline{ctxcld}' \\ \qquad \Longrightarrow \exists \overline{ctxcld}''.\ \ \overline{ctxcld} = \overline{ctxcld}''\ @\ \overline{ctxcld}'\end{array}$$

Explanation. If we look up (in context ctx and program P) an inheritance path, $\overline{ctxcld}$, for a class reference, cl, and then look up the inheritance path, $\overline{ctxcld}'$, for a class, cld', and context ctx' in $\overline{ctxcld}$, then $\overline{ctxcld}'$ is a suffix of $\overline{ctxcld}$.

Proof. By first proving a generalised version of the lemma, which also unfolds **find_path** (page 63) to express the desired property with **find_path_rec** (page 62) instead:

$$\begin{array}{l}\mathbf{find\_path\_rec}\,(P, ctx, cl, \overline{ctxcld}'_1)\ \texttt{=}\ \texttt{Some}\ \ \overline{ctxcld}_1 \implies \\ \quad (\forall (ctx',\ cld) \in \overline{ctxcld}_1.\ \ (ctx',\ cld) \in \overline{ctxcld}'_1 \quad \vee \\ \quad\ (\forall \overline{ctxcld}_2.\ \ \mathbf{find\_path\_rec}\,(P, ctx', \mathbf{class\_name}\,(cld), \overline{ctxcld}'_2)\ \texttt{=}\ \texttt{Some}\ \ \overline{ctxcld}_2 \Longrightarrow \\ \quad\quad (\forall \overline{ctxcld}_3.\ \ \overline{ctxcld}_2 = \overline{ctxcld}'_2\ \ @\ \ \overline{ctxcld}_3 \implies \\ \quad\quad\ (\exists \overline{ctxcld}_4.\ \ \overline{ctxcld}_1 = \overline{ctxcld}_4\ \ @\ \ \overline{ctxcld}_3))))\end{array}$$

Proof. By a well-founded induction on **find_path_rec**. When '$cl = \texttt{Object}$', the lemma trivially holds. When '$\exists fqn.\ cl = fqn$', the call splits into '$\mathbf{find\_cld}\,(P,\ ctx,\ fqn) = (ctx',\ cld')$' and '$\mathbf{find\_path\_rec}\,(P,\ ctx',\ cl',\ \overline{ctxcld}'_1\ @\,[(ctx',\ cld')]) = \overline{ctxcld}_1$' where '$\mathbf{superclass\_name}\,(cld') = cl'$'. Through the induction hypothesis (and a lemma that shows that **find_cld** finds a class definition with the requested class name), we know that '$\mathbf{find\_path\_rec}\,(P,\ ctx',\ cl',\ \overline{ctxcld}'_2\ @\,[(ctx',\ cld')]) = \overline{ctxcld}'_2 @ \overline{ctxcld}_3$'. Through another lemma[3] again proved by a well-founded induction on **find_path_rec**, we know that '$\exists \overline{ctxcld}_5.\ \overline{ctxcld}_1 = \overline{ctxcld}'_1\ @\,[(ctx',\ cld')] @ \overline{ctxcld}_5$' and '$\exists \overline{ctxcld}_6.\ \overline{ctxcld}'_2 @ \overline{ctxcld}_3 = \overline{ctxcld}'_2 @ [(ctx',\ cld')] @ \overline{ctxcld}_6$'. Through a variant of that lemma,[4] we can then deduce '$\overline{ctxcld}_5 = \overline{ctxcld}_6$'. Finally, we can show '$\overline{ctxcld}_1 = \overline{ctxcld}'_1\ @\,[(ctx',\ cld')] @ \overline{ctxcld}_6 = \overline{ctxcld}'_2 @ \overline{ctxcld}_3$'. □

Having proven the general case, we instantiate the accumulator above, $\overline{ctxcld}_1$, with an empty list, simplify, and fold the **find_path_rec** calls to obtain Lemma 2. □

Lemma 3 (Supertype is a valid type)**.** $P \vdash \tau \prec \tau' \implies P \vdash \tau'$

Proof. Using Lemma 2 with the definition of both the subtyping relation (above) and the definition of a valid type (§3.4.4). □

[3] Named `path_append`, it shows that **find_path_rec**'s accumulator is the result's prefix.

[4] Named `fpr_same_suffix`, it shows that changing **find_path_rec**'s accumulator will change the result's prefix accordingly.

3.4.5 Type reflexivity

As mentioned before, type reflexivity is one of the two basic properties of a type system. Here, we show that this property holds for every valid LJ type.

Lemma 4 (Type reflexivity)**.** $P \vdash \tau \implies P \vdash \tau \prec \tau$

Proof. Apart from using the definitions of both the valid type (§3.4.4) and the subtype relation (§3.4.3), proving this lemma requires one to show that 'a class definition corresponding to τ is in the inheritance path of τ', which is relatively easy to prove. □

In our Isabelle/HOL proofs, however, we use the following (and weaker) version of the above lemma, since it proved convenient for the soundness proof.

Corollary 5 (Type Reflexivity — Variation)**.** $P \vdash \tau \prec \tau' \implies P \vdash \tau \prec \tau$

Proof. Trivially follows from Lemma 1 and Lemma 4. □

3.4.6 Type transitivity

The transitivity property is the other basic property of a type system. We express it in a completely standard way.

Lemma 6 (Type transitivity)**.** $P \vdash \tau \prec \tau' \ \wedge \ P \vdash \tau' \prec \tau'' \implies P \vdash \tau \prec \tau''$

Proof. Using the definition of the subtype relation (§3.4.3) and a single application of Lemma 2. □

3.5 Type checking

To prevent type errors, every LJ program is typechecked before it executes. We define type checking in terms of well-formedness relations on the structures in an LJ program. First, we present some lookup functions used in these relations, and then the relations themselves. The definitions of these functions and relations are fairly standard, which is why we (again) recommend a quick scan-through only, on a first reading.

3.5.1 Lookup functions

This subsection shows the lookup functions (directly and indirectly) used within the well-formedness rules (§3.5.2), apart from those that were introduced previously (§3.3.2). The functions are presented using the rules described in §1.6.

Getting method names of a class

Obtaining names of methods of a class is done in the same manner as for fields: we obtain the inheritance path for a class, then scan through each class definition for method definitions. The function **class_methods** simply extracts the method definitions from the given class definition. The following function is used when defining class well-formedness.

```
methods (P, τ)  :  (P × τ) → meth_opt  =
  match find_path (P, τ) with
    None → None | Some ctxcld → Some (methods_in_path (ctxcld))
```

```
methods_in_path (cld)  :  cld → meth  =
  match cld with [] → [] | cld :: cld' →
    (map (λ(cl meth (vd) { meth_body }) → meth)(class_methods (cld)))  @
      methods_in_path (cld')
```

Looking up field types

Field type lookups are necessary when defining heap and statement well-formedness rules. To look up a field type for a specific field, f defined by some type, τ, we first determine the type's inheritance path, and search through it linearly.

```
ftype (P, τ, f)  :  (P × τ × f) → τ_opt  =
  match find_path (P, τ) with
    None → None | Some ctxcld → Some (ftype_in_path (ctxcld, f))
```

Then, a linear search through the inheritance path looks at each class in turn, starting with subclasses. It locates the field declarations of each class, and searches those linearly. If f is found in a class definition, but its type could not have been determined (**ftype_in_fds** below returned $\bot$), the function fails immediately, ignoring unexplored class definitions — because all well-formed classes use only valid field types (shown later in §3.5.2, page 74), this does not detect any errors that would otherwise not be found.

```
ftype_in_path (P, ctxcld, f)  :  (P × ctxcld × f) → τ_opt  =
  match ctxcld with [] → None | ((ctx, cld) :: ctxcld') →
  match ftype_in_fds (P, ctx, class_fields (cld), f) with
    ⊥ → None | Some τ → Some τ | None → ftype_in_path (P, ctxcld', f)
```

Finally, field declarations of a specific class are checked in the order they are defined — as shown later in §3.5.2 (page 74), field names within a class must be distinct, which implies that 'if there is a field with a matching name, it is the only such field within the field declarations of a class.' If no field with name f is found, the function fails by returning None. If a field, f, is found, but its type cannot be determined, the function aborts by returning $\bot$. Otherwise, the type found is returned.

```
ftype_in_fds (P, ctx, fd-bar, f) : (P × ctx × fd-bar × f) → τ_opt^⊥ =
  match fd-bar with [] → None | (cl f' ;) :: fd-bar' →
  if f = f' then match find_type (P, ctx, cl) with None → ⊥ | Some τ → Some τ
  else ftype_in_fds (P, ctx, fd-bar, f)
```

Method type lookup

Method type lookups are required for class and statement well-formedness rules. To find a method type corresponding to a method name, $meth$, within a type, τ, we first locate the corresponding method definition. Then, we find types for all class references in the method's signature: $\overline{\tau}$ for method parameter types, and τ' for method's return type. The function below is used for lifting successful method parameter type lookups.

```
mtype (P, τ, meth) : (P × τ × meth) → π_opt =
  match find_meth_def (P, τ, meth) with
    None → None | Some (ctx, cl meth (vd-bar) { meth_body }) →
      match find_type (P, ctx, cl) with None → None | Some τ' →
      match lift_opts (map (λ(cl var) → find_type (P, ctx, cl)) vd-bar) with
        None → None | Some τ-bar → Some (τ-bar → τ')
```

```
lift_opts (opts) : α_opt list → (α list)_opt =
  match opts with [] → Some [] | (opt :: opts') →
  match opt with None → None | Some v →
  match lift_opts (opts') with None → None | Some vs → Some (v :: vs)
```

3.5.2 Well-formedness rules

Here, we present the well-formedness relations in a top-down, depth-first fashion. Most are fairly straightforward.

Program well-formedness ($\vdash P$)

A program, P, is well-formed *iff* class names bound in P are "distinct," and all its class definitions are well-formed and contain no cycles.

$$\frac{\begin{array}{l}1.\ P = \overline{cld_k}^{\,k} \\ 2.\ \mathbf{distinct\_names}\,(P) \\ 3.\ \overline{P \vdash cld_k}^{\,k} \\ 4.\ \mathbf{acyclic\_clds}\,P\end{array}}{\vdash P}\ \text{WF\_PROGRAM}$$

Distinct names ($\mathbf{distinct\_names}\,(P)$)

Class names bound in P are "distinct" when the names of *all* class definitions in P are distinct from one other,[5] which is important to guarantee non-ambiguity for type references.

$$\frac{1.\ P = \overline{cld_k}^{\,k} \quad 2.\ \overline{\mathbf{class\_name}\,(cld_k) = dcl_k}^{\,k} \quad 3.\ \mathbf{distinct}\,(\overline{dcl_k}^{\,k})}{\mathbf{distinct\_names}\,(P)}\ \text{DN\_DEF}$$

Class well-formedness ($P \vdash cld$)

The following rule defers most checks to the rule below, but also checks that the mentioned class definition is in P. Note that dcl is a name of a derived class, while cl can also be `Object` — the definition is shown in Fig. 3.1 (page 59).

$$\frac{1.\ \mathbf{class}\ dcl\ \mathbf{extends}\ cl\ \{\,\overline{fd}\ \overline{meth\_def}\,\} \in P \quad 2.\ P \vdash (dcl, cl, \overline{fd}, \overline{meth\_def})}{P \vdash \mathbf{class}\ dcl\ \mathbf{extends}\ cl\ \{\,\overline{fd}\ \overline{meth\_def}\,\}}\ \text{WF\_CLASS}$$

Class well-formedness – common ($P \vdash_{ctx} (dcl, cl, \overline{fd}, \overline{meth\_def})$)

Below is a rule, which defines when a class definition (of program P found in context ctx) is well-formed — WF_CLASS_COMMON. The rule abstracts away from the user syntax of a class, which makes it applicable in many language extensions. It ensures that (1) the name of the superclass refers to a valid type,[6] (2) the type of the superclass τ is different from the type of this class, '$ctx.dcl$', (3) all field names are distinct, (4–5) fields of the superclass are name-wise distinct from the fields being defined (no field shadowing), (6) class names of fields being defined refer to valid types, (7) method definitions are well-formed (in the context of the current type), (8–13\9) if the current class is overriding a method in the superclass, then it must preserve its method type (method overriding),[7] and (9) methods' names must be distinct from each other. The function $\mathbf{method\_name}$ simply extracts a method name from a method definition.

[5] This is not true when we introduce modules (Chapter 4).

[6] To determine if a type is *valid* (§3.4.4), we are required to check that the whole inheritance path for a particular class name exists. When a program is checked for well-formedness, this judgement must hold for all its class definitions; therefore, all the links between classes are checked, which is why finding a type ($\mathbf{find\_type}$) is enough to show that the type is valid.

[7] Even though the return types could be co-variant and the argument types contra-variant, we use Java's conservative approach of type invariance for both in order to remain compatible with Java's semantics.

$$\frac{\begin{array}{l}1.\ \mathbf{find\_type}\,(P,\ ctx,\ cl) = \tau \\ 2.\ ctx.dcl \neq \tau \qquad 3.\ \mathbf{distinct}\,(\overline{f_j}^{\,j}) \\ 4.\ \mathbf{fields}\,(P,\ \tau) = \overline{f} \qquad 5.\ \overline{f_j}^{\,j} \perp \overline{f} \\ 6.\ \overline{\mathbf{find\_type}\,(P,\ ctx,\ cl_j) = \tau_j}^{\,j} \\ 7.\ \overline{P \vdash_{ctx.dcl} meth\_def_k}^{\,k} \\ 8.\ \overline{\mathbf{method\_name}\,(meth\_def_k) = meth_k}^{\,k} \\ 9.\ \mathbf{distinct}\,(\overline{meth_k}^{\,k}) \\ 10.\ \mathbf{methods}\,(P,\ \tau) = \overline{meth'_l}^{\,l} \\ 11.\ \overline{\mathbf{mtype}\,(P,\ ctx.dcl,\ meth'_l) = \pi_l}^{\,l} \\ 12.\ \overline{\mathbf{mtype}\,(P,\ \tau,\ meth'_l) = \pi'_l}^{\,l} \\ 13.\ \overline{meth'_l \in \overline{meth_k}^{\,k} \longrightarrow \pi_l = \pi'_l}^{\,l}\end{array}}{P \vdash_{ctx} (dcl, cl, \overline{cl_j\ f_j\ ;}^{\,j}, \overline{meth\_def_k}^{\,k})}\ \text{WF\_CLASS\_COMMON}$$

Method definition well-formedness ($P \vdash_\tau meth\_def$)

A method definition (defined within the type '$ctx.dcl$') is well-formed *iff* (1) parameter names are distinct from each other, (2) class names of parameters refer to valid types, (3–4) the statements of the method are well-formed in a type environment where parameters map to corresponding types, and **this** maps to the owner of the method, (5) the class name for the method's return type refers to a valid type, and (6) the type of the variable returned by the method's body is a subtype of the return type.

$$\frac{\begin{array}{l}1.\ \mathbf{distinct}\,(\overline{var_k}^{\,k}) \\ 2.\ \overline{\mathbf{find\_type}\,(P,\ ctx,\ cl_k) = \tau_k}^{\,k} \\ 3.\ \Gamma = [\overline{var_k \mapsto \tau_k}^{\,k}]\,[\mathbf{this} \mapsto ctx.dcl] \\ 4.\ \overline{P,\ \Gamma \vdash s_l}^{\,l} \\ 5.\ \mathbf{find\_type}\,(P,\ ctx,\ cl) = \tau \\ 6.\ P \vdash \Gamma\,(y) \prec \tau\end{array}}{P \vdash_{ctx.dcl} cl\ meth\ (\overline{cl_k\ var_k}^{\,k})\ \{\ \overline{s_l}^{\,l}\ \mathbf{return}\ y\ ;\ \}}\ \text{WF\_METHOD}$$

Statement well-formedness ($P,\ \Gamma \vdash s$)

For a block of statements, '$\{\ \overline{s_k}^{\,k}\ \}$', to be well-formed, each statement must be well-formed. For variable assignment, '$var = x\ ;$', the type of x must be a subtype of the type of var. For field reading, '$var = x\,.f\ ;$', the type of f must be a subtype of the type of var. For field writing, '$x\,.f = y\ ;$', the type of y must be a subtype of the type of f. For the if statement, '$\mathbf{if}\ (x == y)\ s_1\ \mathbf{else}\ s_2$', the type of x must be a subtype of the type of y, or vice-versa, and both statements must be well-formed. For object creation, '$var = \mathbf{new}_{ctx}\,cl();$', the type of cl must be a subtype of the type of var. For the method call, '$var = x\,.\,meth\,(\overline{y})\ ;$', the types of the arguments must be element-wise subtypes of

the types of the parameters, and the return type of the method must be a subtype of the type of var.[8]

$$\text{WF\_BLOCK}\ \frac{1.\ \overline{P,\ \Gamma \vdash\ s_k}^{\,k}}{P,\ \Gamma \vdash\ \{\ \overline{s_k}^{\,k}\ \}} \qquad \text{WF\_VAR\_ASSIGN}\ \frac{1.\ P \vdash\ \Gamma\,(x) \prec \Gamma\,(var)}{P,\ \Gamma \vdash\ var = x\,;}$$

$$\text{WF\_FIELD\_READ}\ \frac{\begin{array}{l}1.\ \Gamma\,(x) = \tau \\ 2.\ \mathbf{ftype}\,(P,\ \tau,\ f) = \tau' \\ 3.\ P \vdash\ \tau' \prec \Gamma\,(var)\end{array}}{P,\ \Gamma \vdash\ var = x\,.\,f\,;}$$

$$\text{WF\_FIELD\_WRITE}\ \frac{\begin{array}{l}1.\ \Gamma\,(x) = \tau \\ 2.\ \mathbf{ftype}\,(P,\ \tau,\ f) = \tau' \\ 3.\ P \vdash\ \Gamma\,(y) \prec \tau'\end{array}}{P,\ \Gamma \vdash\ x\,.\,f = y\,;} \qquad \text{WF\_IF}\ \frac{\begin{array}{l}1.\ P \vdash\ \Gamma\,(x) \prec \Gamma\,(y)\ \vee\ P \vdash\ \Gamma\,(y) \prec \Gamma\,(x) \\ 2.\ P,\ \Gamma \vdash\ s_1\ 3.\ P,\ \Gamma \vdash\ s_2\end{array}}{P,\ \Gamma \vdash\ \mathbf{if}\,(x == y)\ s_1\ \mathbf{else}\ s_2}$$

$$\text{WF\_NEW}\ \frac{\begin{array}{l}1.\ \mathbf{find\_type}\,(P,\ ctx,\ cl) = \tau \\ 2.\ P \vdash\ \tau \prec \Gamma\,(var)\end{array}}{P,\ \Gamma \vdash\ var = \mathbf{new}_{ctx}\,cl();} \qquad \text{WF\_MCALL}\ \frac{\begin{array}{l}1.\ \overline{y} = \overline{y_k}^{\,k}\ \ 2.\ \Gamma\,(x) = \tau \\ 3.\ \mathbf{mtype}\,(P,\ \tau,\ meth) = \overline{\tau_k}^{\,k}\ \rightarrow\ \tau' \\ 4.\ \overline{P \vdash\ \Gamma\,(y_k) \prec \tau_k}^{\,k} \\ 5.\ P \vdash\ \tau' \prec \Gamma\,(var)\end{array}}{P,\ \Gamma \vdash\ var = x\,.\,meth\,(\overline{y})\,;}$$

Note how the ctx in the object creation statement, '$var = \mathbf{new}_{ctx}\,cl();$', is used to look up the type, τ, in WF_NEW. By simply passing the context through to the lookup function, the rule stays independent of the inner structure of context — this means that languages with different definitions of ctx can still reuse all of the above rules.

3.6 Proof of type soundness

In this section, we make sure that the type system and the well-formedness rules do not contain errors. We then use this result to guarantee that no LJ program will fail due to a type error if it satisfies the well-formedness rules (§3.5).

The well-formedness rules must be strong enough to ensure that the next reduction step can execute successfully. At the same time, the statement reduction rules must respect the well-formedness rules, so that the resulting configuration remains well-formed.

We first describe the conditions for configuration well-formedness, and then prove that our type system and well-formedness rules are error-free, i.e. we prove *type soundness*.

[8]The first premise for WF_FIELD_READ (and similar premises) cannot be inlined, since it is a lookup that lifts an option type to a bare type, which is used as such in the next premise. Since option type lifts are often required for type comparison (subtyping), we created a judgement that lifts both option types (if required), and (if lifting succeeds) compares the two resulting types — an example of this is the premise for WF_VAR_ASSIGN. The first premise of the WF_MCALL is there to make the Ott's Isabelle/HOL easier to use.

The proof is done in the usual way: by proving progress and type preservation properties. The following proofs are the natural language versions of our Isabelle/HOL proofs.

3.6.1 Configuration well-formedness ($\Gamma \vdash \mathit{config}$)

The concept of configuration well-formedness, '$\Gamma \vdash \mathit{config}$', is important for proving type soundness. This subsection defines the property.

A configuration (defined in §3.3.1, page 61) that has reached an exception is well-formed if the program, P, the variable state, L, the heap, H, are well-formed. For a normal configuration, the statements yet to be executed, $\overline{s_k}^k$, must also be well-formed.

WF_ALL_EX

$$\frac{1.\ \vdash P \quad 2.\ P \vdash H \quad 3.\ P,\ \Gamma,\ H \vdash L}{\Gamma \vdash (P,\ L,\ H,\ \mathit{Exception})}$$

WF_ALL

$$\frac{1.\ \vdash P \quad 2.\ P \vdash H \quad 3.\ P,\ \Gamma,\ H \vdash L \quad 4.\ \overline{P,\ \Gamma \vdash s_k}^k}{\Gamma \vdash (P,\ L,\ H,\ \overline{s_k}^k)}$$

In fact, LJ and our extensions of it are all type sound even without the three premises in WF_ALL_EX. This is because (1) the program terminates whenever an exception is thrown, since we do not implement exception handling, and (2) the proof of type soundness does not rely on the consistency between well-formedness rules for the intermediate configurations and well-formedness rules for the final configurations.

The heap, in turn, is well-formed *iff* it has a finite domain, and all field values stored in the heap have types that are element-wise subtypes of the corresponding field types.

WF_HEAP

$$\frac{1.\ \mathbf{finite}\,(\mathbf{dom}\,(H)) \qquad 2.\ \forall oid \in \mathbf{dom}\,(H)\,.\ \left(\begin{array}{l}\exists \tau\,.\,H\,(oid) = \tau\ \wedge\ \exists \overline{f}\,.\,\mathbf{fields}\,(P,\,\tau) = \overline{f}\ \wedge \\ \quad \forall f \in \overline{f}\,.\,\exists \tau'\,.\,\left(\begin{array}{l}\mathbf{ftype}\,(P,\,\tau,\,f) = \tau'\ \wedge \\ P,\ H\ \vdash\ H\,(oid,\,f) \prec \tau'\end{array}\right)\end{array}\right)}{P \vdash H}$$

A variable state, on the other hand, is well-formed *iff* it has a finite domain, and all variables in the type environment[9] have types that are supertypes of the types of values associated with them.

WF_VARSTATE

$$\frac{1.\ \mathbf{finite}\,(\mathbf{dom}\,(L)) \qquad 2.\ \forall x \in \mathbf{dom}\,(\Gamma)\,.\ P,\ H\ \vdash\ L\,(x) \prec \Gamma\,(x)}{P,\ \Gamma,\ H\ \vdash\ L}$$

[9]Not 'all variables in the variable state.' This subtlety is explained in the proof of progress (§3.6.3).

The second premise in the above rule is, in fact, not a subtyping relation, since it compares a *value* and a type. It refers to the *value subtyping relation* defined by the two rules below: WF_NULL says that **null**'s type is a subtype of any type, while WF_OBJECT states that for an object identifier, oid, to be a value subtype of a type, the type associated with oid in the heap, H, must be a subtype of that type — this also implies that oid has a valid mapping in the heap.

$$\frac{1.\ \tau_{opt} = \tau}{P,\ H \vdash \mathbf{null} \prec \tau_{opt}}\ \text{WF\_NULL} \qquad \frac{1.\ P \vdash H\,(oid) \prec \tau_{opt}}{P,\ H \vdash oid \prec \tau_{opt}}\ \text{WF\_OBJECT}$$

The domains of the variable state and the heap need to be finite to ensure that there are always fresh identifiers for each in case of method calls and object creations, respectively.

3.6.2 Helper lemmas

To prove the progress and the type preservation properties, we use many helper lemmas, the most important of which are described in this subsection.

Lemma 7 (No private fields)**.**

$$\vdash P \quad \wedge \quad P \vdash \tau \prec \tau' \quad \wedge \quad \mathbf{ftype}\,(P,\, \tau',\, f) = \tau'' \quad \wedge \quad \mathbf{fields}\,(P,\, \tau) = \overline{f} \quad \Longrightarrow \quad f \in \overline{f}$$

Explanation. If a program, P, is well-formed, τ is a subtype of τ', τ' defines a field f of type τ'', and the fields of τ are $\overline{f}$, then f must be within $\overline{f}$.

Proof. The $\mathbf{ftype}$ function (page 72) is defined in terms of $\mathbf{find\_path}$ (page 63), which is in turn defined by $\mathbf{find\_path\_rec}$ (page 62). In the proof, we unfold the definition of $\mathbf{ftype}$ to $\mathbf{find\_path\_rec}$, and use a well-founded induction to show that '$f \in \overline{f}'$', where '$\mathbf{fields}\,(P,\, \tau') = \overline{f}'$'. Then, we use the definition of subtyping (page 69) and $\mathbf{find\_path\_rec}$ to show that the inheritance path for τ' is a suffix of the one for τ. Finally, we unfold the definition of $\mathbf{fields}$ (page 64) to $\mathbf{fields\_in\_path}$ (page 64), and show by structural induction on the inheritance path that '$f \in \overline{f}$'. □

Lemma 8 (Field type preservation)**.**

$$\vdash P \quad \wedge \quad P \vdash \tau \prec \tau' \quad \wedge \quad \mathbf{ftype}\,(P,\, \tau',\, f) = \tau'' \quad \Longrightarrow \quad \mathbf{ftype}\,(P,\, \tau,\, f) = \tau''$$

Explanation. If a program, P, is well-formed, τ is a subtype of τ', and the type of field f within τ' is τ'', then the type of a field, f, within τ must also be τ''.

Proof. First, we unfold the definition of $\mathbf{ftype}$ to get inheritance paths for both τ and τ'. Then, we use the definition of subtyping and $\mathbf{find\_path\_rec}$ to show that the inheritance path for τ' is a suffix of the one for τ. Through program well-formedness

(page 73), we show that every class definition in either inheritance path must be well-formed. Finally, we induct over the derivation of $\mathbf{find\_path\_rec}$, using premises (4–5) of the WF_CLASS_COMMON (page 74) to show the preservation of field types. □

Lemma 9 (Method type preservation)**.**

$$\vdash P \quad \wedge \quad P \vdash \tau \prec \tau' \quad \wedge \quad \mathbf{mtype}\,(P,\, \tau,\, meth) = \pi \quad \wedge \quad \mathbf{mtype}\,(P,\, \tau',\, meth) = \pi' \quad \Longrightarrow \quad \pi = \pi'$$

Explanation. If a program, P, is well-formed, τ is a subtype of τ', the method type of method named $meth$ within τ is π, the method type of method named $meth$ within τ' is π', then π must be equal to π'.

Proof. First, we unfold the definition of $\mathbf{mtype}$ (page 73) to get inheritance paths for both τ and τ'. Then, we use the definition of subtyping and $\mathbf{find\_path\_rec}$ to show that the inheritance path for τ' is a suffix of the one for τ. Through program well-formedness, we show that every class definition in either inheritance path must be well-formed. Finally, we induct over the derivation of $\mathbf{find\_path\_rec}$, using premises (8–12) of the WF_CLASS_COMMON to show the preservation of method types. □

Lemma 10 (Method type to method definition)**.**

$$\vdash P \quad \wedge \quad \mathbf{mtype}\,(P,\, \tau,\, meth) = \pi \quad \wedge \quad P \vdash \tau' \prec \tau \quad \Longrightarrow \quad \exists ctx \;\; meth\_def.\;\; \mathbf{find\_meth\_def}\,(P,\, \tau',\, meth) = (ctx,\, meth\_def)$$

Explanation. If a program, P, is well-formed, τ defines a method named $meth$ (of some type π), and τ' is a subtype of τ, then τ' must define a method named $meth$.

Proof. By using Lemma 9 to obtain '$\mathbf{mtype}\,(P,\, \tau',\, meth) = \pi$', then unfolding the definition of $\mathbf{mtype}$. □

3.6.3 Progress

The progress property guarantees that an LJ program will not get stuck if there are statements left to execute. There is nothing particularly special about this proof of progress.

Theorem 11 (Progress)**.**

$$\Gamma \vdash (P, L, H, \overline{s}) \;\wedge\; \overline{s} \neq [] \;\Longrightarrow\; \exists \mathit{config}.\; (P, L, H, \overline{s}) \longrightarrow \mathit{config}$$

Explanation. If a configuration, '$(P, L, H, \overline{s})$', is well-formed in some type environment, Γ, and there are still some statements, $\overline{s}$, left to execute, then there exists some configuration, $config$, which the current configuration reduces to in one step, $\longrightarrow$.

Proof. By case splitting on the next statement to execute, s. Due to WF_ALL (page 77), we know that '$P,\, \Gamma \vdash s$', i.e. that s is well-formed. We then consider each case:

1. $s = (\{\, \overline{s_k}^{\,k} \,\})$: application of R_BLOCK (statement reduction rules are on page 65).

2. $s = (var = x\,;)$: from WF_VAR_ASSIGN (statement well-formedness rules are on page 76), we know that '$x \in \mathbf{dom}\,(\Gamma)$'; then, we know from WF_VARSTATE (page 77) that '$x \in \mathbf{dom}\,(L)$';[10] it trivially follows that '$\exists v.\ L\,(x) = v$', with which we can apply R_VAR_ASSIGN.

3. $s = (var = x\,.\,f\,;)$: from WF_FIELD_READ, we know that '$x \in \mathbf{dom}\,(\Gamma)$' (from '$\Gamma\,(x) = \tau'$') and '$\mathbf{ftype}\,(P,\, \tau',\, f) = \tau''$'; from WF_VARSTATE, we then know that '$x \in \mathbf{dom}\,(L)$'; if '$L\,(x) = \mathbf{null}$', then we can apply R_FIELD_READ_NPE; otherwise, we know that '$\exists oid.\ L\,(x) = oid$'; from WF_HEAP (page 77) and WF_OBJECT (page 78) it follows that '$oid \in \mathbf{dom}\,(H)$', '$\exists \tau.\ H\,(oid) = \tau,\ P \vdash \tau \prec \Gamma\,(x)$', and that '$\exists \overline{f}.\ \mathbf{fields}\,(P,\, \tau) = \overline{f}$'; we can then use Lemma 7 (page 78) to show that '$f \in \overline{f}$'; with WF_HEAP, we can then show that '$\exists \tau'''.\ P,\ H \vdash H\,(oid,\, f) \prec \tau'''$', from which we know '$\exists v.\ H\,(oid,\, f) = v$', with which we can apply R_FIELD_READ.

4. $s = (x\,.f = y\,;)$: from WF_FIELD_WRITE, we know that '$x \in \mathbf{dom}\,(\Gamma)$' (from '$\Gamma\,(x) = \tau$') and '$y \in \mathbf{dom}\,(\Gamma)$' (from '$P \vdash \Gamma\,(y) \prec \tau'$'); from WF_VARSTATE, we then know that '$x \in \mathbf{dom}\,(L)$' and '$y \in \mathbf{dom}\,(L)$'; if '$L\,(x) = \mathbf{null}$', then we can apply R_FIELD_WRITE_NPE; otherwise, we know that '$\exists oid.\ L\,(x) = oid$', so we can apply R_FIELD_WRITE.

5. $s = (\mathbf{if}\,(x == y)\,s_1\,\mathbf{else}\,s_2)$: from WF_IF, we know that '$x \in \mathbf{dom}\,(\Gamma)$' and '$y \in \mathbf{dom}\,(\Gamma)$' (from '$P \vdash \Gamma\,(x) \prec \Gamma\,(y) \ \vee\ P \vdash \Gamma\,(y) \prec \Gamma\,(x)$'); from WF_VARSTATE, we then know '$x \in \mathbf{dom}\,(L)$' and '$y \in \mathbf{dom}\,(L)$', which means that '$\exists v.\ L\,(x) = v$' and '$\exists w.\ L\,(y) = w$'; if '$v = w$', then we can apply R_IF_TRUE; otherwise, we can apply R_IF_FALSE.

6. $s = (var = \mathbf{new}_{ctx}\, cl();)$: from WF_NEW, we know that '$P \vdash \tau \prec \Gamma\,(var)$' (where '$\mathbf{find\_type}\,(P,\, ctx,\, cl) = \tau$'); from Lemma 1 (page 69), we know that τ is a valid type; since every valid type has a valid class hierarchy, we know that '$\exists \overline{f_k}^{\,k}.\ \mathbf{fields}\,(P,\, \tau) = \overline{f_k}^{\,k}$'; from WF_HEAP, we know '$\mathbf{finite}\,(\mathbf{dom}\,(H))$', from which we can deduce that '$\exists oid.\ oid \notin \mathbf{dom}\,(H)$'; finally, we can apply R_NEW.

7. $s = (var = x\,.\,meth\,(\overline{y_k}^{\,k})\,;)$: from WF_MCALL (page 76), we know that '$x \in \mathbf{dom}\,(\Gamma)$' (from '$\Gamma\,(x) = \tau$'); from WF_VARSTATE (page 77), we then know that '$x \in \mathbf{dom}\,(L)$'; if '$L\,(x) = \mathbf{null}$', then we can apply R_MCALL_NPE; otherwise, we know that '$\exists oid.\ L\,(x) = oid$'; from WF_HEAP (page 77) and WF_OBJECT (page 78) it follows that '$oid \in \mathbf{dom}\,(H)$', '$\exists \tau'.\ H\,(oid) = \tau'$', and '$P \vdash \tau' \prec$

[10] It is because of steps like this that we need '$\forall x \in \mathbf{dom}\,(\Gamma)$' and not '$\forall x \in \mathbf{dom}\,(L)$' in WF_VARSTATE (page 77), otherwise we would not be able to deduce that '$x \in \mathbf{dom}\,(L)$'.

τ'; from WF_MCALL, we also know '$\exists \pi.\ \mathbf{mtype}\,(P,\ \tau,\ meth)\ =\ \pi$' — using Lemma 10 (page 79), we show '$\exists ctx\ meth\_def.\ \mathbf{find\_meth\_def}\,(P,\ \tau',\ meth) = (ctx,\ meth\_def)$'; from WF_VARSTATE, we also have fresh and distinct variables we can use for the method flattening process; finally, we can apply R_MCALL. □

As mentioned before, the proofs presented here are the natural language versions of the corresponding mechanically-verified Isabelle/HOL proofs. The original progress proof (excluding helper lemmas) is shown in Appendix B; the rest can be found online.

3.6.4 Type preservation

The type preservation property guarantees preservation of configuration well-formedness through execution. There is a subtlety in the method call case of the proof, which we describe right after the main text (page 83). Apart from this, the proof is fairly standard for a Java-like language.

Theorem 12 (Type preservation)**.**

$$\Gamma \vdash \mathit{config}\ \wedge\ \mathit{config} \longrightarrow \mathit{config}'\ \Longrightarrow\ \exists \Gamma'.\ \Gamma \subseteq_m \Gamma'\ \wedge\ \Gamma' \vdash \mathit{config}'$$

Explanation. If *config* is a well-formed configuration in some type environment, Γ, and *config* reduces to *config*$'$ through statement reduction, $\longrightarrow$, then *config*$'$ is well-formed in some greater[11] ($\subseteq_m$[12]) type environment, Γ'.

Proof. By case splitting on the statement reduction relation. In each of the eleven cases, except in R_MCALL, the type environment Γ' is equal to Γ. Due to WF_ALL (page 77), we know that all elements of the initial configuration, *config*, are well-formed. The configuration, *config*, is unpacked into '$(P,\ L,\ H,\ s\ \overline{s_k}^{\,k})$', where s is the statement just executed. We then consider each reduction rule:

1. R_BLOCK, $s = (\{\ \overline{s_k}^{\,k}\ \})$: by WF_BLOCK (statement well-formedness rules are on page 76), *config*$'$ is trivially well-formed with WF_ALL.
2. R_VAR_ASSIGN, $s = (var = x\,;)$: we must show '$P,\ H\ \vdash\ L\,[var\ \mapsto\ v]\,(var) \prec \Gamma\,(var)$', or equivalently '$P,\ H\ \vdash\ v \prec \Gamma\,(var)$', where '$L\,(x) = v$'; if '$v = \mathbf{null}$', WF_NULL applies; otherwise, '$\exists oid.\ v = oid$'; using WF_VARSTATE and WF_HEAP, we know '$\exists \tau.\ H\,(oid) = \tau$' (which simplifies our goal to '$P\ \vdash\ \tau \prec \Gamma\,(var)$') and '$P\ \vdash\ \tau \prec \Gamma\,(x)$'; from WF_VAR_ASSIGN, we know that '$P\ \vdash\ \Gamma\,(x) \prec \Gamma\,(var)$'; finally, we can use type transitivity (Lemma 6) to show '$P\ \vdash\ \tau \prec \Gamma\,(var)$'.

[11] The type preservation property does not require the type environment to grow; this is simply a consequence of our semantics, which we prove here formally.

[12] Since Γ is a map, we need the sub-map relation. The symbol was borrowed from Isabelle/HOL.

3. R_FIELD_READ_NPE, $s = (var = x\,.\,f\,;)$: WF_ALL_EX (page 77) trivially applies.
4. R_FIELD_READ, $s = (var = x\,.\,f\,;)$: we must show '$P,\ H \vdash L\,[var \mapsto v]\,(var) \prec \Gamma\,(var)$', or equivalently '$P,\ H \vdash v \prec \Gamma\,(var)$', where '$L\,(x) = oid$', and where '$H\,(oid,\ f) = v$'; if '$v = \mathbf{null}$', WF_NULL applies; otherwise, we know '$\exists oid'.\ v = oid'$'; using WF_HEAP and WF_VARSTATE, we know '$\exists \tau_1.\ H\,(oid') = \tau_1$', which simplifies our goal to '$P \vdash \tau_1 \prec \Gamma\,(var)$'; from WF_FIELD_READ, we know '$\exists \tau_2.\ \Gamma\,(x) = \tau_2$', '$\mathbf{ftype}\,(P,\ \tau_2,\ f) = \tau_3$', and '$P \vdash \tau_3 \prec \Gamma\,(var)$'; with WF_HEAP and WF_VARSTATE, we know '$\exists \tau_4.\ H\,(oid) = \tau_4$', '$\exists \tau_5.\ \mathbf{ftype}\,(P,\ \tau_4,\ f) = \tau_5$', and '$P \vdash \tau_1 \prec \tau_5$'; using Lemma 8 (page 78), we show that '$\tau_3 = \tau_5$'; finally, we use Lemma 6 (page 71) to show that '$P \vdash \tau_1 \prec \Gamma\,(var)$'.
5. R_FIELD_WRITE_NPE, $s = (x\,.\,f = y\,;)$: WF_ALL_EX trivially applies.
6. R_FIELD_WRITE, $s = (x\,.\,f = y\,;)$: from WF_FIELD_WRITE, we know '$\exists \tau_1.\ \Gamma\,(x) = \tau_1$', '$\mathbf{ftype}\,(P,\ \tau_1,\ f) = \tau_2$', and '$P \vdash \Gamma\,(y) \prec \tau_2$'; we must show that the resulting heap, '$H\,[(oid,\ f) \mapsto v]$', is well-formed, where '$L\,(x) = oid$' and '$L\,(y) = v$'; according to WF_HEAP, this amounts to showing that '$P,\ H \vdash v \prec \tau_4$', where '$H\,(oid) = \tau_3$' and '$\mathbf{ftype}\,(P,\ \tau_3,\ f) = \tau_4$'; if '$v = \mathbf{null}$', WF_NULL applies; otherwise, we know that '$\exists oid'.\ v = oid'$'; using WF_VARSTATE and WF_HEAP, we know that '$\exists \tau_5.\ H\,(oid') = \tau_5$', which simplifies our goal to '$P \vdash \tau_5 \prec \tau_4$'; using Lemma 8 (page 78), we know that '$\tau_2 = \tau_4$'; from WF_VARSTATE, we know that '$P \vdash \tau_5 \prec \Gamma\,(y)$'; finally, we use Lemma 6 (page 71) to show that '$P \vdash \tau_5 \prec \tau_4$'.
7. R_IF_TRUE, $s = (\mathbf{if}\,(x == y)\,s_1\,\mathbf{else}\,s_2)$: by WF_IF, *config'* is trivially well-formed with WF_ALL.
8. R_IF_FALSE, $s = (\mathbf{if}\,(x == y)\,s_1\,\mathbf{else}\,s_2)$: by WF_IF, *config'* is trivially well-formed with WF_ALL.
9. R_NEW, $s = (var = \mathbf{new}_{ctx}\,cl();)$: goals are '$P,\ H\,[oid \mapsto (\tau,\ \overline{f_k \mapsto \mathbf{null}}^{\,k})] \vdash oid \prec \Gamma\,(var)$' and '$P,\ H \vdash H\,(oid,\ f) \prec \tau'$', where oid is the object identifier pointing to the newly created object, τ its type, f any field of τ, and τ' the corresponding type of that field; using WF_VARSTATE, the first goal reduces to '$P \vdash \tau \prec \Gamma\,(var)$', which trivially follows from WF_NEW (page 76); since all field values are initialised to $\mathbf{null}$, the second goal reduces to '$P,\ H \vdash \mathbf{null} \prec \tau'$' for all fields, which holds by WF_NULL (page 78).
10. R_MCALL_NPE, $s = (var = x\,.\,meth\,(\overline{y_k}^{\,k})\,;)$: WF_ALL_EX trivially applies.
11. R_MCALL, $s = (var = x\,.\,meth\,(\overline{y_k}^{\,k})\,;)$: from WF_MCALL (page 76) and R_MCALL (page 65), we know that '$L\,(x) = oid$', a parameter variable var_k has type τ_k, var'_k is a fresh substitute for var_k, and x' is a fresh name replacing $\mathbf{this}$; Γ' is therefore instantiated to '$\Gamma[\overline{var'_k \mapsto \tau_k}^{\,k}]\,[x' \mapsto \tau_1]$', where τ_1 is the type where the method is defined; by unfolding the definition of $\mathbf{find\_meth\_def}$ (page 64) to $\mathbf{find\_path}$

(page 63), and then using a variant of structural induction over inheritance path in $\mathbf{find\_meth\_def\_path}$ (page 64) and Lemma 2 (page 70), we show that τ_1 must be a supertype of $H\,(oid)$, the runtime type of the object on which the method was called, and, together with the program well-formedness (page 73), that the method called must be well-formed (page 75); the variables in the method body are translated according to the translation context, θ, where '$\theta = [\overline{var_k \mapsto var'_k}^{\,k}]\,[\mathbf{this} \mapsto x']$', so we must show well-formedness of the translated method body, $\overline{s''_j}^{\,j}$, within the new type environment, Γ', i.e. '$\overline{P,\, \Gamma' \vdash s''_j}^{\,j}$' — this is done by rule induction on the statement well-formedness relation (page 76), which involves tedious proofs of well-formedness preservation through translation; we must also show the well-formedness of the generated assignment, '$var = y'\,;$' (where y' is the translated version of the variable y the method returns), in Γ', i.e. '$P, \Gamma' \vdash var = y'\,;$', which holds when '$P \vdash \Gamma'\,(y') \prec \Gamma'\,(var)$'; from WF_MCALL, we know that '$P \vdash \tau_2 \prec \Gamma\,(var)$', where τ_2 is the static return type of the method; from WF_METHOD, we know that '$P \vdash \Gamma\,(y) \prec \tau_2$'; since var is not re-mapped in Γ', we know that '$\Gamma'\,(var) = \Gamma\,(var)$'; also, since y is either a var_k or $\mathbf{this}$, we can show that '$\theta\,(y) = y' \implies \Gamma'\,(y') = \Gamma\,(y)$'; then, we can use Lemma 6 (page 71) to show that '$P \vdash \Gamma'\,(y') \prec \Gamma'\,(var)$'; finally, we must show the well-formedness of the new variable stack, L', where '$L' = L\,[\overline{var'_k \mapsto v_k}^{\,k}]\,[x' \mapsto oid]$', and where v_k is the value of the method argument y_k; therefore, we must show that '$P,\, H \vdash v_k \prec \Gamma'\,(var'_k)$' and '$P \vdash H\,(oid) \prec \Gamma'\,(x')$'; from WF_MCALL, we know that '$P \vdash \Gamma\,(y_k) \prec \Gamma\,(var_k)$'; from WF_VARSTATE, we know that '$P,\, H \vdash v_k \prec \Gamma\,(y_k)$'; as above, we show that '$\Gamma'\,(var'_k) = \Gamma\,(var_k)$'; we then use Lemma 6 to show that '$P,\, H \vdash v_k \prec \Gamma'\,(var'_k)$'; the second goal simplifies to '$P \vdash H\,(oid) \prec \tau_1$', which we have already shown above. □

Subtlety in the definition of the method call statement

The use of x' (in R_MCALL, Fig. 3.3, page 65) is necessary for the type-checking to go through, i.e. we could not have used the existing x instead, even though the two have identical dynamic types. This subtlety arises due to LJ's support for method overriding: a method body is well-formed in a type environment where $\mathbf{this}$ is associated with the type where the method is defined — in our case, this type is τ_1. Therefore, the method's body does not (in general) typecheck in a type environment where $\mathbf{this}$ is associated with $\Gamma\,(x)$, since $\Gamma\,(x)$ can be a supertype of τ_1. This property is preserved through variable translation. Associating x with τ_1 rather than with $H\,(oid)$ makes well-formedness preservation proofs easier, since the original statements are well-formed against τ_1, not $H\,(oid)$.

3.7 Conclusion

It is fairly easy to extend LJ. Many terms can be re-defined without affecting the consistency of the rest of the definition. Specifically, our extensions of LJ (Chapter 4 and Chapter 6) make substantial changes to the definition of a program, P, a fully-qualified name, fqn, and the program context, ctx, while reusing most of the rest. Also, since LJ's definition is written in Ott, many consistency checks are performed automatically.

LJ does not implement a frame stack. The drawbacks of this are that (a) the semantics is a bit further away from a standard implementation of an imperative language, (b) the program context needs to be pre-stored within an object creation statement, when it could otherwise have been put inside a frame of a stack, (c) the method call reduction rule (R_MCALL, page 65) has a few extra judgements due to the freshness conditions, and the translation, and (d) we need to prove statement well-formedness preservation under variable translation. However, by not having a stack, we avoid quite a bit of the semantic and proof complexity. More specifically, we avoid (i) a reduction rule and a well-formedness rule for a method return, i.e. for popping the stack, (ii) a more complex (stacked) structure of the typing environment, (iii) a few rules for stack well-formedness [12, p. 22], and (iv) proofs of lemmas about the structure of the stack and the typing environment, and their correspondence. Overall, we obtain a simpler semantics, which was our goal.

Having the semantics within Isabelle/HOL, we can perform symbolic execution. One of the possible future directions is to try to generate a reference implementation of LJ directly from its Isabelle/HOL definition using the latest code generation tools [20].

Our experience shows that small changes to the definition of LJ most often require only small changes to the Isabelle/HOL proof scripts. Type soundness of an extension can either be proven directly, or by showing that every valid program can be simplified to a valid LJ program — the latter approach was taken for LFJ [14].

4

Lightweight Java Module System (LJAM)

In the first part of the introduction (§1.1), we showed that the Java Module System (JMS) adds two key features to Java, component-level information hiding and versioning, while also focusing on easy distribution and deployment. Our overview of the system was based on our detailed analysis of the two hundred pages of draft documents, written solely in natural language and so inevitably containing many ambiguities.

The following example is taken from the draft specification [51, p. 68]:

> *"There is at most one module instance instantiated from each module definition per repository instance. A module definition can have multiple module instances through multiple repository instances."*

From the above statement, and from the rest of the document, it is not clear whether the module instances are created where they are requested, or where their module definitions are stored. Both options are entirely plausible, and since the latter seemed too restrictive to us, we were sure that the authors meant the former. We were wrong. The need to formalise the documents was obvious: to detect errors, remove ambiguities, promote precise discussion of the design, and allow properties to be (dis-)proven.

Therefore, we created mathematical entities for (what we think are) the main concepts in the draft documents, and then related these entities with formal rules according to the informal description. Where the documents were complete, we followed them closely; elsewhere we made reasonable choices. In case of ambiguities, as above, we contacted the authors of the documents for clarification.

There are two draft documents: JSR-294 [52], which outlines the developer's view, and JSR-277 [51], which describes the innerworkings of the module system. We formalise the core of the two documents in an extension of Lightweight Java (LJ), named the Lightweight Java Module System (LJAM).

More specifically, we formalise (i) the syntax for specifying member packages, exported classes, and imported modules, (ii) the semantics of the administrator actions (installing, un-installing, and initialising module definitions), and (iii) class resolution, which searches for class definitions across package, module, and repository boundaries.

However, we do not formalise versions, or any other annotation on modules — we believe their semantics to be orthogonal. Our model also excludes custom import policies, i.e. custom code responsible for selecting and initialising imported module definitions, and then linking together the resulting module instances; formalising this is an interesting option for future work, due to the restricted context of execution and possibly non-terminating initialisation. The draft documents also specify various techniques for backward compatibility, e.g. supporting JAR (Java ARchive) files; we do not model these, since they do not constitute the core of the module system, and would unnecessarily complicate our formalisation. Since the design of Java Module System is still on-going, there are some minor details for which the original design and our formalisation are now inconsistent — these are described at the end of this chapter (§4.7).

First, we give an informal description of JMS (§4.1). Then, we present the syntax, the operational semantics, the type system (§4.4), and type-checking rules (§4.5). We prove in Isabelle/HOL that the resulting formalisation is type sound (§4.6). While this chapter focuses mainly on the formalisation of the module system, the next (Chapter 5) describes the main problems of the system, and our proposals for solving them.

The full Ott definition, the complete Isabelle/HOL proof of type soundness, and various other LJAM documents can be found at the following address:

`http://www.cl.cam.ac.uk/research/pls/javasem/ljam/`

4.1 An informal description

The basic unit of the Java Module System is a *module definition*, a JAR-like archive that contains compiled Java code. A module definition is defined with (i) compiled Java code, and (ii) a module file. We can obtain (i) by compiling regular Java code with a Java compiler, e.g. `javac`, while (ii) is a text file, normally written by hand.

A module file can specify (a) which Java packages should be put into the module definition, (b) which other module definitions this module definition will import, and (c) which types (either imported, or defined locally) should be available to client modules. Therefore, a module file specifies membership, imports, and exports.

In the context of the example in §1.1.2, suppose we were defining the *XSLT* module definition. The module definition should contain compiled Java code belonging to the `xslt` Java package, import the *XMLParser* module definition, and export its own class, `xslt.Processor`. The following *module file* specifies (a) membership, (b) imports, and (c) exports (the abstract syntax for a module file is given in §4.2.3):

```
module XSLT {
   member xslt;
   import XMLParser;
   export xslt.Processor; }
```

To compile the source code for a module definition (in our case, the `xslt` Java package), the definitions of the types exported by the imported module definitions (in our case, the class `xml.parser.Parser`) need to be available; either their source code, or their bytecode. The latter can be obtained from the imported module definitions.

With the compilation of the member Java packages complete, we can put the resulting Java bytecode and the module file into a JAR-like file, i.e. a module definition (in our case, *XSLT*). This will most likely be achieved with a `jar`-like program, which might perform the compilation step automatically.[1]

Once we have obtained a module definition, we can install it into a Java Module System, most likely through an OSGi-like console.[2] If the files containing *XMLParser* and *XSLT* module definitions are named `xmlparser.md` and `xslt.md`, respectively, we would install and initialise them as follows (ignore '`bootstrap_r.`', for now):

```
jms> bootstrap_r.install(file:/programs/jms/xmlparser.md);
jms> bootstrap_r.install(file:/programs/jms/xslt.md);
jms> bootstrap_r.initialise(XSLT);
XMLParser initialised (mi: XMLParser-1).
XSLT initialised (mi: XSLT-2; linked with XMLParser-1).
```

[1] The draft documents do not define the exact packaging procedure.

[2] The administrator interaction with a JMS runtime has not been fully specified.

A module file can also specify the main class for a module definition, i.e. a class whose `main` method will get executed once the module definition has been initialised. In this thesis, we do not formalise the specification or the semantics of main classes.

The execution is guided by module instances, which are classloaders that either load a class from the corresponding module definition, or delegate loading to an import. The import declarations within the module files determine how the module instances are linked together. In the above example, two module instances are created, `XMLParser-1` and `XSLT-2`, one for each module definition, *XMLParser* and *XSLT*, respectively. The module instance `XSLT-2` is linked to `XMLParser-1` (not vice-versa), which implies that the code within `XSLT-2` can access its own types, and the exported public types of `XMLParser-1`. The exact semantics of type resolution in JMS is defined in §4.3.1.

The module system also defines repositories, which are runtime structures used for installing, finding, initialising, and un-installing module definitions. Every JMS runtime must contain the *bootstrap repository*, named `bootstrap_r`, which in turn contains the *core platform* module definition (core Java platform classes). In the above example, the two module definitions were installed and initialised within the bootstrap repository.

Each repository can have multiple child repositories. Since a module definition can import only module definitions within its own and ancestor repositories, we can control the dependency and isolation among module definitions through the use of multiple repositories. The exact semantics of how a particular module definition is located in the hierarchy of repositories is shown in §4.3.1, page 97.

Finally, if we install all module definitions for a particular application in a leaf child repository, we can stop and remove the application simply by removing its repository. In this thesis, we do not define the syntax or the semantics of repository creation and removal.

4.2 Syntax

The following subsections describe how we distinguish between LJAM compile-time and LJAM runtime code (§4.2.1), and show the updated definition of the context (§4.2.2), and LJAM's user (§4.2.3) and inner (§4.2.4) syntax.

4.2.1 Compile-time code vs. runtime code

In LJ, we did not distinguish between the abstract syntax of the compile-time code (pure abstract syntax) and the annotated runtime code (annotated abstract syntax). This is because the ctx annotation (guiding the runtime class resolution) of the object creation statement, '$var = \mathbf{new}_{ctx}\, cl();$', was always empty in LJ, so distinguishing between the two would be an overkill. We introduced only the annotated abstract syntax, because this way we were able to achieve a high level of definition reuse for LJAM and iJAM (Chapter 6).

In LJAM, however, we distinguish between the two versions of the syntax because (i) ctx is not empty (defined in §4.2.2), and plays an important role (shown later in §4.4), and (ii) both versions of the abstract syntax are used extensively in the LJAM's definition, so distinguishing between the two makes the definition more precise and more readable.

The pure abstract syntax (compile-time code) of an LJAM statement is:

s^{c}	::=	statement, compile-time code
	$\{\ \overline{s^{\mathsf{c}}_k}^{\,k}\ \}$	block
|	$var = x\,;$	variable assignment
|	$var = x\,.\,f\,;$	field read
|	$x\,.\,f = y\,;$	field write
|	$\mathbf{if}\ (x == y)\ s^{\mathsf{c}}_1\ \mathbf{else}\ s^{\mathsf{c}}_2$	conditional branch
|	$var = \mathbf{new}\ cl\ ();$	object construction
|	$var = x\,.\,meth\ (\overline{y})\,;$	method call

From here on, any non-terminal superscripted with c, e.g. s^{c}, denotes abstract syntax with no ctx annotations, i.e. compile-time code. The compile-time code is annotated just before the type-checking procedure — this is shown later in §4.3.3. Apart from the ctx annotations, the two versions of the syntax are identical.

Note that the annotation is only required for the 'object creation' statement, since only that statement contains a class identifier. At runtime, a classloader then uses such a context annotation to determine the type and the field names (corresponding to a class identifier), both of which are required to create a valid heap entry. Since the set of class definitions visible from any particular context can only grow (this holds for all our formalisations), we could have alternatively (and equivalently) annotated 'object creation' statements with types and field names (determined at link-time), instead.

4.2.2 LJAM's context (ctx)

Even though the draft documents do not describe any notion of an execution context, this concept is required in the formalisation, since it needs to be clear where the class resolution procedure starts searching, and where a class definition is found. For the module system to support component-level information hiding, module instances must create their own class namespaces. The fact that a module instance will create its own namespace is evident from the intent of implementing module instances with classloaders [51].[3]

Therefore, LJAM context, ctx, which was always empty for LJ, is defined as follows.

ctx	::=	context
|	$mi.pn$	def.

[3] As mentioned in §1.1.3, each classloader creates its own class namespace.

SRC	::=			source files ($\overline{cld^{\mathsf{c}}}$)
	|	$\overline{cld^{\mathsf{c}}}$	M	def.
cld^{c}	::=			class, compile-time code
	|	$pd\ am\ \mathbf{class}\ dcl\ \mathbf{extends}\ cl\ \{\ \overline{fd}\ \overline{meth\_def^{\mathsf{c}}}\ \}$		def.
pd	::=			package declaration (pn)
	|	$\mathbf{package}\ pn\ ;$	M	def.
am	::=			access modifier
	|			default
	|	$\mathbf{public}$		public

Figure 4.1: LJAM's changes to the class syntax

Meta-variables mi and pn are used for module instance identifiers and package names, respectively. Together, the pair uniquely identifies any LJAM's execution context.

4.2.3 User syntax

Developers can now write module files, mf, which have the following user syntax (the details of which are explained in the following paragraphs):

$$\mathbf{superpackage}\ mn\ \{\ \overline{\mathbf{member}pn;}\ \overline{\mathbf{import}m;}\ \overline{\mathbf{export}fqn;}\ \}$$

The definitions for module names, mn, and fully-qualified names, fqn, are:

mn	::=		module name
	|	$\mathbf{core\_m}$	core module
	|	m	standard module
fqn	::=		fully-qualified name
	|	$pn.dcl$	def.

The $\mathbf{core\_m}$ refers to the system's core module definition, which holds all the library classes, while m is a meta-variable that can refer to any other module definition; mn is used to refer to either.

A fully-qualified class name, fqn, can uniquely identify a class within a single module definition, but not within the whole system, since it is not prefixed with a module name. The ability to prefix type references with module names would require changing the underlying user syntax, which JMS cannot do due to backward compatibility constraints [51].

The class source files now include package declarations, pd, and access modifiers, am — see Fig. 4.1. Note that we define only the default (package) and the $\mathbf{public}$ access modifiers, because the draft JSRs change the semantics only for these two.

4.2.4 Inner syntax

The JSRs describe the runtime as a directed graph of repositories related with the parent-child relationship, where the root of the graph is the bootstrap repository, which contains the core Java libraries. Each of the non-bootstrap repositories can be used for installing, un-installing, and initialising module definitions. As mentioned earlier, the documents are not clear as to where and how the module instances are stored. From the available information, we created the following structures that can describe a runtime state.

A program, P, is now composed of a *repository context*, RC, and a *module hierarchy*[4], MH. The former holds information about all repositories and relationships among them, whereas the latter describes the connections among the existing module instances. The meta-variable r is used for repository names (different from $\mathbf{bootstrap\_r}$). Figure 4.2 shows the abstract syntax of these concepts — note that the productions of the repository context, RC, the repository cache, ϕ, and the module hierarchy, MH, are, in fact, meta-productions (§1.6.1).

The repository context, RC, is a partial map, which maps repository names, rn, to corresponding structures. Each repository structure, R, consists of installed module definitions, $\overline{md^{\mathsf{c}}}$, and its own cache, ϕ, where each module definition installed in the repository can be mapped to a module instance identifier of its module instance.

In a module hierarchy, MH, each module instance identifier, mi, is mapped to the corresponding module instance, md, and to a list of module instances identifiers, $\overline{mi}$, of imported module definitions.

Finally, we also have administrator actions, a. Once the module system is running, the administrator can, by inserting these actions into the system, install a compile-time version of a module definition, md^{c}, un-install a non-core module definition (named m), or initialise (and optionally execute) a non-core module definition (named m).

4.3 Operational semantics

The definition of a configuration, *config*, is the same as in LJ, i.e. a tuple of a program, P, a variable state, L, a heap, H, and statements left to execute, $\overline{s_l}^{\,l}$. It differs internally through the different definition of the program, P: in LJ, it is a list of class definitions, while in LJAM, it is a tuple of a repository context, RC, and a module hierarchy, MH.

The rules of the small-step operational semantics for LJAM's statements are syntactically identical to those of LJ (§3.3.3, page 66). LJAM also uses the same variable translation when flattening method calls. The statement semantics in LJAM still differs from that of LJ, though — this is due to a different class resolution, which we show in the following

[4] One should not draw analogies between module hierarchies and class hierarchies, since the former holds information about the runtime links, while the latter defines the inheritance relation.

P	$::=$			program
	$\mid$	$(RC,\, MH)$		def.
RC	$::=$			repository context ($rn \rightharpoonup R$)
	$\mid$	$[\,]$	M	empty repository context
	$\mid$	$RC\,[rn \mapsto R]$	M	rn maps to R in RC
rn	$::=$			repository name
	$\mid$	$\mathbf{bootstrap\_r}$		bootstrap
	$\mid$	r		standard
R	$::=$			repository
	$\mid$	$\mathbf{bootstrap\ repository}\,\{\,\overline{md^{\mathsf{c}}}\,;\,\phi\,\}$		bootstrap
	$\mid$	$\mathbf{repository}\ r\ \mathbf{child\ of}\ rn\,\{\,\overline{md^{\mathsf{c}}}\,;\,\phi\,\}$		standard
md^{c}	$::=$			module definition
	$\mid$	$\mathbf{module}\ mn\,\{\,\overline{cld^{\mathsf{c}}}\ \overline{m}\ \overline{fqn}\,\}$		def.
ϕ	$::=$			repository's cache ($md^{\mathsf{c}} \rightharpoonup mi$)
	$\mid$	$[\,]$	M	empty repository's cache
	$\mid$	$\phi\,[md^{\mathsf{c}} \mapsto mi]$	M	map md^{c} to mi in ϕ
	$\mid$	$\phi \setminus md^{\mathsf{c}}$	M	remove mapping for md^{c}
MH	$::=$			module hierarchy ($mi \rightharpoonup md \times \overline{mi}$)
	$\mid$	$[\,]$	M	empty module hierarchy
	$\mid$	$[mi \mapsto (md,\, \overline{mi})]$	M	maps mi to the given def. and imports
	$\mid$	$MH_1\,..\,MH_k$	M	composes many
a	$::=$			administrator action
	$\mid$	$rn\,.\,\mathbf{install}\,(md^{\mathsf{c}})\,;$		install
	$\mid$	$rn\,.\,\mathbf{uninstall}\,(m)\,;$		uninstall
	$\mid$	$rn\,.\,\mathbf{initialise}\,(m)\,;$		initialise

Figure 4.2: LJAM's inner syntax, and the syntax of its administrator actions

subsection. The same subsection also presents functions for finding module definitions within a repository hierarchy.

As shown in the previous section, LJAM also defines administrator actions. By writing these into an active module system, the administrator can install, un-install, and initialise module definitions, even while some underlying programs are executing. In §4.3.2, we describe the intended semantics, and show our formal rules that precisely define it.

4.3.1 Lookup functions

LJAM's structures for representing the program state are substantially different to those of LJ. However, LJ's function for finding class definitions ($\mathbf{find\_cld}$) already encapsulates these structures, while all the other lookup functions simply use this function. That is, only the $\mathbf{find\_cld}$ needs to be adapted; all the other functions, e.g. the method definition lookup function, $\mathbf{find\_meth\_def}$, remain as they are in LJ.

In this subsection, we first describe, in detail, how class resolution ($\mathbf{find\_cld}$) works in LJAM, and show an example. Then, we explain why inheritance path lookup function,

find_path, can remain unchanged, but its termination condition must be adapted. Finally, we present the module definition lookup function, **find_md**, which is used within administration actions (as shown in Fig. 4.5).

Here, we present these lookup functions according to the rules described in §1.6; examples, elaborate discussion, and suggestions for improvement are given in Chapter 5.

Finding a class definition

The statement semantics of LJ and LJAM differ *only* in their class resolution semantics, i.e. finding a class definition, cld, for a class name, cl. Instead of searching a sequence of class definitions, LJAM's class resolution now searches class definitions in module instances within the module hierarchy, MH. The high-level algorithm is to search (1) the core module, (2) the imported module instances (recursively), and, finally, (3) the module instance itself. The first appropriate class definition found is returned.

The following function implements the above algorithm. The found class definition is returned along with the context where it was located. If no appropriate definition is found, the function fails by returning None.

```
0  find_cld ((RC, MH), mi.pn, fqn)  :  (P × ctx × fqn) → (ctx × cld)_opt  =
1    match find_cld_in_core (P, fqn) with Some ctxcld → Some ctxcld | None →
2    match MH(mi) with None → None | Some (module mn { cld m fqn }, mi) →
3    match find_cld_in_imports (MH, mi, fqn) with
4      Some ctxcld → Some ctxcld | None →
5        match find_cld_in_self (cld, pn, fqn) with None → None | Some cld →
6          Some (mi.(package_name (cld)), cld)
```

In the first stage above (line 1), we call **find_cld_in_core**, the function shown below. This function locates the core library module within the bootstrap repository, and performs a simple search (**find_cld_in_module**, shown later) through its exported class definitions. The function **package_name** extracts the package name from a class definition.

```
find_cld_in_core ((RC, MH), fqn)  :  (P × fqn) → (ctx × cld)_opt  =
  match RC bootstrap_r with None → None | Some R →
  match R with
      repository r child of rn' { md^c ; φ } → None
    | bootstrap repository { md^c ; φ } →
        match find_md_in_mds (md^c, core_m) with None → None | Some md^c →
        match φ md^c with None → None | Some mi →
        match MH mi with None → None | Some (module mn { cld m fqn }, mi) →
        match find_cld_in_module (cld, fqn) with None → None | Some cld →
          Some (mi.(package_name (cld)), cld)
```

If nothing was found in the core library module, we recursively explore the imported modules (line 3, **find_cld**). Since the imports are searched before the importers, we are performing a *reverse* depth-first search, with imports on the same level searched in the order declared in the corresponding module file — see function below.

To respect selective exporting, a module instance (and its imports) is only searched if the name we are searching for, fqn, is within its exported class names, $\overline{fqn}$. The termination of the function is discussed later in this subsection.

```
0  find_cld_in_imports (MH, mi*, fqn)  :  (MH × mi* × fqn) → (ctx × cld)_opt  =
1    match mi* with [] → None | mi :: mi*' →
2    if ¬(acyclic_mh MH ∧ mi*' ⊆ dom (MH)) then None else
3    match MH(mi) with None → None | Some (module mn { cld* m fqn* }, mi*'') →
4    if fqn ∉ fqn* then find_cld_in_imports (MH, mi*', fqn) else
5    match find_cld_in_imports (MH, mi*'', fqn) with
6      Some ctxcld → Some ctxcld | None →
7        match find_cld_in_module (cld*, fqn) with
8          Some cld → Some (mi.(package_name (cld)), cld) | None →
9            find_cld_in_imports (MH, mi*', fqn)
```

When searching an *imported* module, i.e. *not* the module where the search started, the fully-qualified name, $pn.dcl$, must match, *and* the class must be declared **public**:

```
find_cld_in_module (cld*, pn.dcl)  :  (cld* × fqn) → cld_opt  =
  match cld* with [] → None
  | (package pn' ; am class dcl' extends cl { fd* meth_def* }) :: cld*' →
      if ¬distinct_fqns (cld*) then None else
      if am = public ∧ pn = pn' ∧ dcl = dcl' then Some cld else
        find_cld_in_module (cld*', fqn)
```

If the above function, **find_cld_in_module**, fails to find a result in all the imported modules searched by **find_cld_in_imports**, then the final stage of the class resolution begins — searching the module where the search started (line 5, **find_cld**). For this function, the fully-qualified name, $pn'.dcl$, must match *and* 'the class must be located in the initial context, pn, *or* it must be declared **public**.'

```
find_cld_in_self (cld*, pn, pn'.dcl)  :  (cld* × pn × fqn) → cld_opt  =
  match cld* with [] → None
  | (package pn'' ; am class dcl' extends cl { fd* meth_def* }) :: cld*' →
      if ¬distinct_fqns (cld*) then None else
      if pn' = pn'' ∧ dcl = dcl' ∧ (pn = pn' ∨ am = public) then Some cld else
        find_cld_in_self (cld*', pn, pn'.dcl)
```

For example, suppose we have module instances of module definitions *A*, *B*, *C*, *D*, and the core library module, *Core*, all of which are connected as shown in Fig. 4.3. If we started class resolution (**find_cld**) in *A*, the module instances would get searched as indicated by numbers in the brackets. Note that the repository boundaries are not important to class resolution, i.e. each module instance could be stored in a different repository; the figure only assumes that the *Core* module is stored within the bootstrap repository.[5]

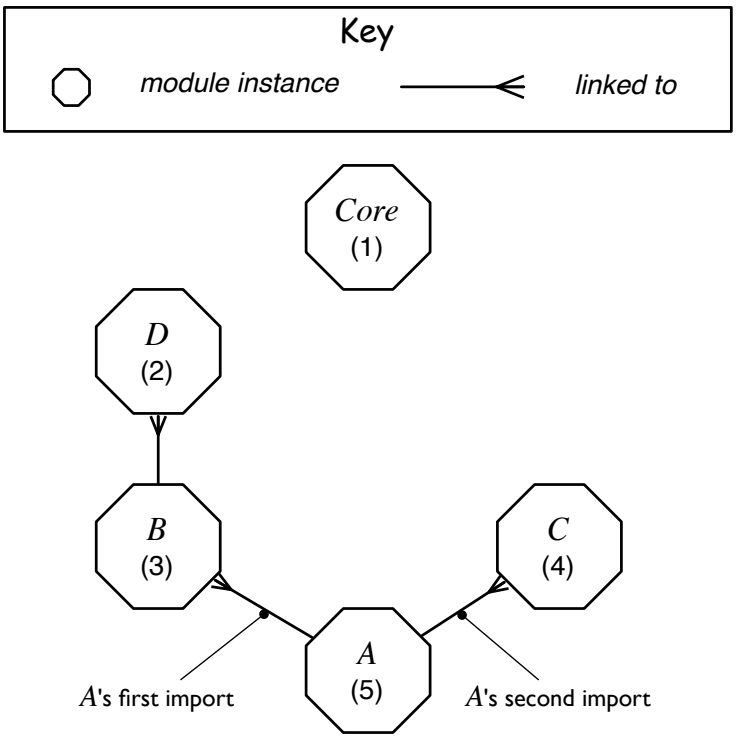


Figure 4.3: LJAM's class resolution order

The two functions above, **find_cld_in_module** and **find_cld_in_self**, both require that the class definitions within the module have distinct fully-qualified names. A simple reason for this is that we do not want the order of class definitions in a module to play a role in the semantics. A somewhat more elaborate and specific reason is the following: if we find a class in an imported module, we would like to find the same class if we started the search in that imported module — this is later formally expressed with Lemma 15 (page 102). As shown later in §4.5, fully-qualified class name distinctness within a module is already guaranteed by typechecking; however, since we do not have dependent types in our formalisation, we must re-state this constraint here.

[5]Type checking (§4.5) ensures that the bootstrap repository contains the core library module instance.

Proving termination of $\mathbf{find\_cld\_in\_imports}$

When defining Isabelle/HOL functions with non-primitive recursion, one also has to prove their termination. We have already seen an example of this when proving termination for LJ's $\mathbf{find\_path}$ (§3.3.2, page 63).

In LJAM, the recursive part of its class resolution, $\mathbf{find\_cld\_in\_imports}$, also contains general recursion. Our first solution to prove the termination of this function relied on the fact that there is only a finite number of module instances that can be explored, i.e. that some measurable (intermediate) quantity cannot be greater than some fixed quantity based on the size of the structure searched.

However, when proving type preservation, we have to prove that this function returns the same class definition even if an extra module instance has been added to MH (§4.6.2). This caused problems with our first solution, since we could no longer prove termination for a larger MH — the function failed in one case and not the other. Therefore, the termination should not be expressed in terms of the size of MH. We later found an appropriate termination condition: the value that decreases is 'the number of reachable module instances'. This property is expressed with the following relation.

REACHABLE_EMPTY

$$\overline{(MH,\ [],\ 0) \in \mathbf{reachable}}$$

REACHABLE_CONS

$$\frac{\begin{array}{l} MH\ (mi) = (md,\ \overline{mi}) \\ (MH,\ \overline{mi},\ nn') \in \mathbf{reachable} \\ (MH,\ mi_2\ ..\ mi_k,\ nn) \in \mathbf{reachable} \end{array}}{(MH,\ mi\ mi_2\ ..\ mi_k,\ nn' + nn + 1) \in \mathbf{reachable}}$$

As for LJ's $\mathbf{find\_path}$, we have to use a definite descriptor. The value that gets smaller with every recursive call is:

$$\iota\ nn.(MH,\ \overline{mi},\ nn) \in \mathbf{reachable}$$

To ensure that there exists a finite nn for each recursive call, we place an acyclicity constraint on the module hierarchy. This constraint is defined as follows.

AM_DEF

$$\frac{\begin{array}{l} 1.\ \mathbf{finite}\ (\mathbf{dom}\ (MH)) \\ 2.\ \forall \overline{mi}\ .\ \overline{mi} \subseteq \mathbf{dom}\ (MH) \longrightarrow \big(\exists nn\ .\ (MH,\ \overline{mi},\ nn) \in \mathbf{reachable}\big) \\ 3.\ \forall mi \in \mathbf{dom}\ (MH)\ .\ \exists md\ \overline{mi}\ .\ MH\ (mi) = (md,\ \overline{mi}) \ \wedge\ \overline{mi} \subseteq \mathbf{dom}\ (MH) \end{array}}{\mathbf{acyclic\_mh}\ MH}$$

Explanation. (1) The domain of MH must be finite; (2) for every subset of that domain, there exists a finite nn that satisfies the $\mathbf{reachable}$ relation; and (3) all module instances imported by any module instance in the domain of MH must also be in it.

Every module hierarchy satisfies the above property (§4.5, page 105); however, since Isabelle/HOL does not support dependent types, we have to check for this property at the beginning of $\mathbf{find\_cld\_in\_imports}$ (line 2 of the function, page 94).

Proving termination of find_path

The definition of the **find_path** function in LJAM is identical to that for LJ (§3.3.2, page 63). The relation that guarantees its termination, however, has a definition adapted to the new program definition (Fig. 4.2).

$$\frac{\forall pn\ fqn\ .\ \left(\begin{array}{c}\exists ctx'\ cld\ .\ \mathbf{find\_cld}\,(P,\ mi.pn,\ fqn) = (ctx',\ cld)\ \longrightarrow \\ \exists nn\ .\ (P,\ mi.pn,\ fqn,\ nn) \in \mathbf{path\_length}\end{array}\right)}{\mathbf{acyclic\_clds}_{mi}\,P}\ \text{ACM\_DEF}$$

$$\frac{\forall mi\ .\ \mathbf{acyclic\_clds}_{mi}\,P}{\mathbf{acyclic\_clds}\,P}\ \text{AC\_DEF}$$

The intuition does not change: (for every mi in program P) if we can find a class definition for some ctx and fqn, then the corresponding inheritance path has a *finite* length.

Finding a module definition

When a repository, R, is required to initialise a module definition named, mn, the corresponding module definition, md^{c}, is looked up in the following way: first (recursively) search within the parent repository of R, then perform a linear search (**find_md_in_mds**) within module definitions in R. The first matching module definition is returned.

The function below implements the above algorithm. The fourth parameter of the function, nn, stores the number of the repositories already explored, and serves to guarantee function's termination. If a named repository doesn't exist, or if none of the repositories in the chain of repositories contain an appropriate module definition, the function fails.

```
find_md_rec (RC, rn, mn, nn) : (RC × rn × mn × nn) → (rn × md^c)_opt =
  match RC rn with None → None | Some R →
  match R with
    bootstrap repository {md^c ; φ} →
      match find_md_in_mds (md^c, mn) with
        None → None | Some md^c → Some (rn, md^c)
  | repository r child of rn' {md^c ; φ} →
      if size (dom (RC)) ≤ nn then None else
      match find_md_rec (RC, rn', mn, nn + 1) of
        Some (rn'', md^c) → Some (rn'', md^c)
      | None → match find_md_in_mds (md^c, mn) of
          None → None | Some md^c → Some (rn, md^c)
```

The following function is a friendly interface to the above function.

$$\mathbf{find\_md}\,(RC, rn, mn) \;:\; (RC \times rn \times mn) \to (rn \times md^{\mathsf{c}})_{opt} \;=\; \mathbf{find\_md\_rec}\,(RC, rn, mn, 0)$$

The recursive part of the method definition lookup function, $\mathbf{find\_md\_rec}$, keeps track of the number of repositories already explored. If this number exceeds the number of repositories in the system, there must be a cycle, so the function terminates. Therefore, with each recursive call, we decrease 'the number of repositories we can still explore' — this measure makes the function well-founded. It would be better to prevent these cycles through the well-formedness rules.

4.3.2 Administrator actions

A system administrator can write administrator actions, a, into the system at any point during normal execution. If execution of an administrator action fails, any partial effects are reverted. These commands can install, un-install, or initialise a module definition.

The draft documents describe that the install and un-install actions only make a module definition visible and invisible to the initialisation procedure, respectively, i.e. they do not interfere with the normal execution of a program. The initialisation action, on the other hand, recursively initialises all the (recursively) imported module definitions. If any of the definitions was initialised before, their existing module instances are reused.

Since every module instance must bind against the module instances of its imports, the initialisation action, '$rn\,.\,\mathbf{initialise}\,(m)\,;$', must return a module instance identifier, mi, for each imported module instance. The module hierarchy, MH, then stores the binding. To store this intermediate result in a way that preserves the syntactic consistency for the reduction statements, we use internal variants of administration actions, ia. While the syntax of internal actions for installing and un-installing is the same, the syntax for initialisation is '$mi = rn\,.\,\mathbf{get\_instance}\,(m)$':

ia ::=		internal action
	$rn\,.\,\mathbf{install}\,(md^{\mathsf{c}})$	install
|	$rn\,.\,\mathbf{uninstall}\,(m)$	uninstall
|	$mi = rn\,.\,\mathbf{get\_instance}\,(m)$	initialise

The reduction of an administration action, '$config \xrightarrow{a} config'$', is then defined in terms of the reduction of internal administrator actions, '$config \xrightarrow{\overline{ia}} config'$' — see Fig. 4.4.

Using our program structures (§4.2.4) and the draft natural language description of the semantics of the administration actions, we wrote semantic rules (shown in Fig. 4.5) that precisely define the internal semantics of these actions. Each of the following paragraphs in this subsection describes one of the rules.

ADMIN_INSTALL

$$\frac{((RC,\ MH),\ L,\ H,\ \overline{s_l}^{\,l}) \xrightarrow{rn\,.\,\mathbf{install}\,(md^{\mathsf{c}})} ((RC',\ MH),\ L,\ H,\ \overline{s_l}^{\,l})}{((RC,\ MH),\ L,\ H,\ \overline{s_l}^{\,l}) \xrightarrow{rn\,.\,\mathbf{install}\,(md^{\mathsf{c}})\,;} ((RC',\ MH),\ L,\ H,\ \overline{s_l}^{\,l})}$$

ADMIN_UNINSTALL

$$\frac{((RC,\ MH),\ L,\ H,\ \overline{s_l}^{\,l}) \xrightarrow{rn\,.\,\mathbf{uninstall}\,(m)} ((RC',\ MH),\ L,\ H,\ \overline{s_l}^{\,l})}{((RC,\ MH),\ L,\ H,\ \overline{s_l}^{\,l}) \xrightarrow{rn\,.\,\mathbf{uninstall}\,(m)\,;} ((RC',\ MH),\ L,\ H,\ \overline{s_l}^{\,l})}$$

ADMIN_NEW_INSTANCE

$$\frac{((RC,\ MH),\ L,\ H,\ \overline{s_l}^{\,l}) \xrightarrow{mi=rn_1\,.\,\mathbf{get\_instance}\,(m)} ((RC',\ MH'),\ L,\ H,\ \overline{s_l}^{\,l})}{((RC,\ MH),\ L,\ H,\ \overline{s_l}^{\,l}) \xrightarrow{rn_1\,.\,\mathbf{initialise}\,(m)\,;} ((RC',\ MH'),\ L,\ H,\ \overline{s_l}^{\,l})}$$

Figure 4.4: LJAM's operational semantics for administrator actions

The installation action, '$rn\,.\,\mathbf{install}\,(md^{\mathsf{c}})$', (1) looks up the repository named rn, R, (2) inspects the body of R, (3-5) checks that the name (which cannot be $\mathbf{core\_m}$) of the given module definition, md^{c}, is distinct,[6] (6) adds md^{c} to R, creating R', and (7) re-maps rn to R'. The functions $\mathbf{R\_body}$ and $\mathbf{R\_update}$ extract and update the contents of a repository, respectively, while $\mathbf{md\_name}$ extracts the name of a given module definition.

The un-installation action, '$rn\,.\,\mathbf{uninstall}\,(m)$', (1) looks up the repository named rn, R, (2) inspects the body of R, (3-4) locates and removes the module definition named m (cannot be $\mathbf{core\_m}$), md^{c}, (5) removes any mapping for md^{c} in repository cache, ϕ, and updates R accordingly, creating R', and (6) re-maps rn to R'. We explain why this and other actions are safe to execute in §4.6. The function $\mathbf{mds\_rm}$ removes any occurrence of the given module definition from the given list.

The initialisation action, '$mi = rn_1\,.\,\mathbf{get\_instance}\,(m)$', is defined with two rules. The first describes what the action does if the appropriate module instance already exists. The action (1) starts the search in the repository named rn_1, finds a module definition named m, md^{c}, within repository named rn_2, (2-3) inspects the contents of this repository, and (4) finds that mi points to an existing instance of md^{c}, and returns it.

If an appropriate module instance does not exist yet, the system (1-3) takes a few steps as before, (4) finds that the repository cache has no mapping for md^{c}, (5) inspects the contents of md^{c}, (6) recursively creates module instances (pointed to by) $\overline{mi_k}^{\,k}$ of imported module definitions named $\overline{m_k}^{\,k}$, resulting in program state '$(RC',\ MH')$', (7) finds a fresh module instance identifier, mi, (8) creates md, a module instance of md^{c}, (9) maps mi to md and $\overline{mi_k}^{\,k}$ in MH', producing MH'', (10-11) finds repository named rn_2 and inspects its contents, (12) maps md^{c} to mi in the cache of R_2', producing R_2'', (13) re-maps rn_2 to R_2'', and (14) typechecks md in the final context.

[6] ϕ could be a map from module names, since (3-5) guarantee presence of modules with distinct names.

R_INSTALL

$$\frac{\begin{array}{l}1.\, RC\,(rn) = R \quad 2.\, \mathbf{R\_body}\,(R) = (\overline{md^{\mathsf{c}}_k}^{\,k},\, \phi) \\ 3.\, \mathbf{md\_name}\,(md^{\mathsf{c}}) = m \\ 4.\, \overline{\mathbf{md\_name}\,(md^{\mathsf{c}}_k) = mn_k}^{\,k} \quad 5.\, m \notin \overline{mn_k}^{\,k} \\ 6.\, \mathbf{R\_update}\,(R,\, md^{\mathsf{c}}\, \overline{md^{\mathsf{c}}_k}^{\,k},\, \phi) = R' \\ 7.\, RC' = RC\,[rn \mapsto R']\end{array}}{((RC,\, MH),\, L,\, H,\, \overline{s_l}^{\,l}) \xrightarrow{rn\,.\,\mathbf{install}\,(md^{\mathsf{c}})} ((RC',\, MH),\, L,\, H,\, \overline{s_l}^{\,l})}$$

R_UNINSTALL

$$\frac{\begin{array}{l}1.\, RC\,(rn) = R \quad 2.\, \mathbf{R\_body}\,(R) = (\overline{md^{\mathsf{c}}}_1,\, \phi) \\ 3.\, \mathbf{find\_md\_in\_mds}\,(\overline{md^{\mathsf{c}}}_1,\, m) = md^{\mathsf{c}} \\ 4.\, \mathbf{mds\_rm}\,(\overline{md^{\mathsf{c}}}_1,\, md^{\mathsf{c}}) = \overline{md^{\mathsf{c}}}_2 \\ 5.\, \mathbf{R\_update}\,(R,\, \overline{md^{\mathsf{c}}}_2,\, \phi \setminus md^{\mathsf{c}}) = R' \\ 6.\, RC' = RC\,[rn \mapsto R']\end{array}}{((RC,\, MH),\, L,\, H,\, \overline{s_l}^{\,l}) \xrightarrow{rn\,.\,\mathbf{uninstall}\,(m)} ((RC',\, MH),\, L,\, H,\, \overline{s_l}^{\,l})}$$

R_EXISTING_INSTANCE

$$\frac{\begin{array}{l}1.\, \mathbf{find\_md}\,(RC,\, rn_1,\, m) = (rn_2,\, md^{\mathsf{c}}) \quad 2.\, RC\,(rn_2) = R_2 \\ 3.\, \mathbf{R\_body}\,(R_2) = (\overline{md^{\mathsf{c}}}_2,\, \phi_2) \quad 4.\, \phi_2\,(md^{\mathsf{c}}) = mi\end{array}}{((RC,\, MH),\, L,\, H,\, \overline{s_l}^{\,l}) \xrightarrow{mi = rn_1\,.\,\mathbf{get\_instance}\,(m)} ((RC,\, MH),\, L,\, H,\, \overline{s_l}^{\,l})}$$

R_NEW_INSTANCE

$$\frac{\begin{array}{l}1.\, \mathbf{find\_md}\,(RC,\, rn_1,\, m) = (rn_2,\, md^{\mathsf{c}}) \quad 2.\, RC\,(rn_2) = R_2 \\ 3.\, \mathbf{R\_body}\,(R_2) = (\overline{md^{\mathsf{c}}}_2,\, \phi_2) \quad 4.\, \phi_2\,(md^{\mathsf{c}}) = \mathbf{null} \\ 5.\, md^{\mathsf{c}} = \mathbf{module}\, m\, \{\, \overline{cld^{\mathsf{c}}}\, \overline{m_k}^{\,k}\, \overline{fqn}\,\} \\ 6.\, ((RC,\, MH),\, L,\, H,\, \overline{s_l}^{\,l}) \xrightarrow{\overline{mi_k = rn_2\,.\,\mathbf{get\_instance}\,(m_k)}^{\,k}} ((RC',\, MH'),\, L,\, H,\, \overline{s_l}^{\,l}) \\ 7.\, mi \notin \mathbf{dom}\,(MH') \quad 8.\, \vdash_{mi} md^{\mathsf{c}} \rightsquigarrow md \\ 9.\, MH'' = MH'\,[mi \mapsto (md,\, \overline{mi_k}^{\,k})] \quad 10.\, RC'\,(rn_2) = R_2' \\ 11.\, \mathbf{R\_body}\,(R_2') = (\overline{md^{\mathsf{c}}}_3,\, \phi_3) \\ 12.\, \mathbf{R\_update}\,(R_2',\, \overline{md^{\mathsf{c}}}_3,\, \phi_3\,[md^{\mathsf{c}} \mapsto mi]) = R_2'' \\ 13.\, RC'' = RC'\,[rn_2 \mapsto R_2''] \quad 14.\, (RC'',\, MH'') \vdash_{mi} md\end{array}}{((RC,\, MH),\, L,\, H,\, \overline{s_l}^{\,l}) \xrightarrow{mi = rn_1\,.\,\mathbf{get\_instance}\,(m)} ((RC'',\, MH''),\, L,\, H,\, \overline{s_l}^{\,l})}$$

Figure 4.5: LJAM's operational semantics for internal administrator actions

4.3.3 Context insertion

Step (8) of the initialisation action (R_NEW_INSTANCE, Fig. 4.5) creates a module instance, md, of module definition md^{c} by creating a copy of md^{c}, and recursively inserting (together with an appropriate package name) the module instance identifier of md, i.e. mi, into every object creation statement. In particular, a statement '$var = \mathbf{new}\ cl\,();$' within a package, pn, becomes '$var = \mathbf{new}_{mi.pn}\, cl();$'. This allows type-checking and statement

reduction to start searching for class definitions within appropriate contexts. Due to the simplicity of the context insertion rules, we put them in the appendix (§C.3, page 175).

4.4 Type system

The type system changes from LJ to LJAM due to a different definition for a context, ctx. Most of this change is hidden within the new definition of a type, τ; however, the subtyping relation and the lemmas based on it change a little, too.

4.4.1 Type (τ)

The definition of a type is syntactically identical to that of LJ's type (page 68): `Object`, or $ctx.dcl$. The definition of the context, however, has changed from empty in LJ (page 60) to a tuple of module instance identifier and package name in LJAM (page 89), e.g. $mi.pn$.

When looking up a class definition for a particular type, LJ's lookup functions (§3.3.2) do not change the context or the class name they are searching for. This means that a class definition in LJ has only one way of referring to it. For example, a class named dcl in a context ctx has a single valid class reference (type): $ctx.dcl$.

LJAM's lookup functions (§4.3.1), however, search different packages and module instances, and so *do* change the context in which they search. Because of this, a class definition in LJAM can have more than a single way of referring to it. For example, if a module instance named mi imports another module instance named mi', and (only) mi' contains a class definition with a fully-qualified name $pn.dcl$, then both $mi.pn.dcl$ and $mi'.pn.dcl$ are valid class identifiers (types) for the class definition.

We refer to the type that directly specifies the context and the name of the referred class as the *primary type*. Each class definition in the system has exactly one primary type. Note that in LJ every valid type is also a primary type. To obtain a primary type from a valid type in LJAM, we have to first look up the class definition for that valid type, then extract the primary type from it.

4.4.2 Subtyping ($P \vdash \tau \prec \tau'$)

The subtyping relation is similar to that of LJ (§3.4.3, page 69). The rule STY_OBJ remains unchanged, while STY_DCL contains a subtle change.

$$\frac{1.\ \mathbf{find\_path}\,(P,\ \tau) = \overline{ctxcld}}{P \vdash \tau \prec \texttt{Object}}\ \text{STY\_OBJ}$$

$$\frac{\begin{array}{l}1.\ \mathbf{find\_path}\,(P,\ \tau) = \overline{ctxcld} \\ 2.\ \mathbf{find\_cld}\,(P,\ mi'.pn',\ pn'.dcl') = ctxcld \\ 3.\ ctxcld \in \overline{ctxcld}\end{array}}{P \vdash \tau \prec mi'.pn'.dcl'}\ \text{STY\_DCL}$$

The **find_path** function (page 63) returns a list of tuples of contexts and class definitions, which is (for purposes of typing) equivalent to a list of primary types. LJ's STY_DCL then simply checks that the given supertype is within those primary types — this is fine, because in LJ every valid type is a primary type. Since this is not the case in LJAM, LJ's subtyping relation would give us only a small subset of all the required subtype pairs.

Using the **find_cld** function (§4.3.1, page 93) on the given supertype returns the corresponding class definition and its context, $ctxcld$, which are (for purposes of typing) its primary type. The third line then simply checks that this primary type is within the primary types of the inheritance path of the subtype.

Lemma 1 (Subtype is a valid type), Lemma 2 (Inheritance sub-path), and Lemma 3 (Supertype is a valid type) remain unchanged, except for **class_name** being replaced with **full_name** to account for the package name. Their proofs required only minor updates.

4.4.3 Type reflexivity

To prove type reflexivity, we need the following lemma.

Lemma 13 (Subtype first)**.**

$$\begin{aligned}&\mathbf{find\_path}\,(P, ctx, fqn) \;=\; \texttt{Some}\;\; \overline{ctxcld} \\ &\quad\Longrightarrow \exists ctxcld.\;\; \mathbf{find\_cld}\,(P, ctx, fqn) \;=\; \texttt{Some}\;\; ctxcld \quad\wedge \\ &\qquad\qquad (\exists \overline{ctxcld}'.\;\; \overline{ctxcld} = ctxcld :: \overline{ctxcld}')\end{aligned}$$

Explanation. If we look up an inheritance path for a class, then the head of that path is the class of which inheritance path we searched for.

Proof. By well-founded induction over **find_path_rec** (§3.3.2, page 63). □

Lemma 14 (Type reflexivity)**.** $P \vdash \tau \implies P \vdash \tau \prec \tau$

Proof. LJ's proof (of Lemma 4, page 71) is adapted to use the above lemma, which accounts for the changes to the subtyping relation. □

4.4.4 Type transitivity

The proof of LJAM's type transitivity is substantially more complex than that of LJ (page 71). This is mainly because Lemma 2 (page 70) used in LJ's lemma deals only with primary types. We need a few more lemmas, three of which we present here.

Lemma 15 (Target context equivalence for class lookups)**.**

$$\begin{aligned}&\mathbf{find\_cld}\,(P, ctx, fqn) \;=\; \texttt{Some}\;\; (ctx', cld') \quad\Longrightarrow \\ &\mathbf{find\_cld}\,(P, ctx', fqn) \;=\; \texttt{Some}\;\; (ctx', cld')\end{aligned}$$

Explanation. If we look for a class named fqn in context ctx, and we find a class definition cld' in context ctx', then we will get the same result if we start the search in ctx' instead.

Proof. By comparing symbolic executions of the two lookups down to every detail of all the dependent lookup functions (definitions of these are shown in §4.3.1). The proof includes a well-founded induction on **find_cld_in_imports** (§4.3.1, page 94). □

Lemma 16 (Inheritance path consistency)**.**

$$\begin{aligned}&\mathbf{find\_path}\,(P, ctx, cl) \;=\; \texttt{Some}\;\; \overline{ctxcld} \quad\wedge\quad (ctx', cld') \in \overline{ctxcld}\\ &\quad\Longrightarrow \mathbf{find\_cld}\,(P, ctx', (\mathbf{full\_name}\,(cld'))) \;=\; \texttt{Some}\;\; (ctx', cld')\end{aligned}$$

Explanation. Taking any class definition (and its context) from an inheritance path, and looking up a class definition with the corresponding primary type, gives the original class definition (and context).

Proof. By well-founded induction over **find_path_rec** (§3.3.2, page 63), using also lemmas about context equivalence such as Lemma 15. The proof has the same structure as the one for Lemma 2 (page 70). □

Lemma 17 (Target context equivalence for paths lookups)**.**

$$\begin{aligned}&\mathbf{find\_path}\,(P, ctx, cl) \;=\; \texttt{Some}\;\; ((ctx', cld') :: \overline{ctxcld}) \quad\Longrightarrow\\ &\mathbf{find\_path}\,(P, ctx', (\mathbf{full\_name}\,(cld'))) \;=\; \texttt{Some}\;\; ((ctx', cld') :: \overline{ctxcld})\end{aligned}$$

Explanation. Looking up the inheritance path with a primary type corresponding to the head of an existing inheritance path will give the same result.

Proof. With a well-founded induction over **find_path_rec**, and Lemma 16. □

Lemma 18 (Type transitivity)**.** $P \vdash \tau \prec \tau' \;\wedge\; P \vdash \tau' \prec \tau'' \;\Longrightarrow\; P \vdash \tau \prec \tau''$

Proof. We know that the valid (but not necessarily primary) types τ' and τ'' have corresponding class definitions within inheritance paths of τ and τ', respectively. Using the definition of the subtype relation (§4.4.2, page 101) and a single application of Lemma 2 (page 70), we are left with proving that the inheritance paths of τ' and its primary type, τ'_p, are the same. With Lemma 13, we split the inheritance path into its head and tail. With Lemma 16, we show that we find the same class definition for both τ' and τ'_p. Finally, Lemma 17 allows us to combine the inheritance path lookup for τ', and the class definition lookup for τ'_p, to get the same inheritance path for τ'_p. □

4.5 Type checking

Much of LJ's type checking is reused in LJAM. Specifically, the well-formedness of statements, variable state, and heap is identical. The main changes are due to the different definition of the program structure.

Program ($\vdash\ P$)

A program, P, is well-formed *iff* its repository context, RC, and module hierarchy, MH, are well-formed, and there are no cycles in the class inheritance graph (§4.3.1, page 97).

$$\frac{1.\ MH \vdash RC \quad 2.\ RC \vdash MH \quad 3.\ \mathbf{acyclic\_clds}\,(RC,\ MH)}{\vdash\ (RC,\ MH)}\ \text{WF\_P}$$

Repository context ($MH\ \vdash\ RC$)

A repository hierarchy, RC, is well-formed *iff* all repositories in RC are properly mapped and well-formed, and RC includes a bootstrap repository. The function $\mathbf{R\_name}$ extracts the name of a given repository.

$$\frac{1.\ \forall rn \in \mathbf{dom}\,(RC)\,.\,\exists R\,.\,\left(\begin{array}{l} RC\,(rn) = R\ \wedge\ \mathbf{R\_name}\,(R) = rn\ \wedge \\ (RC,\ MH)\ \vdash\ R \end{array}\right) \quad 2.\ \mathbf{bootstrap\_r} \in \mathbf{dom}\,(RC)}{MH\ \vdash\ RC}\ \text{WF\_RC}$$

Repository ($P\ \vdash\ R$)

A repository is well-formed when its repository cache, ϕ, is well-formed. A bootstrap repository must also contain a core module definition, whereas a normal repository must specify a valid parent repository.

$$\frac{1.\ \mathbf{find\_md\_in\_mds}\,(\overline{md^c},\ \mathbf{core\_m}) = md^c \quad 2.\ MH\ \vdash\ \phi}{(RC,\ MH)\ \vdash\ \mathbf{bootstrap\ repository}\,\{\,\overline{md^c}\,;\,\phi\,\}}\ \text{WF\_BOOTSTRAP\_REP}$$

$$\frac{1.\ r\ \neq\ rn \qquad 2.\ rn \in \mathbf{dom}\,(RC) \quad 3.\ MH\ \vdash\ \phi}{(RC,\ MH)\ \vdash\ \mathbf{repository}\ r\ \mathbf{child\ of}\ rn\,\{\,\overline{md^c}\,;\,\phi\,\}}\ \text{WF\_NORMAL\_REP}$$

Repository cache ($MH\ \vdash\ \phi$)

A repository cache, ϕ, is well-formed *iff* all of its module instances identifiers are within the program's module hierarchy, MH. Since there are strict requirements on the domain of MH (below), the following rule is not as weak as it might appear.

$$\frac{1.\ \mathbf{ran}\,(\phi)\ \subseteq\ \mathbf{dom}\,(MH)}{MH\ \vdash\ \phi}\ \text{WF\_RMIS}$$

Module hierarchy ($RC \vdash MH$)

A module hierarchy, MH, is well-formed *iff* its import graph is acyclic (§4.3.1, page 96), and all module instance identifiers are mapped to well-formed module instances.

WF_MH

$$\frac{1.\ \mathbf{acyclic\_mh}\ MH \quad 2.\ \forall mi \in \mathbf{dom}\,(MH)\,.\,\exists md\,\overline{mi}\,.\,MH\,(mi) = (md,\,\overline{mi})\ \wedge\ (RC,\,MH) \vdash_{mi} md}{RC \vdash MH}$$

Module instance ($P \vdash_{mi} md$)

The module instance, md, is an instance of some module definition, md^{c}, which has been annotated with its own module instance identifier, mi. After context insertion (§4.3.3), the instance is inserted into the module hierarchy, and linked to other instances. At this point, the lookup functions can be used, both for type checking and for execution of md.

Therefore, a module instance, md (named mi), is well-formed *iff* its class definitions are well-formed, their names are distinct, and their inheritance paths are finite. Note that class definitions are well-formed with respect to mi, i.e. mi is the context in which *all* type references within the class definitions are resolved. All classes within md must also have finite inheritance paths to guarantee termination of the lookup functions.

$$\frac{1.\ \overline{\mathbf{full\_name}\,(cld_k) = fqn_k}^{\,k} \quad 2.\ \mathbf{distinct}\,(\overline{fqn_k}^{\,k}) \quad 3.\ \overline{P \vdash_{mi} cld_k}^{\,k} \quad 4.\ \mathbf{acyclic\_clds}_{mi} P}{P \vdash_{mi} \mathbf{module}\ mn\ \{\ \overline{cld_k}^{\,k}\ \overline{m}\ \overline{fqn}\ \}}\ \text{WF\_MODULE}$$

Please note that, due to Ott naming rules (§1.6.1, page 38), '$\overline{fqn_k}^{\,k}$' and '$\overline{fqn}$' refer to distinct lists of fully-qualified names.

Class ($P \vdash_{mi} cld$)

This rule simply refers to LJ's rule for common class well-formedness (page 74) — the rule for class definition well-formedness, and all the rules used within, are identical to those in LJ. The semantics of these rules changes only due to modified lookup functions.

$$\frac{1.\ P \vdash_{mi.pn} (dcl, cl, \overline{fd}, \overline{meth\_def})}{P \vdash_{mi} \mathbf{package}\ pn\ ;\ am\ \mathbf{class}\ dcl\ \mathbf{extends}\ cl\ \{\ \overline{fd}\ \overline{meth\_def}\ \}}\ \text{WF\_CLASS}$$

4.6 Proof of type soundness

LJAM's configuration has the same well-formedness conditions as that of LJ (§ 3.6.1, page 77), i.e. that its program, P, variable state, L, heap, H, and statements to be executed, $\overline{s_k}^k$, must all be well-formed. We again prove type soundness by proving progress and type preservation. Like all other in this document, the following proofs are the natural language versions of our Isabelle/HOL proofs.

4.6.1 Progress

Since administrator actions can be performed at any point in the execution, and since all partial effects of a failed action are reverted (§4.3.2), reduction of administrator actions does not appear in the progress theorem. For this reason, the progress property is identical to the one for LJ.

Theorem 19 (Progress)**.**

$$\Gamma \vdash (P, L, H, \overline{s}) \wedge \overline{s} \neq [] \implies \exists \mathit{config}.\ (P, L, H, \overline{s}) \longrightarrow \mathit{config}$$

Proof. The proof of progress for LJ (§3.6.3, page 79) directly depends on the statement reduction, the type system, some well-formedness rules (statement, variable state, and heap), some lookup functions, $\mathbf{find\_type}$ and $\mathbf{find\_meth\_def}$, and some helper lemmas. These helper lemmas were adapted according to the small changes in the type system (§4.4), while all the other (direct) dependencies have identical definitions in LJ and LJAM.

Both of the mentioned lookup functions indirectly depend on the function defining the class resolution, $\mathbf{find\_cld}$, which has a substantially different definition in LJ and LJAM. However, the progress proof is, in fact, independent from the implementation of $\mathbf{find\_cld}$. We are required to know only that the function returns a valid result in certain cases, all of which are guaranteed by the well-formedness rules, i.e. if a the program typechecks, these function calls must have succeeded, and will do so at runtime, too.

Due to a more complex program structure, LJAM has a different definition for the context, ctx. This definition is used within the subtyping relation, and $\mathbf{find\_cld}$. The adapted lemmas encapsulate concrete uses of ctx within the subtyping relation; while, as mentioned above, LJ's progress proof is invariant of the implementation of $\mathbf{find\_cld}$.

Because of all of the above reasons, LJAM's progress proof is identical to the one for LJ, modulo small updates to the proofs of a few helper lemmas (as outlined in the previous sections). Since our Isabelle/HOL scripts respect the above-mentioned abstraction boundaries, we also achieve a high reuse of the Isabelle/HOL proof scripts (§6.6). □

4.6.2 Type preservation

The proof of LJAM's type preservation is more complex than that of LJ, since it must also deal with the complexities of LJAM's class resolution, and ensure the well-formedness preservation of the administrator action reductions.

Theorem 20 (Type preservation)**.**

$$\begin{array}{l} \Gamma \vdash \mathit{config} \ \wedge \ (\mathit{config} \longrightarrow \mathit{config}' \ \vee \ \mathit{config} \xrightarrow{a} \mathit{config}') \\ \quad \Longrightarrow \ \exists \Gamma'.\ \Gamma \subseteq_m \Gamma' \ \wedge \ \Gamma' \vdash \mathit{config}' \end{array}$$

Explanation. If *config* is a well-formed configuration in a type environment, Γ, and *config* reduces in one step to *config*$'$ through either statement reduction, $\longrightarrow$, or administrator action reduction, $\xrightarrow{a}$, *config*$'$ is well-formed in some greater ($\subseteq_m$) type environment, Γ'.

Proof. First, we consider the case where a statement reduction, $\longrightarrow$, has occurred. As for the progress proof, this part can be identical to the one for LJ (modulo small updates to the proofs of a few helper lemmas), since most dependencies have the same definitions.

The administrator action reduction, $\xrightarrow{a}$, case is more complicated. Intuitively, the proof goes as follows. The installation and the un-installation of a module definition (R_INSTALL and R_UNINSTALL, Fig. 4.5, page 100) are safe, since they only modify (in a well-formed way) a repository in the repository context, RC, which cannot affect currently executing code: the statement semantics only uses the module hierarchy part of the program, except when locating the core library module, which the administration actions, by definition, cannot modify. The initialisation action (R_NEW_INSTANCE), recursively creates instances of module definitions, which are all typechecked before made available to the statement execution semantics; the action also updates (in a well-formed way) the repository caches.

We start by defining a *well-formed program change*. The judgement '$(P,\ mi,\ P') \in \mathbf{wf\_P\_change}$', defined in Fig. 4.6,[7] states that an administrator action has successfully executed, changing the program state from P to P'. The mi is an identifier of a module instance, which might exist in P' due to the reduction of the administrator action associated with this program change, but is otherwise free in P. This relation closely corresponds to the semantics of administrator actions (Fig. 4.5, page 100), but abstracts away from details unrelated to program well-formedness. Since it applies to *any* administration action, the relation allows easy reuse of helper lemmas that do not depend on a specific action.

For each of the LJAM's lookup functions we then prove that searching within P' (starting in some ctx) gives the same result as searching within P (starting in the same ctx) *if* ctx is not mi. We also prove that if searching within P (starting in some ctx) gives a

[7] The relation was originally written within Isabelle/HOL. Later, we re-wrote it Ott, added it to LJAM's definition, and so also obtained a version that has its LaTeX consistent with the other LJAM rules.

WRC_INSTALL

$$\frac{1.\ \vdash (RC,\ MH)\ \ 2.\ mi \notin \mathbf{dom}\,(MH)\quad 3.\ RC\,(rn) = R\quad 4.\ \mathbf{R\_body}\,(R) = (\overline{md^{\mathsf{c}}},\ \phi)\quad 5.\ \mathbf{md\_name}\,(md^{\mathsf{c}}) = m\quad 6.\ \mathbf{R\_update}\,(R,\ md^{\mathsf{c}}\,\#\,\overline{md^{\mathsf{c}}},\ \phi) = R'}{((RC,\ MH),\ mi,\ (RC\,[rn\ \mapsto\ R'],\ MH)) \in \mathbf{wf\_P\_change}}$$

WRC_UNINSTALL

$$\frac{1.\ \vdash (RC,\ MH)\ \ 2.\ mi \notin \mathbf{dom}\,(MH)\quad 3.\ RC\,(rn) = R\quad 4.\ \mathbf{R\_body}\,(R) = (\overline{md^{\mathsf{c}}}_1,\ \phi)\quad 5.\ \mathbf{find\_md\_in\_mds}\,(\overline{md^{\mathsf{c}}}_1,\ m) = md^{\mathsf{c}}\quad 6.\ \mathbf{mds\_rm}\,(\overline{md^{\mathsf{c}}}_1,\ md^{\mathsf{c}}) = \overline{md^{\mathsf{c}}}_2\quad 7.\ \mathbf{R\_update}\,(R,\ \overline{md^{\mathsf{c}}}_2,\ \phi \setminus md^{\mathsf{c}}) = R'}{((RC,\ MH),\ mi,\ (RC\,[rn\ \mapsto\ R'],\ MH)) \in \mathbf{wf\_P\_change}}$$

WRC_NEW_INSTANCE

$$\frac{\begin{array}{l}1.\ \vdash (RC,\ MH)\qquad 2.\ mi \notin \mathbf{dom}\,(MH)\\ 3.\ RC\,(rn) = R\\ 4.\ \mathbf{R\_body}\,(R) = (\overline{md^{\mathsf{c}}},\ \phi)\\ 5.\ \overline{mi}\ \subseteq\ \mathbf{dom}\,(MH)\\ 6.\ \mathbf{md\_name}\,(md^{\mathsf{c}}) = m\\ 7.\ \mathbf{R\_update}\,(R,\ \overline{md^{\mathsf{c}}},\ \phi\,[md^{\mathsf{c}}\ \mapsto\ mi]) = R'\\ 8.\ RC' = RC\,[rn\ \mapsto\ R']\\ 9.\ MH' = MH\,[mi\ \mapsto\ (md,\ \overline{mi})]\\ 10.\ (RC',\ MH') \vdash_{mi} md\end{array}}{((RC,\ MH),\ mi,\ (RC',\ MH')) \in \mathbf{wf\_P\_change}}$$

Figure 4.6: LJAM's well-formed program change

result (not None), then searching within P' (starting in the same ctx) gives the same result, regardless of ctx. Both steps are non-trivial, especially for functions that use general recursion, e.g. **find_path_rec** (page 62).

Using the above result, we then prove that all well-formedness relations are preserved through a well-formed program change. For example, we prove that if a statement s is well-formed in P, it is also well-formed in P', where '$(P, mi, P') \in \mathbf{wf\_P\_change}$'.

Finally, we induct on the definition of the reduction for administrator actions. We show for each action that it satisfies the well-formed program change, which allows us to use all the above-mentioned lemmas. For R_NEW_INSTANCE (page 100), the well-formedness of the final state relies on the presence of the imported module instances, $\overline{mi_k}^{\,k}$, within an intermediate module hierarchy, MH' — therefore, the induction hypothesis must include a statement saying "if a module initialisation action, '$mi = rn\,.\,\mathbf{get\_instance}\,(m)$', executes successfully, mi is bound in the resulting module hierarchy." With this, we can then show that each administrator action preserves configuration well-formedness. □

4.7 Recent changes to the Java Module System

At the time of this writing, the Java Module System was still in development, and the draft documents describing its design were being continually updated. Our formalisation is based on a snapshot [51, 52] in this process. This section tries to describe the changes made to the draft design after this snapshot, which are inconsistent with our formalisation, or our description of the documents themselves:

- JSR-294 has merged into JSR-277;
- keyword **superpackage** has been replaced with **module**;
- a source-level entity (class, interface, etc.) annotated with the keyword **public** is now implicitly exported by any module it is a member of;
- a new entity-accessibility keyword, **module**, has been introduced, which takes on the previous meaning of **public**, i.e. visible to all other members in the module;
- accessibility constraints for module members are no longer expressed within a module file, i.e. the list of exports has become purely implicit.

It is not clear how forward bytecode-compatibility is preserved through the introduction of the new keyword. Our guess is that, with respect to the Java's functions checking for accessibility, the keyword is bit-wise compatible with **public**. We do know, however, that the keyword is context sensitive, i.e. one can still declare a variable named '`module`.'

We are not aware of any other incompatible changes. All of the above change only the surface of the module system, which means that our formalisation is still an accurate model for the system's semantics.

4.8 Conclusion

We now have a rigorous definition of a core of the Java Module System with theoretical confidence in its type soundness, which tells us that the system "will not go wrong" according to the semantics; however, it does *not* tell us that its semantics is what we intend it to be, or that it is useful.

In Chapter 7, we write a proof-of-concept implementation that closely corresponds to the semantics specified in LJAM. The examples shown in the draft documents run in our implementation as intended. This gives us some confidence that our formalisation does, in fact, reflect the draft documents.

With its formalisation in hand, we are able to precisely analyse and discuss possible deficiencies of the module system. The next chapter reveals two key weaknesses — we hope that both of these will be addressed appropriately before the final release of JMS.

5

Problems with the Java Module System

In this chapter, we identify and analyse Java Module System's two key problems: unintuitive and insufficiently expressive class resolution, and inflexible module instantiation. For each, we show what they are, the reasons for corresponding design choices, their implications in practice, and how they can be fixed.

5.1 Class resolution

The first key problem with the module system concerns its definition of class resolution. The procedure (recursively) searches the imported module definitions (following the order specified in the module file) before searching the client module. This is done in order to (1) prevent anyone from overriding the core library classes, since the *core module* (Java's core library classes) is logically the root of the import graph, and therefore looked up first, (2) to minimise `ClassCastException`s, and (3) to promote the sharing of static data and types. The latter two are achieved by preferring a class that is more accessible to the whole system; therefore, it is more likely that identical class references in different contexts refer to the same class definition.

A `ClassCastException` is thrown when an object is cast to an inappropriate type. For example, an object of type `T` is sent as an `Object` from module instance *A* to module instance *B*, where it is cast back to "`T`", but "`T`" in the new context, *B*, resolves to a different class definition; therefore, an exception is thrown. If the resolution algorithm tends

to resolve to definitions towards the root of the module import graph, then *A*'s "`T`" and *B*'s "`T`" likely point to the same `T`, avoiding `ClassCastException`s when exchanging objects of type `T` — in §5.3, we further discuss the prevention of such exceptions.

The above-described name resolution order does, however, have two deficiencies. First, the name resolution is highly unintuitive in practice (§5.1.1). And second, it is impossible to access all visible types (§5.1.2).

5.1.1 Unintuitive class resolution

Problem Suppose, for example, that the developers of a module, *XMLParser*, release an update, which makes some new functionality available through a new (and exported) class, `ParserX`. Since the new *XMLParser* module is compatible with the old one, the developers only modify its micro version number, which implies that the modules that previously imported the old version will now likely import the new version, automatically (due to commonly-used flexible version constraints). However, if an importing module, e.g. *XSLT*, already contained a class named "`ParserX`", then any reference to a class `ParserX` in *XSLT* will now incorrectly resolve to `ParserX` in *XMLParser*. The resolution problem is shown in Fig. 5.1.

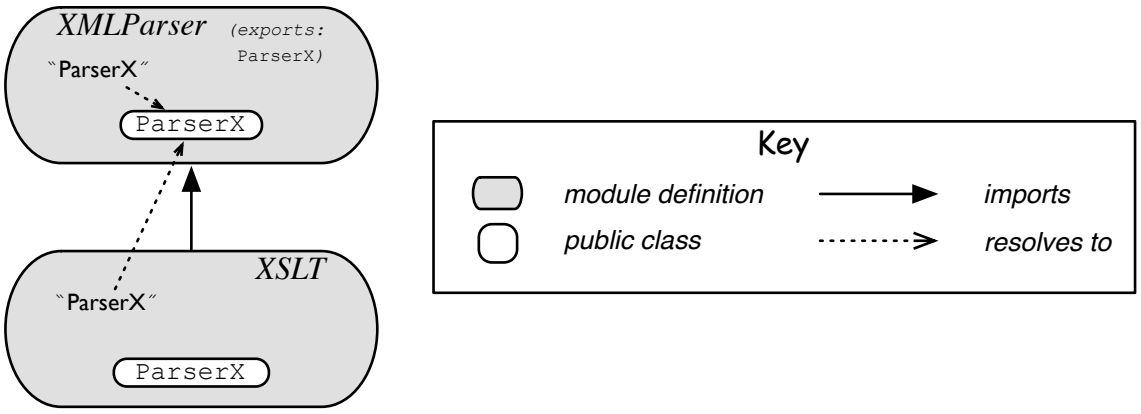


Figure 5.1: LJAM's unintuitive and inexpressive class resolution

One might argue that due to Java's class naming conventions, i.e. always prefixing its package name with a reversed domain name of the company that wrote the class, the above scenario is unlikely. However, such a scenario can easily appear if the two module definitions are written within the same company, and the developers of the module definitions genuinely want to use different versions of `ParserX`, respectively. Furthermore, the reason for these naming conventions is Java's lack of namespace control.

Furthermore, due to the JMS's oversimplified relation between modules, the importing module has no control over which exported classes of the imported modules are visible. This ultimately means that a class within *XSLT* has *no* way of referring to *XSLT*'s

`ParserX` in the example shown in Fig. 5.1 without module-wide renaming.

Any change to the underlying language is highly undesirable due to various compatibility issues. Because of this, we cannot introduce proper namespaces to the source language, which would allow class references independent of the class resolution semantics, e.g. `XMLParser::ParserX`.

Solution We adapt the class resolution algorithm to search the core library module, then the module itself, and finally the imported modules (recursively) — the resolution in the above example according to the new algorithm is shown in Fig. 5.2.

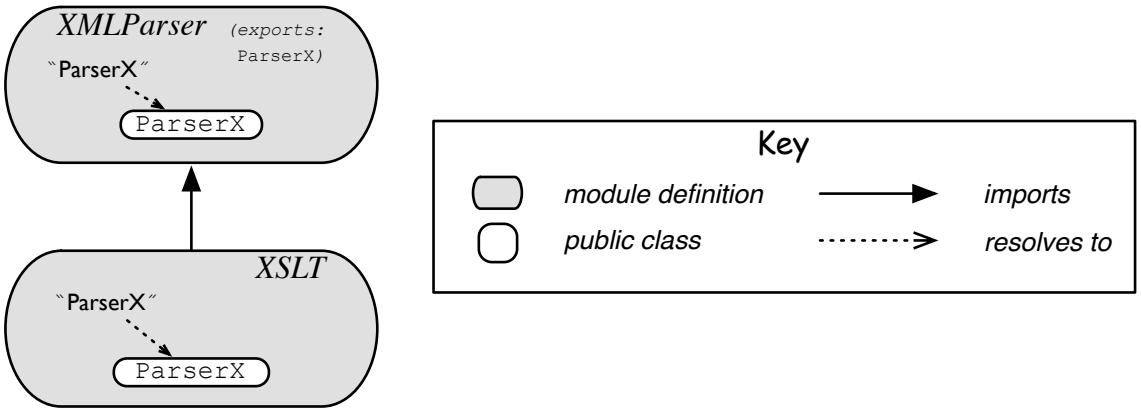


Figure 5.2: A more intuitive, but still inexpressive, class resolution

If a new version of an imported module, e.g. *XMLParser*, now exports an extra name, e.g. `ParserX`, which is already bound in the client module, no names will get re-bound. That is, the semantics is now robust against incremental interface evolution of the imported modules. On the other hand, if a name is bound locally when it should be bound in an imported module, the fix is module-local — by removing/renaming the local definition.

Note that the new algorithm still protects the core library (by searching it first). The down side is that more scenarios result in `ClassCastException`s being thrown, since resolution no longer prefers definitions towards the root of the module import graph; however, we can avoid such exceptions by following a simple rule: "if sharing objects through a common import, make sure that only that module declares the types of the objects shared."

5.1.2 Inexpressive class resolution

Problem Even with the modified resolution algorithm (§5.1.1), developers remain unable to choose between alternative bindings for a particular fully-qualified name. For example, with the original algorithm, a class within *XSLT* cannot refer to *XSLT*'s `ParserX`, while with the modified algorithm, the same class cannot refer to *XMLParser*'s `ParserX`.

We would like to guarantee, without changing the underlying language, that the programmer can *always* choose which class of the available alternatives he wants to use.

Solution Add support for renaming of exports by importers. Since the source code within the importer will not be "aware" of the renaming, we refer to this type of renaming more specifically as *module-boundary renaming*.

With this feature, developers can now import *XMLParser*'s `ParserX` under a different name, e.g. `ImportedParser`. Therefore, a class in *XSLT* can now access *both* `ParserX`s, something not possible before. The resulting resolution semantics is shown in Fig. 5.3, and the actual code change required in the *XSLT*'s module file is:

```
import XMLParser with ParserX as ImportedParser;
```

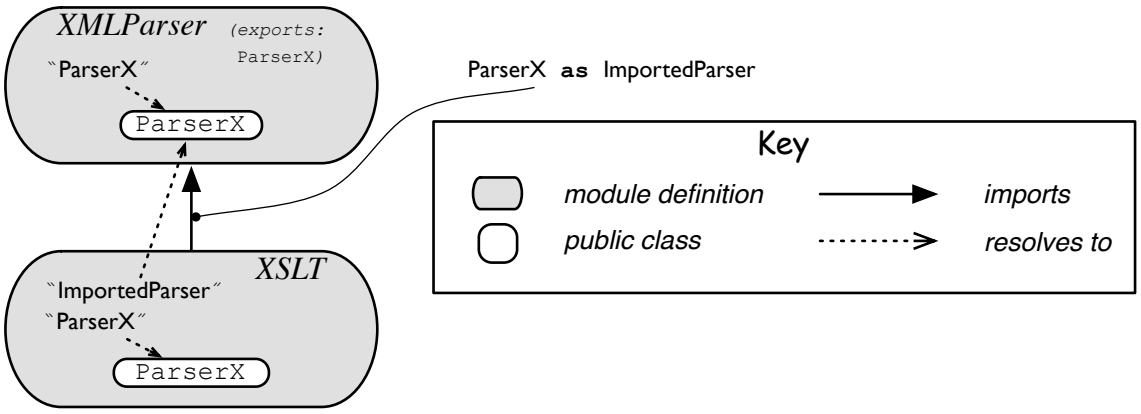


Figure 5.3: Adapted class resolution with module-boundary renaming

Note that switching between name bindings, as well as making the otherwise hidden bindings accessible, is now possible with simple, module-local operations — no module-wise refactoring, or modifying of imported modules, is required.

Now, suppose that *XSLT* needs to use `ParserX` from version *2.0* of *XMLParser*, and `ParserX` from version *3.0* of *XMLParser*. Such a scenario was impossible to express with previous semantics. With the adapted semantics, however, the solution is simple, as shown in Fig. 5.4.

The Java Module System already supports selective exporting, i.e. leaking only a part of the interface. It does not support *selective importing*, i.e. allowing developers to import a subset of the exported classes coming from the imported modules (in order to bind to a local class instead). Selective importing would make the module system robust against incremental interface evolution of imported modules, which we achieved with an adapted class resolution. However, selective importing would *not* allow the scenario in Fig. 5.3 nor the one in Fig. 5.4, both of which are possible through module-boundary renaming.[1]

[1]Module-boundary renaming can be simulated in an ad-hoc manner with an intermediate module in the

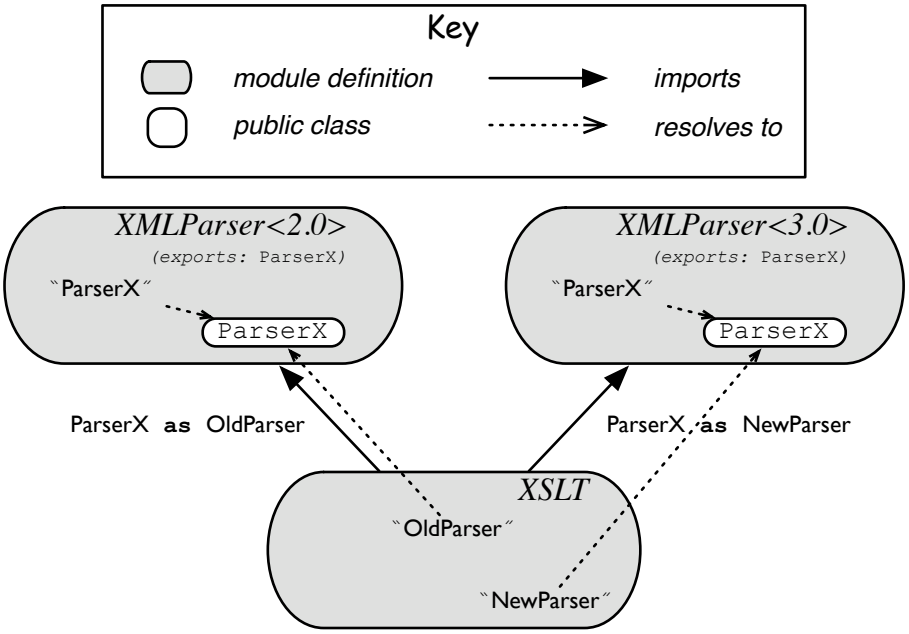


Figure 5.4: Accessing two different versions from a single context

5.2 Inflexible module instantiation

Problem In JMS, each module definition can only have a *single* module instance, i.e. its module generators are applicative even across applications. This means that all clients necessarily have to share the module's static data and types, which is often considered desirable, because it saves space, and again prevents some `ClassCastException`s by forcing fewer definitions a type reference can resolve to.

However, suppose that *XSLT* and *ServletEngine* depend on *XMLParser* (our original example, Figure 1.2, page 27). Furthermore, suppose that *XSLT* and *ServletEngine* (i) rely on conflicting invariants of the internal state of *XMLParser*, or (ii) must run concurrently to achieve a high throughput, but *XMLParser* does not guarantee correct operation in such a concurrent environment.

With the existing proposal, the only solution available to us in case (i) is to make the two invariants somehow compatible. In case (ii), we need to rewrite *XMLParser*'s code (assuming we have access) to add sufficient locking to handle concurrent accesses from multiple users. Both alternatives are often time-consuming and error-prone tasks, but are necessary if *XSLT* and *ServletEngine* have to share data or types through *XMLParser*.

Solution The sharing of data or types through a common import is infrequent, especially across different programs. In many such cases, we can replicate the common import as required. This way the users can maintain conflicting invariants on separate instances

import hierarchy, which extends (but adds no state or functionality) and exports the imported class. Each such hack therefore introduces a dummy "link class" in the class hierarchy for each renaming.

of the module definition, since they use independent static data. We can also avoid unneeded contention in the imported module definition, allowing all importers to execute static methods in parallel without worrying about breaking each other's invariants.

The fundamental point here is that we should give module developers a choice of whether they want a shared or a new instance of an imported module. We achieve this by allowing small annotations on the import statements within module files, as well as on modules themselves.

Here, we give an informal overview of the alternatives that allow expressing a wide variety of sharing scenarios. First, we look at the annotations for the import statements.[2]

IMPORT OPTION	SHORT DESCRIPTION
import m	Uses the Java Module System's sharing policy. (Can be overridden by **replicating**.)
import shared m	(explicitly) Requests a shared instance of m.
import own m	Requests a separate instance of m.
import m **as** amn	Requests an instance, which is shared under name amn.

The difference between the first two options is subtle, but important, and will be explained later. The common semantics of both is to request a shared instance of a particular module definition. In the third case, the client module is requesting an instance of the imported module definition just for itself. In the last case, a module instance is created under a name, amn: if another module imports the module definition (named m) as amn, then they share the same instance. The last option is here to cover the general case.[3]

However, there are cases where the developer would want to specify module's own *replicating policy*. If they know that a module is not concurrency-safe, then they would tag it with **replicating**, and so prevent automated sharing; if they wanted to track some system-wide information, they would tag it with **singleton**, and so force sharing.

ANNOTATION	SHORT DESCRIPTION
(no annotation)	Instantiation depends solely on the importer's policy.
replicating	Default import of this module results in a new instance.
singleton	Always shares a single instance (ignores importer's policy).

Next, we define how the different annotations interact. The intended meaning of the '**replicating**' flag is "use this module as shared at your own risk," while the meaning of the '**singleton**' flag is "this module only makes sense if there is a single instance of it." From this, we decided that '**replicating**' can be overridden by the client module, whereas

[2] Keywords are introduced for the sake of clarity. A real system might use different syntax.

[3] In fact, the last option can simulate the **shared** and the **own** cases: the former by always choosing a pre-defined constant for amn, and the latter by choosing a unique value for amn.

'**singleton**' cannot. A more expressive approach would allow custom annotations, where the developer would specify the override direction.

We effectively have three different types of dependency between the importing and the imported superpackage: *shared*, *own*, and *as*. The following table summarises the above-described interaction between different annotations, and shows how we put the intended semantics before safety in a concurrent environment.

IMPORTING \ IMPORTED	*default*	**replicating**	**singleton**
default	*shared*	*own*	*shared*
shared	*shared*		
own	*own*		
as	*as*		

Therefore, if *XSLT* and *ServletEngine* required the same version of *XMLParser*, but had incompatible invariants on *XMLParser*'s static data, and/or wanted to run safely and efficiently in parallel, then either (or both) would add **own** to the import statement:

```
module XSLT {
  ... import own XMLParser; ...
}
```

```
module ServletEngine {
  ... import own XMLParser; ...
}
```

On the other hand, if the developer of *XMLParser* knows that the module is not concurrency safe and/or has a flexible invariant, he can annotate its source as follows:

```
replicating module XMLParser { ... }
```

In both of the above cases, we end up with the module instances as shown in Fig. 5.5.

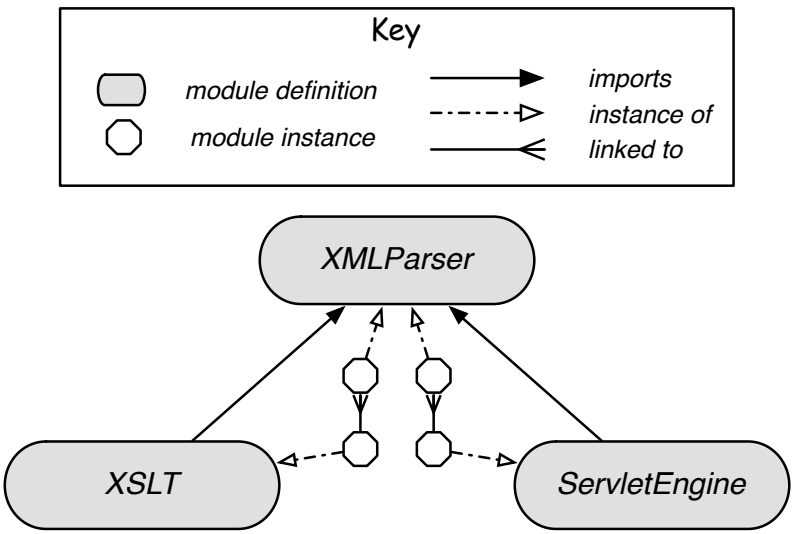


Figure 5.5: Generating multiple instances of a single module definition

5.3 Shallow validation

The Java Module System defines an operation called *shallow validation*, which checks if a fully-qualified name of a class within a module instance clashes with any of the fully-qualified names of classes exported from imported module instances. Shallow validation is performed by default during initialisation of a module definition (but can also be turned off by the administrator) and aborts the process if any matches are found.

There is also a *deep validation* (not performed by default), which performs shallow validation and additionally validates the dependencies of classes in the module definition. We say validation to mean shallow validation.

Let us consider the example in Fig. 5.1 (page 112) again. If no validation occurs, "`ParserX`" resolves to the definition in `XMLParser` in the original semantics. However, if validation *is* performed, it fails, since there is a name clash, and prevents execution.

If the developer is aware of the name clash and wants to use *XMLParser*'s `ParserX` (to allow for sharing between the module definitions), he has to disable the validation for *XSLT* (validation can be disabled per module definition).

The developer cannot force the classes in *XSLT* to use the local `ParserX`: if he leaves the validation on, it will tell him that there is a name clash, and execution will stop. Even if execution was to somehow continue (by ignoring the exceptions thrown), all references to `ParserX` would still resolve to *XMLParser*'s `ParserX`.

Therefore, the only guarantee validation gives is that, if it succeeds, there will be no unexpected behaviour due to the unintuitive class resolution semantics. The draft specification [51] states that names clashes should be avoided, and are prevented by validation; however, as shown in this chapter, throwing an exception when a name clash occurs makes any software design in this system fragile. As the validation is enabled by default, the class resolution ordering is, in most cases, redundant.

Since validation prevents many realistic (and useful) scenarios from ever executing, we did not include it in our formalisation of JMS. However, if a developer intended to share objects of some type across module instances with our reversed resolution algorithm, a form of shallow validation applied only for that specific type and only to a specific part of the module import graph would be useful — we leave this as an option for future work.

5.4 A stronger form of information hiding

JMS enables a form of component-level information hiding with (i) selective exporting, and (2) optional re-exporting. However, even though its repositories provide a limited form of context-sensitive importing, its module visibility remains transitive (§1.1.5).

For example, if a module, *M*, imports another, *N*, which in turn imports a third, *O*, then *N* can hide any name exported by *O* from *M*. However, the developer of *M* is free to

directly import *O*. In JMS, all instances are shared, so *N* cannot encapsulate neither *O*'s exported names, nor *O*'s accessible static data, from *M*. For this reason, we refer to JMS's information hiding as *weak information hiding*.

Our solution to the problem of inflexible module instantiation allows developers to define module instantiation policies. Through these policies, a module can create its own instances of the module definitions it imports. Continuing with the above example, if *N* creates its own instance of *O*, then it encapsulates the accessible static data of that instance from *M*. We refer to this type of information hiding as *instance-based information hiding*.

5.5 Conclusion

Using Lightweight Java Module System, we have uncovered two key deficiencies of JMS, and informally described our proposals for fixing them.

OSGi uses the same unintuitive name resolution as JMS. Both .NET and OCaml use module-prefixed name references to select among the available imported names, while an ambiguous name reference throws an error at compile-time. Jiazzi's unit files fix package bindings to specific Java packages, and thus a package-qualified name can only refer to a single entity; references to unit names within the underlying language are not supported. Our solution provides a user with an intuitive class resolution, where one is always able to resolve an ambiguous reference to any of the visible alternative even though the underlying language lacks support for module-prefixed references.

Our tests showed that all analysed module systems enforce sharing of module instances within a single application, and that sharing of module instances across multiple applications is either enforced (OSGi), or not supported (e.g. .NET, OCaml, Jiazzi). While JMS has similar sharing semantics to OSGi, our solution is, in this respect, more expressive than all of the mentioned module systems. In our solution, module instances can be shared or not, whether in a single program or across multiple programs.

The next chapter formalises our proposals, and checks whether type soundness is preserved. In Chapter 8, we develop an even stronger form of information hiding.

6

Improved Java Module System (iJAM)

In this chapter, we formalise the improvements to the Java Module System that we proposed in the previous chapter. This is done by adapting our existing formalisation (LJAM). The resulting language is called the Improved Java Module System (iJAM).

The following sections show the difference between LJAM's and iJAM's syntax (§6.1), operational semantics (§6.2), type system (§6.3), type checking (§6.4), and proof of type soundness (§6.5).

The full Ott definition, the complete Isabelle/HOL proof of type soundness, and various other related documents can be found at the following address:

`http://www.cl.cam.ac.uk/research/pls/javasem/iJAM/`

6.1 Syntax

The core language remains unchanged; we only modify the module-level language. The following section shows the changes to the user and inner syntax.

6.1.1 User syntax

The user syntax of module files, mf, for iJAM is a subset of the one for LJAM:

$$\mathit{repl}\ \mathbf{superpackage}\ mn\ \{\ \overline{\mathbf{member}\ pn;}\ \overline{\mathit{imp};}\ \overline{\mathbf{export}\ \mathit{fqn};}\ \}$$

Compared to the user syntax of LJAM's module files (§4.2.3), the definition is now prefixed with a replication modifier, $repl$, while the previous import statements, '$\overline{\mathbf{import}\, m;}$', are replaced with '$\overline{imp;}$', which can specify client's replication policy and boundary renaming — see Fig. 6.1. Meta-variable amn stands for abstract module name.

$repl$	::=			replication modifier
	|			default
	|	$\mathbf{replicating}$		replicating
	|	$\mathbf{singleton}$		singleton
imp	::=			import statement
	|	$\mathbf{import}\ m\ br$		default
	|	$\mathbf{import\ shared}\ m\ br$		shared
	|	$\mathbf{import\ own}\ m\ br$		own
	|	$\mathbf{import}\ m\ \mathbf{as}\ amn\ br$		as
br	::=			boundary renaming ($(fqn \times fqn)$ list)
	|		M	no renaming
	|	$\mathbf{with}\ fqn_1\ \mathbf{as}\ fqn'_1,\ ...,\ fqn_k\ \mathbf{as}\ fqn'_k$	M	renaming pairs

Figure 6.1: iJAM's changes to user syntax

6.1.2 Inner syntax

The inner syntax for a module definition, md^{c}, has changed following the changes in the user syntax. The added and updated parts of the inner syntax are shown in Fig. 6.2.

As in LJAM, the module instance identifiers are stored in repository caches. LJAM's caches, however, simply mapped module definitions, md^{c}'s, to module instance identifiers of their module instances: '$md^{\mathsf{c}} \rightharpoonup mi$'. In iJAM, the import dependencies need to be taken into account, so the cache type becomes: '$md^{\mathsf{c}} \rightharpoonup (imp\_dep \rightharpoonup mi)$'.

The two user syntax terms, $repl$ and imp, correspond to the replication policy annotations described in Chapter 5, on a module and on its client's import statement, respectively. The import dependency, imp_dep, is the replication policy that results from the interaction between the two — this interaction is also described in the previous chapter. The mi parameter to its $\mathbf{Own}$ case is a reference to the module instance created (not its owner).

The module hierarchy, MH, stores the connection between module instances (through their identifiers). In LJAM, this was simply '$mi \rightharpoonup \overline{mi}$', but in iJAM each imported module instance is also associated with the appropriate boundary renaming of class names, so MH's type becomes '$mi \rightharpoonup \overline{mi\ br}$'. This way the class lookup function can easily update the name of the class it is looking for when crossing module boundaries.

Finally, the initialisation action, '$rn\,.\,\mathbf{initialise}\,(imp)\,;$', now takes the whole import statement, imp, instead of simply the name of the imported module, m. This way the semantics of administration actions can take replication policy into account.

md^{c}	::=			module definition
	|	$repl\ \mathbf{module}\ mn\ \{\ \overline{cld^{\mathsf{c}}}\ \overline{imp_k}^{\,k}\ \overline{fqn}\ \}$		def.
ϕ	::=			R cache ($md^{\mathsf{c}} \rightharpoonup (imp\_dep \rightharpoonup mi)$)
	|	$[\,]$	M	empty repository's cache
	|	$\phi\,[md^{\mathsf{c}} \mapsto\ imp\_dep\ \mapsto\ mi]$	M	map imp_dep to mi in map for md^{c}
	|	$\phi \setminus md^{\mathsf{c}}$	M	remove mapping for md^{c}
imp_dep	::=			import dependency
	|	**Shared**		default import
	|	**Own** mi		instance of imported module
	|	**As** amn		ref. to imported module
MH	::=			module hierarchy ($mi \rightharpoonup mhv$)
	|	$[\,]$	M	empty module hierarchy
	|	$[mi\ \mapsto\ mhv]$	M	maps mi to its def. and imports
	|	$MH_1\ ..\ MH_k$	M	composes many
mhv	::=			module hierarchy value ($md \times \overline{mibr}$)
	|	$(md,\ \overline{mibr})$	M	def.
$mibr$	::=			assoc. boundary renaming ($mi \times br$)
	|	$mi\ br$	M	def.

Figure 6.2: iJAM's changes to the inner syntax

6.2 Operational semantics

The semantics of statement reduction changes only through the adapted class resolution, which has to allow for module-boundary renaming. On the other hand, module definition initialisation now has to include replication policies, and to make user-provided boundary-renaming available to class resolution.

6.2.1 Adapted class resolution

Here, we implement the class resolution described in the previous chapter. The high-level algorithm is to search (1) the core module, (2) the module instance itself, and, finally, (3) the imported module instances (recursively). The main difference with the LJAM's algorithm (§4.3.1) is the swapping of (2) and (3).

$\mathbf{find\_cld}\,((RC,\ MH), mi.pn, fqn)\ :\ (P \times ctx \times fqn) \to (ctx \times cld)_{opt}\ =$
  if $\neg(\mathbf{no\_core\_renaming}\,(RC,\ MH))$ **then** `None` **else**
  match $\mathbf{find\_cld\_in\_core}\,((RC,\ MH), fqn)$ **with** `Some` $ctxcld \to$ `Some` $ctxcld$
    | `None` $\to$ **match** $MH\ \ mi$ **with**
        `None` $\to$ `None` | `Some` $(repl\,\mathbf{module}\,mn\,\{\,\overline{cld}\,\overline{imp_k}^{\,k}\,\overline{fqn}\,\}, \overline{mibr}) \to$
          match $\mathbf{find\_cld\_in\_self}\,(\overline{cld}, pn, fqn)$ **with**
            `Some` $cld \to$ `Some` $(mi.(\mathbf{package\_name}\,(cld)), cld)$
          | `None` $\to \mathbf{find\_cld\_in\_imports}\,(MH, \overline{mibr}, fqn)$

The reason for the first line of the above function (**no_core_renaming**) is explained in §6.5.1. The functions **find_cld_in_core** and **find_cld_in_self** are practically identical to the ones in LJAM — we do not show the definitions of these functions here.

As for the example illustrating LJAM's class resolution ordering (Fig. 4.3, page 95), suppose we have instances of module definitions *A*, *B*, *C*, *D*, and the core library module, *Core*, all of which are connected as shown in Fig. 6.3. If we started class resolution (**find_cld**) in *A*, the instances would get searched as indicated by numbers in the brackets.

The function that searches the imports, **find_cld_in_imports**, now has to take boundary renaming into account, in addition to information hiding. As mentioned before, each imported module instance mi is now associated with boundary renamings, br.

```
1  find_cld_in_imports (MH, mibr, fqn) : (MH × mibr × fqn) → (ctx × cld)_opt =
2    match mibr with [] → None | (mi, br) :: mibr' →
3    if ¬(acyclic_mh MH ∧ mis_of (mibr') ⊆ dom (MH)) then None else
4    match MH mi with None → None
5                     | Some (repl module mn { cld imp_k^k fqn }, mibr'') →
6    if br[fqn] ∉ fqn ∨ (fqn ∉ dom (br) ∧ fqn ∈ ran (br))
7    then find_cld_in_imports (MH, mibr', fqn) else
8    match find_cld_in_module (cld, br[fqn]) with
9      Some cld → Some (mi.(package_name (cld)), cld) | None →
10       match find_cld_in_imports (MH, mibr'', br[fqn]) with
11         Some ctxcld → Some ctxcld | None →
12           find_cld_in_imports (MH, mibr', fqn)
```

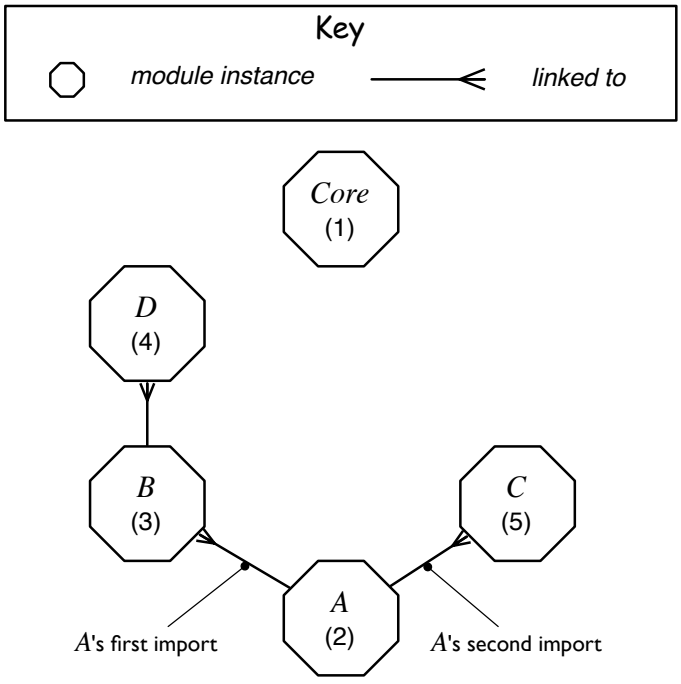


Figure 6.3: iJAM's class resolution order

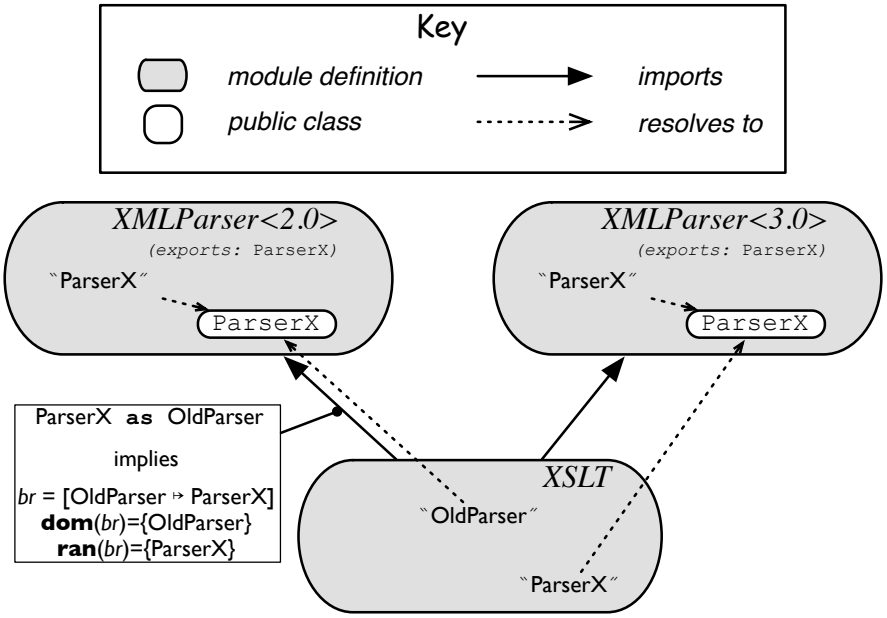


Figure 6.4: Accessing two different versions from a single context

For example, suppose that *XSLT* imports (1) `ParserX` from version *2.0* of *XMLParser* as `OldParser`, and (2) `ParserX` from version *3.0* of *XMLParser* (with no boundary renaming) — see Fig. 6.4. Therefore, the boundary renaming, br, for the first import, mi, is '`ParserX` **as** `OldParser`'. When resolving "`OldParser`" within *XSLT*, we need to apply br to the class name when the search crosses into mi — we write $br[\texttt{OldParser}]$ to mean "*rename* `OldParser` *according to* br if *there is a mapping for it*." Due to the direction of class resolution, we see '`ParserX` **as** `OldParser`' as a map from `OldParser` to `ParserX`, i.e. $[\texttt{OldParser} \mapsto \texttt{ParserX}]$, so the domain of br is $\{\texttt{OldParser}\}$, while its range (co-domain) is $\{\texttt{ParserX}\}$.

Line 6 in the above definition of **find_cld_in_imports** expresses the condition when an import is *not* searched. It states that an import, mi, is ignored if $br[fqn] \notin \overline{fqn}$, i.e. if (possibly renamed) fqn is not exported by mi, *or* if $(fqn \notin \mathbf{dom}\,(br) \wedge fqn \in \mathbf{ran}\,(br))$, i.e. if "$fqn$ *does not have a different name in* mi, and *some other name has* fqn *as a different name in* mi." In other words, the second part of the condition states that boundary renaming is, in fact, renaming, and not aliasing, i.e. the original name is hidden.

Therefore, if fqn = `ParserX`, the first import is ignored, since `ParserX` does not have a different name in it ($\texttt{ParserX} \notin \{\texttt{OldParser}\}$) *and* some other name, `OldParser`, has `ParserX` as a different name in it ($\texttt{ParserX} \in \{\texttt{ParserX}\}$).

6.2.2 Replication policies

The semantics of the install and uninstall administrator actions is identical to that of LJAM, since the two actions are independent from replication policies and boundary renamings.

R_EXISTING_INSTANCE

$$\frac{\begin{array}{l}1.\ \mathbf{imp\_name}\,(imp) = m \\ 2.\ \mathbf{find\_md}\,(RC,\ rn_1,\ m) = (rn_2,\ md^{\mathsf{c}}) \qquad 3.\ RC\,(rn_2) = R_2 \\ 4.\ \mathbf{R\_body}\,(R_2) = (\overline{md^{\mathsf{c}}}_2,\ \phi_2) \quad 5.\ mi' \notin \mathbf{dom}\,(MH) \\ 6.\ \mathbf{imp\_dep\_of}\,(md^{\mathsf{c}},\ mi',\ imp) = imp\_dep \\ 7.\ \phi_2\,(md^{\mathsf{c}},\ imp\_dep) = mi\end{array}}{((RC,\ MH),\ L,\ H,\ \overline{s_l}^{\,l}) \xrightarrow{mi = rn_1\,.\,\mathbf{get\_instance}\,(imp)} ((RC,\ MH),\ L,\ H,\ \overline{s_l}^{\,l})}$$

R_NEW_INSTANCE

$$\frac{\begin{array}{l}1.\ \mathbf{imp\_name}\,(imp) = m \\ 2.\ \mathbf{find\_md}\,(RC,\ rn_1,\ m) = (rn_2,\ md^{\mathsf{c}}) \qquad 3.\ RC\,(rn_2) = R_2 \\ 4.\ \mathbf{R\_body}\,(R_2) = (\overline{md^{\mathsf{c}}}_2,\ \phi_2) \qquad 5.\ mi' \notin \mathbf{dom}\,(MH) \\ 6.\ \mathbf{imp\_dep\_of}\,(md^{\mathsf{c}},\ mi',\ imp) = imp\_dep' \\ 7.\ \phi_2\,(md^{\mathsf{c}},\ imp\_dep') = \mathbf{null} \\ 8.\ md^{\mathsf{c}} = repl\ \mathbf{module}\ m\ \{\ \overline{cld^{\mathsf{c}}}\ \overline{imp_k}^{\,k}\ \overline{fqn}\ \} \\ 9.\ ((RC,\ MH),\ L,\ H,\ \overline{s_l}^{\,l}) \xrightarrow{\overline{mi_k = rn_2\,.\,\mathbf{get\_instance}\,(imp_k)}^{\,k}} ((RC',\ MH'),\ L,\ H,\ \overline{s_l}^{\,l}) \\ 10.\ mi \notin \mathbf{dom}\,(MH') \quad 11.\ \mathbf{imp\_dep\_of}\,(md^{\mathsf{c}},\ mi,\ imp) = imp\_dep \\ 12.\ \vdash_{mi} md^{\mathsf{c}} \rightsquigarrow md \qquad 13.\ \overline{\mathbf{imp\_br}\,(imp_k) = br_k}^{\,k} \\ 14.\ MH'' = MH'\,[mi \mapsto (md,\ \overline{mi_k\ br_k}^{\,k})] \qquad 15.\ RC'\,(rn_2) = R'_2 \\ 16.\ \mathbf{R\_body}\,(R'_2) = (\overline{md^{\mathsf{c}}}_3,\ \phi_3) \\ 17.\ \mathbf{R\_update}\,(R'_2,\ \overline{md^{\mathsf{c}}}_3,\ \phi_3\,[md^{\mathsf{c}} \mapsto imp\_dep \mapsto mi]) = R''_2 \\ 18.\ RC'' = RC'\,[rn_2 \mapsto R''_2] \qquad 19.\ (RC'',\ MH'') \vdash_{mi} md\end{array}}{((RC,\ MH),\ L,\ H,\ \overline{s_l}^{\,l}) \xrightarrow{mi = rn_1\,.\,\mathbf{get\_instance}\,(imp)} ((RC'',\ MH''),\ L,\ H,\ \overline{s_l}^{\,l})}$$

Figure 6.5: iJAM's operational semantics for initialisation actions

On the other hand, obtaining an existing or a new module instance must take both of the new concepts into account. Figure 6.5 shows the detailed semantics of the two cases. Here, we explain the differences with the corresponding rules in LJAM.

The first rule, R_EXISTING_INSTANCE, proceeds as follows: (1) with **imp_name**, extracts the module name, m, from the given import statement, imp; (2) starting the search in repository named rn_1, finds a module definition named m, md^{c}, within repository named rn_2; (3-4) inspects the contents of the repository named rn_2; (5) finds a fresh module instance identifier, mi'; (6) with **imp_dep_of**, generates the import dependency, imp_dep, from the annotations in md^{c}, from mi', and from imp; and, (7) finds an existing mapping, mi, for md^{c} and imp_dep in rn_2's cache.

The function **imp_dep_of** creates the appropriate import dependency given the annotation on both the import statement and the imported module definition — it respects

the function presented in §5.2 (page 117). In case the resulting dependency is **Own**, the function needs a way to generate a unique import dependency that will not clash with any other when cached. So, a fresh module instance, mi, is passed into the function, which outputs '**Own** mi' when a fresh module instance is required. Therefore, step (5) of R_EXISTING_INSTANCE is there just so step (6) can go through. [1]

The second rule, R_NEW_INSTANCE, describes the case where no appropriate module instance is already cached. It (1-6) takes a few steps as before, (7) finds that no module instance is mapped for md^{c} and imp_dep, (8) inspects the contents of md^{c}, (9) recursively creates module instances named $\overline{mi_k}^k$ from import statements $\overline{imp_k}^k$, resulting in program state $(RC',\ MH')$, (10) finds a fresh module instance identifier, mi, (11) generates the import dependency, imp_dep, from md^{c}, mi, and imp, (12) creates md, a module instance of md^{c}, (13) extracts boundary-renamings, $\overline{br_k}^k$, from module's import statements, $\overline{imp_k}^k$, with **imp_br**, (14) maps mi to md, $\overline{mi_k}^k$, and $\overline{br_k}^k$ in MH', producing MH'', (15-16) finds the repository named rn_2 and inspects its contents, (17) caches mi for md^{c} under import dependency imp_dep in R_2', producing R_2'', (18) re-maps rn_2 to R_2'', and (19) typechecks md in the final context.

Steps (10-11) generate a fresh import dependency after the recursive initialisation has completed. This is required, since mi' might have been bound during step (9). An alternative approach would be to bind mi' to a dummy value before step (9) to guarantee that it does not get overridden.

Step (14) associates imported module instances to the corresponding boundary renamings within the module hierarchy. The adapted class resolution (§6.2.1) uses these name pairs to appropriately rename the target class name when crossing module boundaries.

6.3 Type system

The definitions of the type and the subtyping relation are identical to those of LJAM — see §4.4.1 (page 101) and §4.4.2 (page 101).

6.4 Type checking

Most well-formedness relations are identical to those of LJAM. Only minor changes appear in relations for well-formedness of the adapted inner structures. These are the module hierarchy, MH, the repository cache, ϕ, and the module instance, md. The updated relations are shown in Fig. 6.6.

[1] Since mi' is fresh, and mi is not (it is in cache), they are necessarily distinct, which implies that **Own** module instances stored in the cache cannot be reused. An alternative would be to not cache them at all.

WF_MH

$$\frac{1.\ \mathbf{acyclic\_mh}\ MH \qquad 2.\ \forall mi \in \mathbf{dom}\,(MH)\,.\,\exists md\,\overline{mibr}\,.\,MH\,(mi) = (md,\ \overline{mibr})\ \wedge\ (RC,\ MH) \vdash_{mi} md}{RC\ \vdash\ MH}$$

WF_RMIS

$$\frac{1.\ \forall md^{\mathsf{c}}\ imp\_dep\,.\,\forall mi\,.\,\phi\,(md^{\mathsf{c}},\ imp\_dep) = mi\ \longrightarrow\ mi \in \mathbf{dom}\,(MH)}{MH\ \vdash\ \phi}$$

WF_MODULE

$$\frac{\begin{array}{ll}1.\ \overline{\mathbf{full\_name}\,(cld_j) = fqn_j}^{\,j} & 2.\ \mathbf{distinct}\,(\overline{fqn_j}^{\,j}) \\ 3.\ \overline{(RC,\ MH) \vdash_{mi} cld_j}^{\,j} & 4.\ \mathbf{acyclic\_clds}_{mi}(RC,\ MH) \\ 5.\ MH\,(mi) = (md,\ \overline{mibr})\ \wedge\ \mathbf{no\_core\_renaming\_in\_mibrs}\,((RC,\ MH),\ \overline{mibr}) & \end{array}}{(RC,\ MH) \vdash_{mi} repl\ \mathbf{module}\ mn\ \{\ \overline{cld_j}^{\,j}\ \overline{imp_k}^{\,k}\ \overline{fqn}\ \}}$$

Figure 6.6: iJAM's updated well-formedness relations

The WF_MH rule exposes some of the changed structure of MH, but the premises are essentially the same: MH must be acyclic (§4.3.1), and all module instances identifiers are mapped to well-formed module instances.

Similarly, the cache well-formedness rule still states that the co-domain of the cache must be a subset of the module hierarchy's domain. The premise is more complicated than in LJAM due to the more complex structure of the cache.

The first four premises of WF_MODULE are identical to those in LJAM. The last one has been added: it states that there can be no renaming of the exported core library classes by this module instance, mi — this property is explained further in §6.5.1 (page 129).

6.5 Proof of type soundness

One of the key lemmas in the LJAM's proof, which iJAM's adapted class resolution breaks, is Lemma 15 (§4.4.4, page 102). It says that if we look for a class with name fqn in context ctx, and we find a class definition, cld', in context ctx', then we will get the same result if we start the search at ctx' instead:

$$\mathbf{find\_cld}\,(P, ctx, fqn)\ =\ \texttt{Some}\ (ctx', cld') \quad \Longrightarrow$$
$$\mathbf{find\_cld}\,(P, ctx', fqn)\ =\ \texttt{Some}\ (ctx', cld')$$

In iJAM, this lemma does not hold any more, because fqn is not necessarily the same as the fully-qualified name of cld' — this is due to boundary renaming. For this reason, we modify the lemma by replacing fqn in the goal with $\mathbf{full\_name}\,(cld')$.

Lemma 21 (Target context equivalence for class lookups — adapted)**.**

$$\textbf{find\_cld}\,(P, ctx, fqn) \;=\; \texttt{Some}\;\,(ctx', cld') \quad \Longrightarrow$$
$$\textbf{find\_cld}\,(P, ctx', \textbf{full\_name}\,(cld')) \;=\; \texttt{Some}\;\,(ctx', cld')$$

Explanation. If we look for a class named fqn in context ctx, and we find a class definition cld' in context ctx', then we will get the same result if we start the search in ctx' and look for a class with the same name as cld'*, instead.*

However, the lemma did not hold initially (when developing the semantics), because the two function calls first search within the core libraries, each with a possibly different class name, which can therefore lead to a different result. For example, suppose the first call searches for a class named `A`: it does not find a class named `A` in the core libraries, but finds a class named `B` in one of the imported modules. The second call then searches for a class `B`, and finds a class `B` in the core libraries. The contexts of the two class definitions, as well as the definitions themselves, are different.

If we could not use the modified lemma in iJAM's proof, we would not be able to reuse large parts of LJAM's proof. (See Appendix A for the dependency graph of the lemmas and theorems presented in the thesis.)

We solved this problem by placing a well-formedness condition on boundary renaming (shown in §6.5.1), which prevents module definitions from renaming a class *from* and *to* a name already exported by the core library module. In fact, already the *from* part makes the modified lemma hold again, but we added the *to* part, too, to avoid unexpected class resolution results. Without the *to* part, developers could rename an imported class to a class already exported by a core library — any use of that name would resolve to the class in the core library, not in the imported module, which would be unintuitive.

The restrictions ensure that a class reference resolves to a core library class *iff* the reference is a name of a class exported by the core library. It is not clear to us whether iJAM is type-sound without the these restrictions; however, this is not important, since, as described above, we would obtain less intuitive semantics.

6.5.1 Well-formedness for boundary renaming

The main relation to restrict boundary renaming (**no_core_renaming**) is defined on a program, P, which is in the NCR_DEF decomposed into '$(RC,\ MH)$' — see Fig. 6.7. The rule states that the helper relation, **no_core_renaming_in_mibrs**, holds for all boundary renamings found within MH.

The second rule defines the **no_core_renaming_in_mibrs** relation. It states that *if we can find a class within the core library, its name cannot be used in any boundary renaming, i.e. a class cannot be renamed* from *or* to *a class exported by the core library.*

With this restriction in place, Lemma 21 (page 129) becomes valid, and we can reuse LJAM's proof of type soundness with only trivial modifications.

$$\frac{\forall mi \in \mathbf{dom}\,(MH)\,.\,\exists md\,\overline{mibr}\,.\,\left(\begin{array}{l} MH\,(mi) = (md,\,\overline{mibr}) \wedge \\ \mathbf{no\_core\_renaming\_in\_mibrs}\,((RC,\,MH),\,\overline{mibr}) \end{array}\right)}{\mathbf{no\_core\_renaming}\,(RC,\,MH)}\ \text{NCR\_DEF}$$

$$\frac{\forall br \in br_1\,..\,br_k\,.\,\forall fqn\,.\,\left(\begin{array}{l} (\exists ctx\ cld\,.\,\mathbf{find\_cld\_in\_core}\,(P,\,fqn) = (ctx,\,cld)) \longrightarrow \\ \quad fqn \notin br \end{array}\right)}{\mathbf{no\_core\_renaming\_in\_mibrs}\,(P,\,mi_1\,br_1\,..\,mi_k\,br_k)}\ \text{NCRIM\_DEF}$$

Figure 6.7: No renaming of classes exported by the core library

6.6 Reuse within the definitions and proof scripts

Since LJAM is an extension of LJ, and iJAM is based on LJAM, they share much of the semantic definitions. Specifically, the language statements, e.g. the method call statement, have syntactically identical definitions in all three languages; also syntactically identical are the statement well-formedness and reduction relations — that is, the semantics of these relations differ through different definitions of the syntactically identical judgements (e.g. the class resolution judgement) used in their rules.

The proof scripts for the progress and well-formedness of the statement reduction relation are practically identical for all the three languages (5 lines out of 350 lines differ). This is achieved by carefully abstracting the key lemmas, e.g. Lemma 21 mentioned in this chapter. Due to such abstractions, we were able to achieve high reuse within both the definitions and their proof scripts as shown by the two diagrams in Fig. 6.8 (relative area corresponds to relative number of lines of definition/proof script).

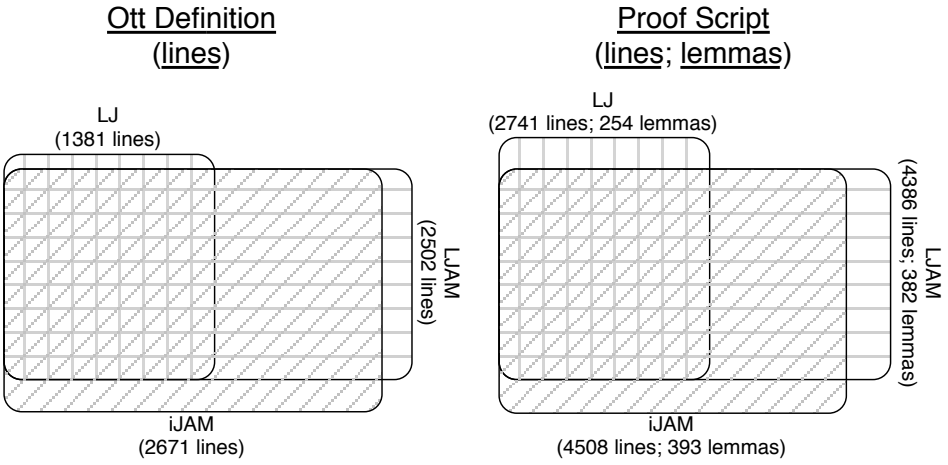


Figure 6.8: Reuse within the language definitions and their proof scripts

7 Implementation

In this chapter, we describe our proof-of-concept implementation of both LJAM and iJAM. We only describe the most interesting parts of the implementation, focusing on module initialisation (§7.2) and class resolution (§7.3). More information, the documentation, and the complete source code can be found online:

`http://www.cl.cam.ac.uk/research/pls/javasem/iJAM/`

7.1 Overview

We implement our module system on top of Java. The system can model the semantics of either LJAM or iJAM, and the code encapsulated by module definitions can contain any valid Java code. The parser for the module files was generated with JavaCC [26], a Java compiler compiler. The system simulates the compilation of module files into module definitions, and implements the repository context, RC, the module hierarchy, MH, administration actions (according to replication policies), and class resolution (which can include boundary renaming). Since we do not modify the JVM, sharing of renamed classes is not supported (§7.6).

7.2 Creation of module instances

When a module instance is required, the repository that holds the corresponding module definition checks the import dependency, which is specified by the client and the imported module definitions (§5.2). The repository either returns an existing instance, or creates a new one. A repository stores its module instances in a map of maps:

```
Map<ModuleDefinition, Map<ImportDependency, Module>>
```

The above Java type corresponds exactly to iJAM's repository cache, ϕ, in our formal definition (Fig. 6.2, page 123): $md \rightharpoonup (imp\_dep \rightharpoonup mi)$. LJAM's cache, $md \rightharpoonup mi$, is simulated by always using **Shared** for imp_dep.

Since `ImportDependency` acts as a key in the inner maps, we define its `equals` and `hashCode` methods. Both methods guarantee uniqueness for an **own**-type import dependencies, equality for **shared**-type dependencies, and equality based on developer-assigned codenames (abstract module names, amn) for **as**-type dependencies.

This implementation allows us to use Java's `HashMaps` to determine if a module instance for a module definition with a particular import dependency already exists in the cache simply by using the map's `contains` method.

7.3 Class resolution

In LJAM, the class resolution searches (1) the core library module, (2) the imported modules (recursively), and, finally, (3) the module itself. For iJAM, we have swapped steps (2) and (3), and added module-boundary renaming. The reasons for this are explained in §5.1 (page 111), while the formalisation of the class resolution can be found in §6.2.1 (page 123). By default, classloading is done according to iJAM's semantics; however, in compatibility mode, classloading behaves as specified in LJAM (§4.3.1, page 93).

The search is performed on module instances within the module hierarchy (MH). As explained in §1.1.3, module instances are classloaders in disguise, which delegate the class resolution according to the semantics of either LJAM or iJAM.

In our implementation, each module instance (an object of class `Module`) holds references to module instances it imports. Each of those imports is also associated with a module-boundary renaming map — these renaming maps are ignored when executing according to LJAM's semantics.

Apart from the search order and boundary renaming, we also have visibility constraints. That is, a class in the imported module definition is visible only if it is (recursively) exported and public.

The implementation of the `loadClass` function is shown in Fig. 7.1. Note that the classloader first checks (with `findLoadedClass`) if it has loaded that class before —

```
1  public Class<?> loadClass(final String name)
2      throws ClassNotFoundException {
3    // CHECK CLASSLOADER'S CACHE
4    Class<?> cl = findLoadedClass(name);
5    if (cl != null)
6      return cl;
7    // CHECK SYSTEM'S CORE CLASSES
8    if (name.startsWith("java.") || name.startsWith("javax."))
9      return findSystemClass(name);
10   // CHECK ITS OWN CONTENTS (if in iJAM mode)
11   if (!Runtime.COMPATIBILITY_MODE && md.members.contains(name)
12       && (cl = loadClassInSelf(name)) != null)
13     return cl;
14   // CHECK IMPORTS (with importer's renaming)
15   for (final Module link : imports.keySet()) {
16     final Map<String, String> renaming = imports.get(link);
17     String nameInImport = name;       // LJAM → ignore renaming.
18     if (!Runtime.COMPATIBILITY_MODE)  // iJAM &&
19       if (!renaming.containsKey(name)) {    // name not in domain,
20         if (renaming.containsValue(name))   // but found in range
21           continue;                         // → skip this import.
22       } else nameInImport = renaming.get(name);
23     if (link.md.exports.contains(nameInImport)) // Exported?
24       try {
25         cl = link.loadClass(nameInImport);  // Recursive search.
26         cl = renameClass(cl, name);         // Explained in § 7.6.
27         if (Modifier.isPublic(cl.getModifiers())) // Public?
28           return cl;
29       } catch (final ClassNotFoundException ex) {}
30   }
31   // CHECK ITS OWN CONTENTS (if in LJAM mode)
32   if (Runtime.COMPATIBILITY_MODE && md.members.contains(name)
33       && (cl = loadClassInSelf(name)) != null)
34     return cl;
35   // THROW EXCEPTION SINCE ALL ABOVE FAILED
36   throw new ClassNotFoundException(name);
37 }
```

Figure 7.1: Implementation of LJAM's/iJAM's class resolution

if so, it returns the previous result. We do not show the implementation of functions `findSystemClass` and `loadClassInSelf`, which have obvious semantics.

The variable `imports` (lines 15–16) refers to a map from imported modules to corresponding renaming pairs. Since we use Java's `java.util.LinkedHashMap`, the order of keys is preserved, i.e. imports are searched as they are declared in the module file.

Lines 19–23 correspond to the condition described in §6.2.1, which states when class resolution skips an import: lines 19–20 represent '$fqn \notin \mathbf{dom}\,(br) \wedge fqn \in \mathbf{ran}\,(br)$', i.e. "$fqn$ (name being resolved) does not have a different name in the import, *and* some other name has fqn as a different name in the import." The line 23 checks if $br[fqn] \in \overline{fqn}$, i.e. if fqn is exported (possibly under a different name) by the import.

If none of the steps find an appropriate class definition, the function throws an exception (line 36). However, if class resolution is delegated to an imported module instance, which fails to find a class and throws an exception, that exception is ignored (line 29), since the search should continue in the module instance that delegated its class resolution.

7.4 Making the JVM use our code

So far, we have described the code that resolves a type reference to a type definition. Now, we explain how to get the Java Virtual Machine to use this code for type resolution when executing Java code.

The solution is fortunately quite straightforward. When a Java classloader resolves a given type reference to a type definition, the same classloader will be used to resolve all type references within that type definition. Of course, a classloader can always delegate its classloading to another classloader.

Therefore, our implementation simply bootstraps the classloading process, while the JVM makes sure that our classloaders are being used. More specifically, our implementation (1) uses the classloader that corresponds to a module instance to load the module's main class using the above `loadClass` method, (2) creates an instance of that class with Java reflection libraries, and (3) executes the `main` method of the class with no arguments. When the JVM needs to resolve a type reference in that class, it calls the `loadClass` method of the same classloader, which then delegates classloading to classloaders corresponding to imported module instances, as required.

The code within the member classes is free to use custom classloaders to resolve a type reference. Of course, such a type reference (and the type references occurring within the resolved type definitions) will be resolved according to the code within the custom classloader, completely ignoring our classloaders.

7.5 Example runs

In this section, we show a few example runs, which demonstrate the key features of the implementation. Our example is based on the WebCalendar example shown in the introduction (§1.1.1); however, we put all classes into the *default* Java package for a clearer illustration of the features. The source code of the classes can be found in Appendix D. The example program's module-level source code is displayed in Fig. 7.2, while the program's runtime structure is shown graphically in Fig. 7.3.

The '`member default;`' statement within each module file is our syntax for making the (locally available) classes of the *default* Java package part of the module definitions. The *default* package is a package with no name, a special case in our formalisation.

```
superpackage XMLParser     {member default; export Parser;}
superpackage XSLT          {member default;
                            import XMLParser; export Config;}
superpackage ServletEngine {member default;
                            import XMLParser; export Config;}
superpackage WebCalendar   {member default;
                            import XSLT with Config as XSLT_Config;
                            import ServletEngine;}
```

Figure 7.2: The module-level source code for the example program

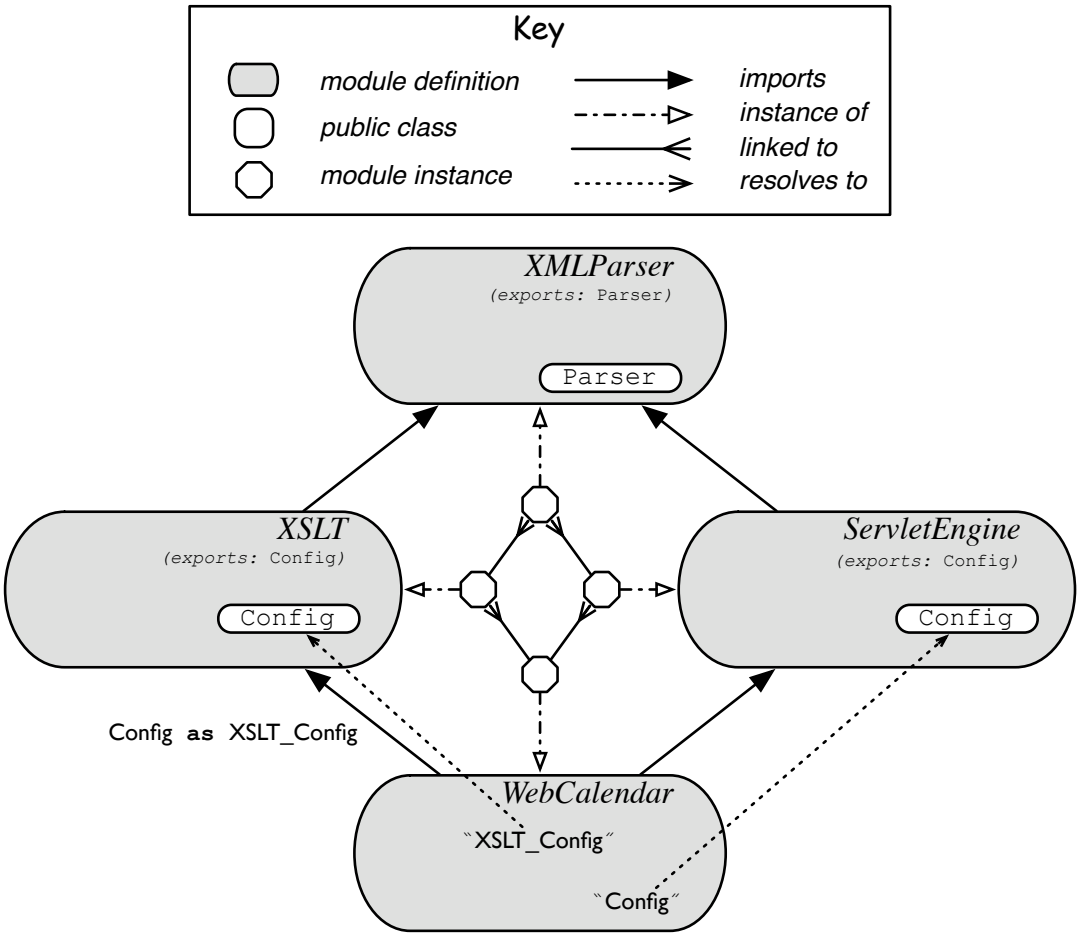


Figure 7.3: The high-level structure of the example program

Both `Config` classes use the *XMLParser*::`Parser`, which tracks the number of its instances using a static variable. *WebCalendar*::`Main` (not visible in the figure) creates instances of "`XSLT_Config`" and "`Config`". Running `Main` outputs:

```
XSLT::Config using 1. instance of Parser.
ServletEngine::Config using 2. instance of Parser.
```

This indicates that there is only one instance of the *XMLParser* module definition. The example also shows that the `Config` class in *XSLT* is correctly ignored when resolving "`Config`" within *WebCalendar*.

Now we modify the existing example by adding a `UnitTest` class to *ServletEngine* and *WebCalendar*, and annotating *ServletEngine*'s import statement with **own**. Of the underlying Java source code, we modify only *WebCalendar*::`Main` to start by running a `UnitTest`. Figure 7.4 gives a pictorial representation.

The program's output is the following:

```
WebCalendar::UnitTest complete.
XSLT::Config using 1. instance of Parser.
ServletEngine::Config using 1. instance of Parser.
```

The "`UnitTest`" class reference resolved to the local class, not the imported one (from *ServletEngine*). Also, both *XSLT* and *ServletEngine* report that they are using the first instance of `Parser` — this is because they used two different `Parser` classes coming from two separate module instances of *XMLParser*.

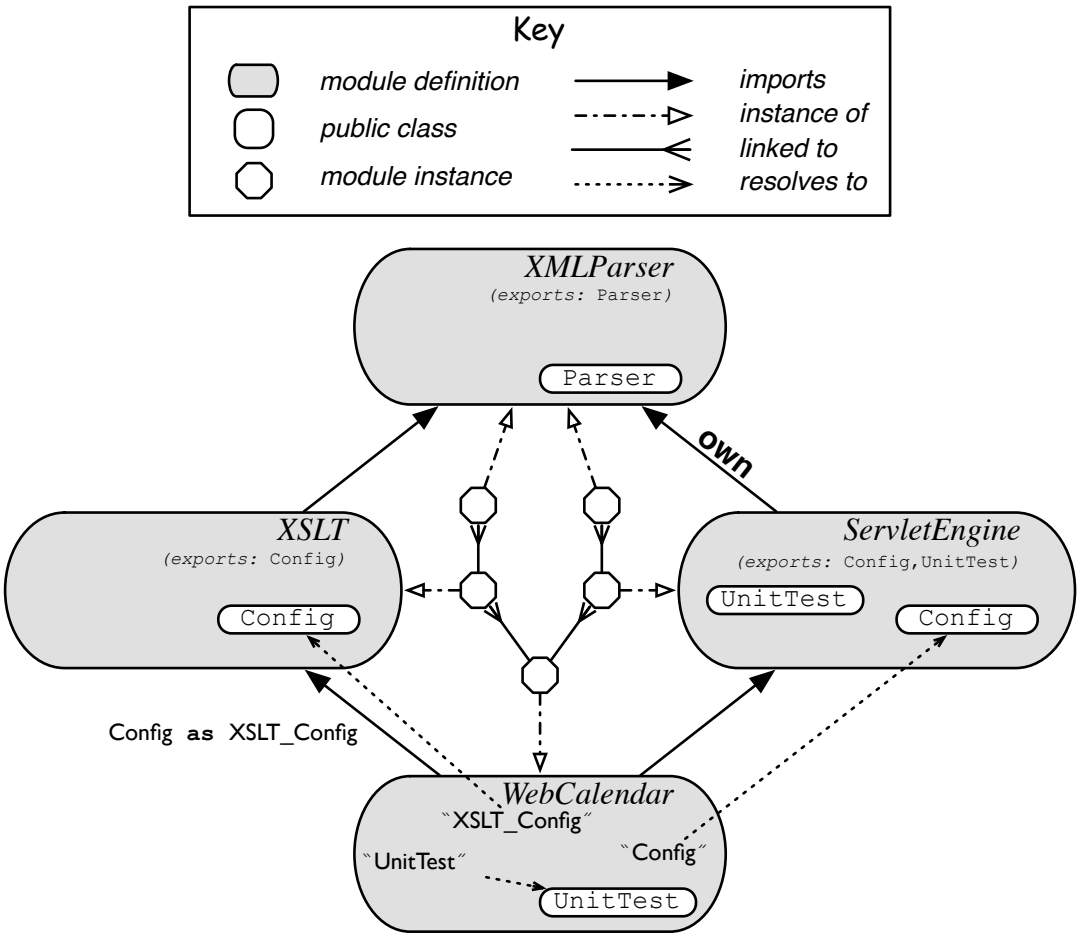


Figure 7.4: The example program (modified)

If we run the same program in compatibility mode, i.e. according to LJAM's semantics, the program throws a `NoClassDefFoundError` exception, since it cannot locate `XSLT_Config`. If we remove references to `Config` classes, then the output

shows how unintuitive and undesired LJAM's class resolution can be, i.e. *WebCalendar*'s `UnitTest` is ignored even though the class reference appears in the same module:

```
ServletEngine::UnitTest complete.
```

7.6 A limitation

When a Java Virtual Machine (JVM) encounters a class reference, e.g. "`XSLT_Config`", it calls the `loadClass` method of the appropriate classloader, as described in the previous section. However, once a `Class` object is returned by the method, JVM performs a series of runtime checks on it for consistency and security purposes.

Among these, JVM also checks that the obtained class has the name that was originally requested. This poses a problem for our boundary renaming, e.g. the class returned is named "`Config`", not "`XSLT_Config`" as requested. Our implementation uses the Byte Code Engineering Library (BCEL) [42] to satisfy this check in an ad-hoc manner. That is, we rename the class just before passing it to JVM.

The problem is that JVM stores a *distinct* `Class` object for every different naming of a single logical class. This implies that sharing over renamed classes is currently not supported. This is essentially because it is not possible to map two distinct class names to the same `Class` object; however, we believe that only a small change to the JVM is required to allow this.

Note that such a change to the JVM will break any program that relies on the invariant that "two distinct class names must point to two distinct classes." However, programmers tend to use the Java's **instanceof** operator, which checks if a certain object is compatible with a certain type, or the `Class`'s method `isAssignableFrom`, which checks two types for subtyping. Both the operator and the method could easily be made compatible with our class renaming operations.

7.7 Conclusion

In this section, we have presented our proof-of-concept implementation of both LJAM and iJAM. The implementation can closely follow the semantics of both formalisations. Note that we do not implement features not formalised by the semantics — see the start of Chapter 4 for the list of formalised features.

Consisting of only about 1200 lines of code, our efforts prove that the core concepts of the module system (and our improvements of it) can be implemented relatively easily.

8

Case study — Thorn

Scripting languages are highly popular due to their support for rapid and exploratory development. They typically have lightweight syntax, weak data privacy, dynamic typing, powerful aggregate data types, and allow execution of the completed parts of incomplete programs. The price of these features comes later in the software life cycle, since scripts are hard to evolve and compose, and are often slow. An additional weakness of most scripting languages is their lack of support for concurrency, which is increasingly required for scalability on parallel architectures, for handling concurrent real-world events, and for interacting with remote distributed services.

Thorn is a novel, object-oriented, general purpose programming language targeting the JVM. It has a careful selection of features that support the evolution of scripts into industrial grade programs. For example, it has an expressive module system that supports evolution and scalability, an optional type annotation facility for declarations, and support for concurrency based on message passing between lightweight, isolated processes. On the implementation side, Thorn has been designed to accommodate the evolution of the language itself through a compiler plugin mechanism.

An example Thorn program, a solution to the Dining Philosophers problem, is shown in Fig. 8.1. In the example, forks and philosophers are represented as *components*, analogous to threads or processes, which communicate by sending messages back and forth, and do not share any state. Demonstrated features include asynchronous message passing, pattern matching, and list comprehension. More information about the example and the language itself can be found elsewhere [8].

```
component Fork(n) {
  var holder := null;
  var waiting := null;
  fun taken(phil) {holder := phil; phil <<< "taken";}
  body {
    while (true) {
      receive {
         "die" => {break;}
       | {: take:_ :} from phil => {
           if (holder == null) taken(phil);
           else waiting := phil;
         }
       | "drop" from phil => {
           phil <<< "dropped";
           if (waiting != null) {
             taken(waiting);
             waiting := null;
           } else holder := null;
  } } } }
}Fork;

forks = %[ spawn Fork(i) | for i <- 0 .. 2];

component Phil(name, ln, rn, iter) {
  body {
    left = forks(ln);
    right = forks(rn);
    for(i <- 1 .. iter) {
                  # THINK: I think, therefore I am.
       left  <<< {: take:name :}; receive{"taken" => {}};
       right <<< {: take:name :}; receive{"taken" => {}};
                  # CRITICAL: I eat, therefore I am fed.
       right <<< "drop"; receive{"dropped" => {}};
       left  <<< "drop"; receive{"dropped" => {}};
  } }
}Phil;

phils = [
  spawn Phil("Kant", 0, 1, 10),
  spawn Phil("Hume", 1, 2, 12),
  spawn Phil("Marx", 0, 2, 8)];
  # A 3-way philosophical dinner now ensues.
```

Figure 8.1: Dining Philosophers in Thorn

This chapter focuses on Thorn's module system only. For the purposes of this chapter, one can think of Thorn as a Java-like language with no packages or backward compatibility constraints, and with a goal of allowing quick prototyping. Some of the other language details are explained when relevant.

Our high-level goals for Thorn's module system were to make its semantics intuitive and expressive, and its modules reusable and robust against change — one of the main

priorities was to localise the influence of a single module (discussed in §1.2). As in the JMS, we have module definitions and module instances. Modules contain member Thorn entities, import other modules, and control visibility of their entities to client modules.

We also make the module system non-intrusive (appropriate for scripting), and give developers an iJAM-like control over data sharing and separation. Finally, we explore some extra features: per-import overriding of the repository, module-prefixed references, name aliasing, value modules, module-level generics, and inclusion policies for imports.

We design the syntax, and give the informal semantics for Thorn's module system. We also list the conditions required for avoiding name ambiguity and for namespace localisation, find the effects of module-prefixed type references on class resolution, and explore the interaction between multiple module instances and value (un-)marshalling.

8.1 Non-intrusiveness

For Thorn, it is important that the module system does not get in the way of rapid prototyping. We made the following design decisions to achieve this goal.

Thorn source files, by default, belong to the same, unnamed module definition, i.e. a module definition that cannot be imported. Consequently, a source file does not represent a semantic boundary. When an application grows, language entities (excluding modules) can be enclosed within the module construct, mc.

mc	::=		module construct
	|	va **module** mn { $\overline{imp}$ $\overline{mem}$ $\overline{def}$ }	def.

The definitions within a module, $\overline{def}$, can refer to definitions outside of the module only through explicit module imports, $\overline{imp}$, or through file inclusions, $\overline{mem}$ — a module can include top-level language definitions (excluding modules) defined in the same or in other files, by explicitly listing file names by URI. With this, a single source file can be referred to within multiple module files. The value annotation, va, is explained in §8.3.

mem	::=		membership declaration in a module file
	|	**member** URI ;	include file at URI

A module can be compiled just-in-time (before being executed), or it can be compiled normally, which places the packaged bytecode into the default, filesystem-based repository, making it available for importing. The location of an imported module can also be overridden by specifying its URL as an annotation on an import statement (§8.6).

m_loc	::=		module location override
	|		default
	|	**from** URL	load from URL

8.2 Namespace control & robustness

Through its source files, a module defines various top-level entities, such as classes, fields, and methods. In Thorn, we refer to their names with meta-variables cn, vn, and $meth$, respectively. To refer to any of them, we use the meta-variable id.

id ::=		non-fully-qualified name
	cn	class name
|	vn	variable name
|	$meth$	method name

Since id does not specify a module name, it cannot (in general) uniquely identify an entity in the system — in Thorn's world, this makes it a non-fully-qualified reference. We can prefix id with mn, a meta-variable we use for module names, to obtain a fully-qualified reference, i.e. '$mn.id$'. We use the meta-variable $name$ for both non-fully-qualified and fully-qualified references.

$name$ ::=		possibly fully-qualified name
	id	simple identifier
|	$mn.id$	fully qualified name

If a name is not fully-qualified, i.e. it is not prefixed with a module name, the name is resolved first within the members the module, then within the imported modules. This precedence ordering makes the code of a module more robust to changes outside of it (explained in Chapter 5). For example, if a non-fully-qualified name already resolves to a local entity, then no changes within the imported modules can affect this.

Exported namespaces of imported modules are allowed to overlap. As with existing Java packages, it is only an error if an ambiguous name is actually used. This approach makes non-fully-qualified references that resolve to local entities robust against changes outside the module. However, a non-fully-qualified reference that resolved to a name exported by an imported module can become ambiguous if a different imported module starts exporting that name, too — the only way to protect against such breakage is to use fully-qualified names (or their aliases) for names that resolve within imports.

We allow aliasing of (i) the names of imported modules ($\mathbf{alias}\, mn = mn'\,;$), and (ii) the visible names of entities ($\mathbf{alias}\, id = name\,;$). Note that the alias, id, must be non-fully-qualified; the declared id becomes part of the local namespace, i.e. it must not clash with the name of any other member or alias.

ali ::=		name alias
|	$\mathbf{alias}\, mn = mn'\,;$	module name alias
|	$\mathbf{alias}\, id = name\,;$	entity name alias

To promote rapid prototyping, the whole local namespace (except for the aliases) is exported by default; however, no imported name is re-exported by default — this localises the influence of a single module, which is essential for scalability. A name is not exported if it is declared **private**, while a name is (re-)exported by declaring it **public**.

vis ::=		visibility declaration
|	*id* : **private**;	do not export *id*
|	*name* : **public**;	(re-)export *name*

To guarantee that a fully-qualified name is never ambiguous, (i) the exported names must not clash, and (ii) the names of imported modules must not clash. To allow developers to import same-named modules without breaking condition (ii), we allow module alias annotations, '**as** *mn*', on import statements. For example, if module mn' is imported '**as** *mn*', then all references to mn' must use *mn* (mn' is not bound in the local scope).

m_ali ::=		module name aliasing
|		no aliasing
|	**as** *mn*	alias as *mn*

To further control the locality of a module, we prevent the name of any module from propagating within clients of its clients through re-export statements. We achieve this as follows: if *mn* re-exports '$mn'.id$', then the fully-qualified name for that *id* within the clients of *mn* (if they do not use module aliasing) is '$mn.id$', not '$mn'.id$'. To prevent even *id* from propagating, one can use the combination of entity aliasing and re-exporting.

8.3 Sharing vs. isolation

A Thorn *component* is logically an independent unit of computation. Components have no common mutable state, and communicate with each other through message queues. The concept was introduced for easier parallelisation and process mobility.

As in iJAM, we can have multiple module instances of a single module definition in a single runtime environment. There can be a single shared module instance for each component, and as many non-shared module instances as required. We can place an optional replication parameter, **own**, on an import statement, which creates a non-shared module instance; otherwise, a component-wide instance is obtained. That is, the semantics of the replication parameter is similar to that of iJAM.

rep ::=		replication parameter
|		shared
|	**own**	own

Two non-**own** import statements referring to the same module name (with the same import constraints) always refer to the same module instance. Two **own** import statements (even with the same import constraints) always refer to different module instances.

Modules do not have any explicit initialisation code. However, they can define top-level variables of which initial values refer to functions — these are the implicit initialisation functions of that module. This is similar to the approach taken for ML modules [30].

Imported modules must be initialised before the module itself. They are initialised according to the order of import statements. Any reference to a yet non-initialised module during execution throws a runtime exception. Some of the offending references can be found at compile-time through a control-flow analysis.

Developers can annotate certain modules as *value modules*. Then, the compiler will report an error if the module does not define only non-mutable fields with statically-computable values, or if the imported modules are not value modules themselves. Since Thorn classes cannot define their own static state, it is safe to share a single instance of a value module among all components; therefore, value modules are *always* shared.

va	::=		value annotation
	|		a standard module
	|	**value**	a value module

8.4 Module-level generics

A Thorn module can also have *generic parameters*, which are name arguments for explicitly specified name parameters of a module. These names can refer to any entity, i.e. a class, a top-level variable, or a top-level function.

The generic names of a module must be declared with a require clause. Such a clause declares a name, but provides no binding. Note that this name becomes a part of the local namespace, but is not exported by default.

req	::=		require clause
	|	**require** $name$;	$name$ declared, but not bound

Binding for the names declared through the require clause is specified when a generic module is imported. That is, the import statement can explicitly provide bindings for all of the imported module's parameters. As a matter of convenience, if binding for a name is not specified explicitly, then Thorn tries to resolve the name within the client module.

ga	::=		module-level, generic arguments
	|		no arguments
	|	$(\overline{id_k = name_k}^{\,k})$	arbitrary number of arguments

The following example shows how the module *Concrete* specifies GraphNode as the binding for module *Abstract*'s Node:

```
// ======= module Abstract =======
require Node;
// ======= module Concrete =======
import Abstract(Node=GraphNode);
```

Although the generics of this sort provides more opportunities for reuse, one needs to take care not to introduce cycles in module initialisation. At the moment, we take a conservative approach for value modules, i.e. a value module cannot be parametric.

8.5 Module archives

A module definition is defined as a module construct, mc, together with the top-level entities of the included files. The compiled code of a module definition is stored in a JAR-like archive, and put into a repository, which makes it available for importing.

By default, no imported module is included in the archive. However, we allow developers to annotate import statements with the **include** keyword. This tells the compiler to include the corresponding module definition into the same archive file.

inc ::=		optional include of import
|		do not include
|	**include**	include import

Such *included modules* are also referred to as *inner modules* or *member modules*. To further control the influence of a single module, the inclusion does not work recursively, i.e. modules that an included module imports are not included automatically.

The only change to the runtime semantics of an included module is that it cannot be directly imported. Every use of an included module must go through its owner. This provides a stronger form of component-level information hiding.

In § 5.4, we discussed JMS's weak information hiding, and iJAM's instance-based information hiding. In the example we used, a module, *M*, imported another, *N*, which in turn imported a third, *O*. We found that, even in iJAM, *N* could not encapsulate *O*'s exported names from *M*, since *M* was free to import an instance of *O* directly. In Thorn, *N* can now *include O*, preventing *M* from directly importing *O*. This provides a better form of information hiding, which we refer to as *strong information hiding*.

8.6 Overview of the high-level syntax

In this section, we combine the concepts and annotations introduced thus far by presenting the syntax of an import statement, imp, and a source file, $file$.

An import statement, imp, can specify an inclusion policy, inc (§ 8.5), replication policy, rep (§8.3), generic arguments, ga (§8.4), module name aliasing, m_ali (§8.2), and the module's repository override, m_loc (§8.1).

imp	::=		import statement
	|	$inc\ \mathbf{import}\ rep\ mn\ ga\ m\_ali\ m\_loc\,;$	def.

A source file, $file$, can specify import statements, $\overline{imp}$, definitions, $\overline{def}$, and module constructs, $\overline{mc}$. Definitions include aliases, $\overline{ali}$, visibility declarations, $\overline{vis}$, and entities, $\overline{entity}$. A component, $component$, is classified as an entity. A module construct, mc, can be prefixed with the value annotation, va, and can contain import statements, $\overline{imp}$, file inclusions, $\overline{mem}$, and definitions, $\overline{def}$.

$file$	::=		source file
	|	$\overline{imp}\ \overline{def\_or\_mc}$	def.
def_or_mc	::=		definition or module construct
	|	def	definition
	|	mc	module construct
def	::=		definition
	|	ali	alias
	|	vis	visibility declaration
	|	$entity$	entity
$entity$	::=		entity definition
	|	fd	field declaration
	|	$meth\_def$	method definition
	|	cld	class definition
	|	$component$	component
	|	...	(other entities)
mc	::=		module construct
	|	$va\ \mathbf{module}\ mn\ \{\ \overline{imp}\ \overline{mem}\ \overline{def}\ \}$	def.

8.7 Components and (de-)serialisation

Developers refer to entities with *user names*, which are the (possibly fully-qualified) names in scope. For serialisation, it is important to be able to uniquely identify a type across different runtimes. In this thesis, we assume the presence of an operation `uniq`,

which returns a *unique name* for any given user name of any definition.[1] When we serialise a value, we attach to the message the unique name corresponding to the value's type.

Now, suppose that (i) a module, `M`, defines a class, `List`, (ii) modules `A` and `B` both import their own instance of the module `M`, (iii) `A` and `B` re-export `List` under names `AList` and `BList`, respectively, (iv) `C` imports both `A` and `B`, and (v) `C` receives a message:

```
// ======= M.thm =======
member List.th;                 // Class List is a member of module M.
// ======= A.thm =======
import own M;                   // A imports its own M.

alias AList = M.List;
AList: public;                  // List is re-exported as AList.
// ======= B.thm =======
import own M;                   // B imports its own M.

alias BList = M.List;
BList: public;                  // List is re-exported as BList.
// ======= C.thm =======
import A;                       // C imports A.
import B;                       // C imports B.
val msg = receive();            // C receives a message.
```

Suppose that the unique name received at the de-serialisation point (last line above) has value u. The runtime will compare u to the unique names of types visible in the current context. If a single match exists, the received value will assume the type corresponding to that unique name. Otherwise, the runtime will throw an exception.

If u is equal to `uniq(A.AList)`, then u is also equal to `uniq(B.BList)`, since `A.AList` and `B.BList` point to the same class definition. However, the types corresponding to `A.AList` and `B.BList` are incompatible, since they belong to different module instances (and could so have different invariants by relying on different static state). Therefore, we have to decide whether we want to de-serialise the received value as a value of type `A.AList`, or as a value of type `B.BList`. To choose the latter, for example, we must annotate the last line as follows:

```
val msg : B.BList = receive();
```

Now, there is no runtime ambiguity. The above syntax corresponds to the syntax of the proposed gradual typing scheme for Thorn.

[1] The implementation of `uniq` is not important for this thesis. For more information on possible implementations, see work on Acute [45] and HashCaml [7].

8.8 Versions and other custom properties

The current design does not include module properties such as versions. Since the feature will likely be added in the future, this section overviews our proposal for it.

As in the Java Module System, modules definitions will have many *properties*. Versions are an example of a property. When properties are specified, modules could be looked up and instantiated according to values of their properties.

Our proposal for Thorn is that each property should be implemented as a class that implements the Java's `Comparable` interface. This would promote a total ordering among values for a particular property. An ordering is important for choosing a single result when loose property constraints are used. Constraints could be defined through *constraint classes*, which could be extensions of a class like the one below.

```
abstract class ModuleConstraint<P> {
  public Collection<P> filter(Collection<P> ps);
}
```

The `filter` method would be used to filter a collection of values for a property, i.e. to narrow down the search for an appropriate module to import.

Versions would then be an instance of the above module property. For example, one class would define the version property, `Version`, while another would implement the version constraint, `VersionConstraint`. Java code for the latter would be:

```
class VersionConstraint extends ModuleConstraint<Version> {
  ... public Collection<Version> filter(Collection<Version> vs) {...} ...
}
```

Versions could then be used as follows:

```
module MyModule; Version("3.1");
import MyOtherModule : VersionConstraint("5.2+");
```

8.9 Conclusion

In this chapter, we have sketched our design for Thorn's module system. We found useful combinations of features applicable to other module systems, and discovered a potential hazard that comes with multiple module instances.

Thorn aims to support rapid prototyping. To that goal, we designed a module system to be non-intrusive. We have (a) automatic, unnamed module definitions, (b) a small difference between source files and module constructs, (c) just-in-time compilation that allows easy executions, (d) optional overrides of imported modules' locations to avoid

repositories altogether, and (e) implicit application of generic arguments. These do not depend on any specific Thorn features, and could fairly easily be applied to JMS and iJAM, with the exception of (b). Furthermore, a module construct can be moved to a different source file without changing its semantics.

We use iJAM's resolution algorithm for non-fully-qualified names, and allow the user to start the same algorithm in any of the direct imports, instead, through the use of module-prefixed type references. By using a few simple rules for namespace control, and by promoting the use of module-local names, or names from direct imports only, we obtain the following important properties: (i) no propagation of module names; (ii) the ability to prevent propagation of entity names; (iii) a strong form of robustness against interface evolution of the imported modules; and, (iv) a guarantee that every visible entity can be accessed, and that a fully-qualified name is never ambiguous. Points (i-ii) enable strong namespace localisation, which, together with points (iii-iv), make the module system highly scalable. Such namespace resolution can be applied to any language with support for module-prefixed type references and module/entity aliasing, but is especially useful for languages of which module visibility is not transitive (§1.1.5), and where non-fully-qualified names can refer to entities in the imported modules.

In our module system for Thorn, a client module can include an imported module into the same distribution package. With this feature in place, we can prevent all other modules from importing the included module. As explained in §8.5, this leads to a strong form of information hiding. This feature can easily be added to both JMS and iJAM.

When going through de-serialisation examples in Thorn, we noticed a problem that can appear in any language that supports multiple module instances, including iJAM: in the presence of multiple instances, a de-serialised value can match multiple types — even though these types have the same definition, they are distinct, since they belong to different module instances. We found that such ambiguities can be resolved through the use of module-prefixed type annotations at de-serialisation points (§8.7).

To make it easier to implement distributed systems, Thorn defines components as logically independent units of computation. Since components do not share state, there can be a "shared" module instance for every component. To improve performance, we defined value modules (modules with final state) that are shared even across components.

9 Conclusion

In this thesis, we formalised a core of the Java Module System (JMS), found two key deficiencies with it, and proposed and formalised our solutions for them. In the process, we explored the details of better component-level information hiding, module-boundary renaming, and module instance sharing between multiple applications. Finally, we analysed these properties within a Java-like language where module-prefixed type references are permitted, and investigated marshalling in the context of multiple module instances.

We found that, apart from being unintuitive, JMS is not expressive enough to allow its users to disambiguate *any* non-fully-qualified type name to *any* of the visible and applicable class definitions without modifying any of the imported modules. We show that this problem arises from the module system's lack of module-boundary renaming of types (Chapter 5), or module-prefixed type references (Chapter 8) — we also explained that selective importing is too weak to solve this problem. In both solutions, it is critical to ensure distinctness of type names of both members and exports of module definitions. We also observe that module-boundary renaming, combined with the reversal of the class resolution order, arguably makes a substantial improvement to the robustness of module definitions to external changes, which is critical for scalability of a module system.

As mentioned in the introduction, one the the key properties of a module system is its localisation of influence for each module definition and instance. Above, we outline how we improve information hiding while ensuring essential expressivity. This thesis also shows that with a few annotations on the import statements, we allow the developer to control type and data sharing of modules' imports; having multiple instances of a single

module definition, the users can avoid conflicting invariants on a common import, as well as contention due to parallel use of a single runtime structure. We localise the effect of these annotations to direct imports only, which promotes the desired locality, but also implies that the developer must trust the annotations set within the imports.

We discovered that the JMS provides only a weak form of information hiding, since any module instance is free to access any other. iJAM improves on this, since it allows encapsulation of static data through own module instances. Thorn enables a strong form of information hiding through inclusion of imported modules in the same distribution package, which prevents other modules from importing it directly. An overview of what module system has which property (§1.2) is shown in Fig. 9.1 (page 155) — it includes LJAM, iJAM, and Thorn; the properties listed are a subset of those in Fig. 1.5 (page 32).

We also observed that in the presence of multiple module instances, ambiguity can arise during de-serialisation, since a serialised value can match multiple types at the receiving end. Although it would be easy to default all de-serialisation to a type of a specific module instance, e.g. the first one syntactically declared, we discovered a way for developers to choose among the available module instances on a per-type basis.

The above paragraphs have already confirmed the first part of our 'thesis' (§ 1.3), namely that the lack of explicit module interfaces together with the parent-then-self name resolution leads to poor support for localisation of a module's influence and for code reuse, while the absence of module-prefixed references results in fragile and inexpressive semantics at the user level. Now, we evaluate the second part of the 'thesis,' which states that a rigorous formalisation of a programming language (and its add-ons):

- gives a valuable insight into the details of the semantics that can find illusive design problems early (before release);

  > There are many subtle issues of the semantics that we would not have noticed had we not done the rigorous formalisations and their proofs of soundness. For example, JMS's class resolution first searches the core libraries to prevent the system classes from getting overridden. When combined with module-boundary renaming, we lose the property of 'target context equivalence for class lookups' (Lemma 21, page 129), a property that should intuitively hold. The falsity of this property was noticed only once we failed to prove type soundness in Isabelle/HOL. The property was re-established by enforcing a constraint, which prevents renaming a class from a name already exported by the core library module.

- promotes a precise discussion of the definition;

  > This is evident throughout this document, but most obvious from the example figures discussing the problems of LJAM (Chapter 5).

- allows important properties to be proven about the language;

  > We mechanically prove type soundness for all our formalisations — this tells us that the programs in these languages will "not go wrong," i.e. they will never result in a malformed program state. However, it does not entail that the semantics of these formalisations are what we want them to be, or that they are useful in practice. We propose more properties that can be proven on our formalisations as part of future work.

- is cost-effective, and can be done on the same timescale as the industrial design and standardisation process.

  > The analysis of the draft documents, the design and the formalisations of our languages, their evaluation, and the documentation took in total about two man-years of work. From this, we can conclude that a large software company could produce a formalisation for a complete definition of a general-purpose programming language within years, possibly including mechanical proofs of a few important properties. Making such a definition freely available, universities and research departments from all over the world could contribute to it, and its proofs of properties.

The thesis (§1.3) also states that it is possible to modularise language definitions and their proofs scripts into *language modules*. Although our languages are not structured from truly independent modules, the potential for this is clear — we have achieved an astonishing amount of language definition and proof script reuse (§6.6), where most of the proof scripts "belong" to some part of the language definition.

The formalisation tools were key to this work. Ott found many consistency errors with the definitions automatically. This gave more courage to experiment with alternative definitions, and allowed compiler-error based regression testing. Ott also allows source-level merging of language definition parts, which we make heavy use of — it is critical here that Ott can detect dependencies among terms (even across different Ott files), and can then correctly order them in the theorem prover (currently not also in the LaTeX) output — at the term level, however, Ott could use a few more features for definition reuse, such as generic meta-terms. Empty productions are permitted, which allows Ott source files to be closer to what one would normally write; however, if a possibly-empty term is added to a production (of another term), then Ott will silently accept existing productions of this form as special cases (where the possibly-empty term is empty). Although such consistency problems can be prevented with clever use of the tool, they are almost definitely detected when proving general properties within the theorem prover.

By mechanising the meta-theory in Isabelle/HOL, we found it easy to identify incorrectly formulated judgements and incomplete relations. Since Ott generates the Isabelle/HOL definition of the language, we can check at any point whether the language still satisfies the properties we have already proven, automatically. We observed that it is wise to change the definition in the smallest possible steps that are expected to preserve the properties — this way, proof regression is often trivial. We also learnt that the termination proof of a function should try to be independent of any property that can change (or become invalid) through any process in the system. Finally, Isabelle/HOL made it clear, time and time again, that "a rule with exceptions is incomplete."

So far, the discussion with the developers of JMS regarding our work on iJAM has been limited. The idea about module-boundary renaming and the reversal of class resolution was dismissed for the time being, since the possibility of more `ClassCastExceptions` being thrown (an issue that requires substantial test cases) has so far detracted from the benefits gained. The feedback to multiple instances was more positive; an iJAM-like approach was said to be included in a future version of JMS. Hopefully, this thesis will have more impact: promoting better module systems for modern programming languages.

We conclude with a list of options for future work:

- formalise more features of JMS, such as versions, version constraints, and custom import code (when executing such code, many definitions are not yet available, while cyclic dependencies can arise easily, which makes it interesting to find properties that the code must satisfy for the language to remain type sound);
- prove more properties about the formalisations, for example that (1) accessibility policies are guaranteed, and (2) with boundary renaming, any non-fully-qualified type name can be disambiguated to refer to any visible alternative;
- extend LJ with support for type casts and static data, and formally prove properties regarding class-cast exceptions and module instance invariants, respectively;
- investigate the benefits of a form of shallow validation for iJAM (§5.3);
- determine exactly what changes to the Java Virtual Machine are required to allow class name aliasing (§7.6);
- extract reference implementations from LJAM and iJAM, and compare their execution traces to those of our reference implementation, as well as implementations of JMS, on large examples;
- Lightweight Java (LJ) has already been used by others [14] — due to this success story, we think it is worth investing some effort to simplify its definition further; and
- explore the theory of *module systems for programming language definitions*, i.e. a module system that supports separate compilation, and where modules are reusable parts of a language definition that also contain the corresponding parts of the proof scripts; design such a module system for Ott.

PROPERTY	JAVA CLASSES	JAVA PACKAGES	JMS MODULES	OSGI BUNDLES	.NET ASSEMBLIES	OCAML MODULES	JIAZZI UNITS	LJAM MODULES	IJAM MODULES	THORN MODULES
	MODULE(-LIKE) FEATURE									
parametricity	✓	✗	✗	✗	✗	✓	✓	✗	✗	✓
module-prefixed type references	✓	✓	✗	✗	✗	✓	✗	✗	✗	✓
module instantiation policies	✓	✗	✗	✗	✗	✗	✗	✗	✓	✓
multiple files per module	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓
non-transitive module visibility	✗	✗	✗	✗	✗	✓	✗	✗	✓	✓
optional re-exporting	✗	✗	✓	✓	✗	✓	✓	✓	✓	✓
renaming of exports by exporters	✗	✗	✗	✗	✗	✓	✗	✗	✗	✓
renaming of exports by importers	✗	✗	✗	✗	✓	✓	✗	✗	✓	✓
renaming of imports	✗	✗	✗	✗	✓	✓	✗	✗	✗	✓
runtime sharing among programs	✗	✗	✓	✓	✗	✗	✗	✓	✓	✓
selective exporting	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
sub-modules	✓	✗	✗	✗	✗	✓	✓	✗	✗	✓
sub-modules defined in own files	✗	✗	✗	✗	✗	✓	✓	✗	✗	✓
versioning	✗	✗	✓	✓	✓	✗	✗	✗	✗	✗

Figure 9.1: Overview of properties/features for our module systems in the context of a few related module systems. *Please note that (i) iJAM has non-transitive module visibility only for its module instances, while Thorn also has it for its module definitions, and (ii) Thorn supports runtime sharing among programs only for value modules.*

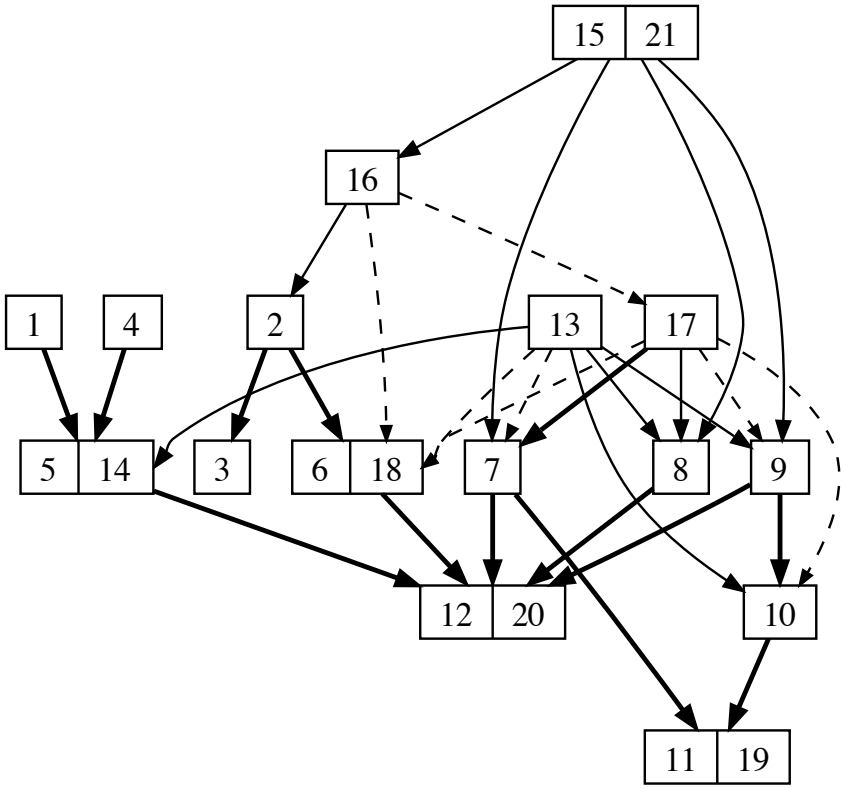
A

Dependency among lemmas and theorems

The following figure shows the dependency among the lemmas and theorems mentioned in the thesis. Numbers in the same box represent equivalent lemmas (but with different proofs) from different chapters, e.g. 5 and 14 correspond to proofs of type reflexivity in LJ (Chapter 3) and LJAM (Chapter 4), respectively.

The lines are thick if they are used within all formalisms, thin if they are used for LJAM and iJAM only, and dashed if they are used for LJAM only. No other permutation appears in our proofs.

The fact that we get LJAM-only dependencies, and not iJAM-only dependencies, might seem strange, since iJAM is an extension of LJAM. This is due to extensive simplification of iJAM proofs; it seems likely that the same simplification can be achieved for LJAM proofs, too. Therefore, the dashed lines are probably not essential.


15
21
16
1
4
2
13
17
5
14
3
6
18
7
8
9
12
20
10
11
19

B

LJ's proof of progress in Isabelle/HOL

theorem progress:

$[\![(\Gamma,$ config-normal P L H S$) \in$ wf-config; S $\neq []]\!] \Longrightarrow \exists$ config$'$. (config-normal P L H S, config$'$) $\in$ r-stmt

apply(case-tac S)

apply(simp)

apply(clarsimp) **apply**(rename-tac s ss)

apply(case-tac s)

apply(erule-tac[1−7] wf-configE) **apply**(simp-all)

```
--- block
```
apply(force intro: r-blockI[simplified])
```
--- variable assignment
```
apply(clarsimp, erule wf-stmtE, simp-all, clarsimp)
apply(frule type-to-val, simp) **apply**(clarify) **apply**(frule r-var-assignI) **apply**(force)
```
--- field read
```
apply(clarsimp, erule wf-stmtE, simp-all, clarsimp)
apply(erule wf-varstateE) **apply**(drule-tac x = xa **in** bspec, simp add: domI)
apply(erule wf-objectE) **apply**(clarsimp) **apply**(frule r-field-read-npeI) **apply**(force)
apply(clarsimp) **apply**(erule-tac ?a3.0 = Some ty **in** sty-option.cases) **apply**(clarsimp split: option.splits)
apply(erule wf-heapE) **apply**(drule-tac x = oid **in** bspec, simp add: domI) **apply**(clarsimp)
apply(rename-tac x oid ty-x-s ty-x-d fields-oid fs)
apply(frule no-field-hiding, simp+) **apply**(drule-tac x = f **in** bspec, simp) **apply**(clarsimp)
apply(erule wf-objectE) **apply**(clarsimp, frule-tac H = H **in** r-field-readI, simp, force)+
```
--- field write
```
apply(clarsimp, erule wf-stmtE, simp-all, clarsimp)
apply(erule wf-varstateE) **apply**(frule-tac x = x **in** bspec, simp add: domI)
apply(erule wf-objectE) **apply**(clarsimp) **apply**(frule r-field-write-npeI) **apply**(force)
apply(clarsimp) **apply**(erule sty-option.cases) **apply**(clarsimp) **apply**(rename-tac ty-y ty-f)
apply(drule-tac x = y **in** bspec, simp add: domI) **apply**(clarsimp)
apply(erule sty-option.cases) **apply**(clarsimp split: option.splits)
apply(erule wf-objectE) **apply**(clarsimp)
apply(frule-tac H = H **and** y = y **in** r-field-writeI, simp, force)+

--- conditional branch

apply(clarsimp, erule wf-stmtE, simp-all, clarsimp) **apply**(erule disjE)

apply(frule type-to-val, simp, clarify) **apply**(case-tac v = w)

apply(frule-tac y = y **in** r-if-trueI, force+)

apply(frule-tac y = y **in** r-if-falseI, force+)

apply(frule type-to-val, simp, clarify) **apply**(case-tac v = w)

apply(frule-tac y = y **and** v = w **in** r-if-trueI, force+)

apply(frule-tac y = y **and** v = w **in** r-if-falseI, force+)

--- object creation

apply(clarsimp, erule wf-stmtE, simp-all, clarsimp) **apply**(rename-tac cl ctx ty var)

apply(erule sty-option.cases) **apply**(clarsimp) **apply**(rename-tac ty-cl ty-var)

apply(simp add: is-sty-one-def split: option.splits) **apply**(rename-tac path)

apply(frule find-path-fields) **apply**(erule exE)

apply(frule fresh-oid) **apply**(erule exE)

apply(frule-tac H = H **and** L = L **and** var = var **and** s-list = ss **and** f-list = fs **in** r-newI[simplified])

apply(clarsimp simp add: fields-f-def split: option.splits) **apply**(assumption) **apply**(simp) **apply**(force)

--- method call

apply(clarsimp, erule wf-stmtE, simp-all, clarsimp)

apply(rename-tac ss y-ty-list x ty-x-s m ty-r-s var)

apply(erule wf-varstateE) **apply**(frule-tac x = x **in** bspec, simp add: domI) **apply**(clarsimp)

apply(erule wf-objectE) **apply**(clarsimp) **apply**(frule r-mcall-npeI) **apply**(force)

apply(elim sty-option.cases) **apply**(clarsimp split: option.splits)

apply(rename-tac ty-r-s ty-var-s ty-x-s ty-x-d fs-x)

apply(frule mtype-to-find-md, simp+) **apply**(clarsimp)

apply(frule-tac A = dom L **and** i = length vds **in** fresh-vars) **apply**(clarsimp) **apply**(rename-tac vars′)
apply(frule exist-lifted-values) **apply**(simp) **apply**(clarify)
apply(frule-tac vars′ = vars′ **in** fresh-x-not-in-vars′) **apply**(erule exE) **apply**(erule conjE)
apply(subgoal-tac ∃ vars. vars = map (λvd. case vd of vd-def cl var ⇒ var) vds)
 apply(erule exE) **apply**(subgoal-tac length vars = length vds)
  apply(frule length-y-ty-list-vs)
  apply(subgoal-tac ∃ T. T = (map-of (zip (map (λvd. case vd of vd-def cl var ⇒ x-var var) vds) (map x-var vars′)))(x-this ↦ x′))
   apply(erule exE)
   apply(frule-tac H = H **and** P = P **and** meth = m **and** ctx = ctx **and** cl = cl-r **and** y = y **and** ty = ty-x-d **and** y-cl-var-var′-v-list =
        zip (map fst y-ty-list) (zip (map (λvd. case vd of vd-def cl var ⇒ cl) vds) (zip vars (zip vars′ vs)))
        and s′′-s′-list = zip (tr-ss-f T ss′) ss′ **and** var = var **and** s-list = ss **and** x′ = x′ **and** T = T
        in r-mcallI[simplified])
   apply(force) **apply**(simp) **apply**(simp) **apply**(simp add: vars′-eq) **apply**(simp) **apply**(assumption) **apply**(simp add: vars′-eq)
   apply(cut-tac L = L **and** y-ty-list = y-ty-list **in** lift-opts-mapping) **apply**(erule-tac x = vds **in** allE)
   apply(erule-tac x = vars **in** allE) **apply**(erule-tac x = vars′ **in** allE) **apply**(erule-tac x = vs **in** allE)
   apply(simp) **apply**(simp) **apply**(simp) **apply**(simp add: translation-eq) **apply**(simp) **apply**(force)
by force+

C

Other relational definitions

C.1 LJ lookup rules

Class (find_cld_f)

$\boxed{\textbf{find\_cld}\,(P,\ ctx,\ fqn) = ctxcld_{opt}}$ class lookup

FC_EMPTY

$$\frac{}{\textbf{find\_cld}\,([\,],\ ctx,\ fqn) = \textbf{null}}$$

FC_CONS_TRUE

$$\frac{\begin{array}{l}1.\ P = cld\ cld_2\ ..\ cld_k \\ 2.\ cld = \textbf{class}\ dcl\ \textbf{extends}\ cl\ \{\ \overline{fd}\ \overline{meth\_def}\ \}\end{array}}{\textbf{find\_cld}\,(P,\ ctx,\ dcl) = (ctx,\ cld)}$$

FC_CONS_FALSE

$$\frac{\begin{array}{l}1.\ cld = \textbf{class}\ dcl'\ \textbf{extends}\ cl\ \{\ \overline{fd}\ \overline{meth\_def}\ \} \\ 2.\ dcl\ \neq\ dcl' \\ 3.\ \textbf{find\_cld}\,(cld_2\ ..\ cld_k,\ ctx,\ dcl) = ctxcld_{opt}\end{array}}{\textbf{find\_cld}\,(cld\ cld_2\ ..\ cld_k,\ ctx,\ dcl) = ctxcld_{opt}}$$

Type (find_type_f)

$\boxed{\mathbf{find\_type}\,(P,\ ctx,\ cl) = \tau_{opt}}$ type lookup

FT_OBJ

$$\frac{}{\mathbf{find\_type}\,(P,\ ctx,\ \texttt{Object}) = \texttt{Object}}$$

FT_NULL

$$\frac{1.\ \mathbf{find\_cld}\,(P,\ ctx,\ fqn) = \mathbf{null}}{\mathbf{find\_type}\,(P,\ ctx,\ fqn) = \mathbf{null}}$$

FT_DCL

$$\frac{1.\ \mathbf{find\_cld}\,(P,\ ctx,\ dcl) = (ctx',\ cld)}{\mathbf{find\_type}\,(P,\ ctx,\ dcl) = ctx'.dcl}$$

Inheritance path – recursive part (find_path_rec_f)

$\boxed{\mathbf{find\_path\_rec}\,(P,\ ctx,\ cl,\ \overline{ctxcld}) = \overline{ctxcld}_{opt}}$ class path lookup (recursive part)

FPR_OBJ

$$\frac{}{\mathbf{find\_path\_rec}\,(P,\ ctx,\ \texttt{Object},\ \overline{ctxcld}) = \overline{ctxcld}}$$

FPR_NULL

$$\frac{1.\ (\neg\mathbf{acyclic\_clds}\,P)\ \vee\ \mathbf{find\_cld}\,(P,\ ctx,\ fqn) = \mathbf{null}}{\mathbf{find\_path\_rec}\,(P,\ ctx,\ fqn,\ \overline{ctxcld}) = \mathbf{null}}$$

FPR_FQN

$$\frac{\begin{array}{l}1.\ \mathbf{acyclic\_clds}\,P\ \wedge\ \mathbf{find\_cld}\,(P,\ ctx,\ fqn) = (ctx',\ cld)\\ 2.\ \mathbf{superclass\_name}\,(cld) = cl\\ 3.\ \mathbf{find\_path\_rec}\,(P,\ ctx',\ cl,\ \overline{ctxcld}\,@\,[(ctx',\ cld)]) = \overline{ctxcld}_{opt}\end{array}}{\mathbf{find\_path\_rec}\,(P,\ ctx,\ fqn,\ \overline{ctxcld}) = \overline{ctxcld}_{opt}}$$

Inheritance path (find_path_f)

$\boxed{\mathbf{find\_path}\,(P,\ ctx,\ cl) = \overline{ctxcld}_{opt}}$ class path lookup with a class name

FP_DEF

$$\frac{1.\ \mathbf{find\_path\_rec}\,(P,\ ctx,\ cl,\ [\,]) = \overline{ctxcld}_{opt}}{\mathbf{find\_path}\,(P,\ ctx,\ cl) = \overline{ctxcld}_{opt}}$$

Inheritance path – for type (find_path_f)

$\boxed{\mathbf{find\_path}\,(P,\,\tau) = \overline{ctxcld}_{opt}}$ class path lookup with a type

FPTY_OBJ

$$\frac{}{\mathbf{find\_path}\,(P,\,\texttt{Object}) = [\,]}$$

FPTY_DCL

$$\frac{1.\,\mathbf{find\_path}\,(P,\,ctx,\,dcl) = \overline{ctxcld}_{opt}}{\mathbf{find\_path}\,(P,\,ctx.dcl) = \overline{ctxcld}_{opt}}$$

Fields in path (fields_in_path_f)

$\boxed{\mathbf{fields\_in\_path}\,(\overline{ctxcld}) = \overline{f}}$ fields lookup in a class path

FIP_EMPTY

$$\frac{}{\mathbf{fields\_in\_path}\,([\,]) = [\,]}$$

FIP_CONS

$$\frac{\begin{array}{l}1.\,\mathbf{class\_fields}\,(cld) = \overline{cl_j\ f_j\ ;}^{\,j} \\ 2.\,\mathbf{fields\_in\_path}\,(ctxcld_2\,..\,ctxcld_k) = \overline{f} \\ 3.\,\overline{f}' = \overline{f_j}^{\,j}\ ;\ \overline{f}\end{array}}{\mathbf{fields\_in\_path}\,((ctx,\ cld)\ ctxcld_2\,..\,ctxcld_k) = \overline{f}'}$$

Fields (fields_f)

$\boxed{\mathbf{fields}\,(P,\,\tau) = \overline{f}_{\,opt}}$ fields lookup in type τ

FIELDS_NONE

$$\frac{1.\,\mathbf{find\_path}\,(P,\,\tau) = \mathbf{null}}{\mathbf{fields}\,(P,\,\tau) = \mathbf{null}}$$

FIELDS_SOME

$$\frac{\begin{array}{l}1.\,\mathbf{find\_path}\,(P,\,\tau) = \overline{ctxcld} \\ 2.\,\mathbf{fields\_in\_path}\,(\overline{ctxcld}) = \overline{f}\end{array}}{\mathbf{fields}\,(P,\,\tau) = \overline{f}}$$

Methods in path (methods_in_path_f)

$\boxed{\mathbf{methods\_in\_path}\,(\overline{cld}) = \overline{meth}}$ method names lookup in a path

MIP_EMPTY

$$\frac{}{\mathbf{methods\_in\_path}\,([\,]) = [\,]}$$

MIP_CONS

$$\frac{\begin{array}{l}1.\,\mathbf{class\_methods}\,(cld) = \overline{meth\_def_l}^{\,l} \\ 2.\,\overline{meth\_def_l = cl_l\ meth_l\ (\overline{vd}_l)\ \{\ meth\_body_l\ \}}^{\,l} \\ 3.\,\mathbf{methods\_in\_path}\,(cld_2\,..\,cld_k) = \overline{meth}' \\ 4.\,\overline{meth} = \overline{meth_l}^{\,l}\ ;\ \overline{meth}'\end{array}}{\mathbf{methods\_in\_path}\,(cld\ cld_2\,..\,cld_k) = \overline{meth}}$$

Methods (methods_f)

$\boxed{\mathbf{methods}\,(P,\,\tau) = \overline{meth}}$ method names lookup in a type

METHODS_METHODS

$$\frac{1.\ \mathbf{find\_path}\,(P,\,\tau) = \overline{(ctx_k,\,cld_k)}^{\,k} \quad 2.\ \mathbf{methods\_in\_path}\,(\overline{cld_k}^{\,k}) = \overline{meth}}{\mathbf{methods}\,(P,\,\tau) = \overline{meth}}$$

Field type in fields (ftype_in_fds_f)

$\boxed{\mathbf{ftype\_in\_fds}\,(P,\,ctx,\,\overline{fd},\,f) = \tau^{\perp}_{opt}}$ field type lookup in a list

FTIF_EMPTY

$$\frac{}{\mathbf{ftype\_in\_fds}\,(P,\,ctx,\,[\,],\,f) = \mathbf{null}}$$

FTIF_CONS_BOT

$$\frac{1.\ \mathbf{find\_type}\,(P,\,ctx,\,cl) = \mathbf{null}}{\mathbf{ftype\_in\_fds}\,(P,\,ctx,\,cl\,f\,;\,fd_2\,..\,fd_k,\,f) = \perp}$$

FTIF_CONS_TRUE

$$\frac{1.\ \mathbf{find\_type}\,(P,\,ctx,\,cl) = \tau}{\mathbf{ftype\_in\_fds}\,(P,\,ctx,\,cl\,f\,;\,fd_2\,..\,fd_k,\,f) = \tau}$$

FTIF_CONS_FALSE

$$\frac{1.\ f \neq f' \quad 2.\ \mathbf{ftype\_in\_fds}\,(P,\,ctx,\,fd_2\,..\,fd_k,\,f') = \tau^{\perp}_{opt}}{\mathbf{ftype\_in\_fds}\,(P,\,ctx,\,cl\,f\,;\,fd_2\,..\,fd_k,\,f') = \tau^{\perp}_{opt}}$$

Field type in path (ftype_in_path_f)

$\boxed{\mathbf{ftype\_in\_path}\,(P,\,\overline{ctxcld},\,f) = \tau_{opt}}$ field type lookup in a path

FTIP_EMPTY

$$\frac{}{\mathbf{ftype\_in\_path}\,(P,\,[\,],\,f) = \mathbf{null}}$$

FTIP_CONS_BOT

$$\frac{1.\ \mathbf{class\_fields}\,(cld) = \overline{fd} \quad 2.\ \mathbf{ftype\_in\_fds}\,(P,\,ctx,\,\overline{fd},\,f) = \perp}{\mathbf{ftype\_in\_path}\,(P,\,(ctx,\,cld)\,ctxcld_2\,..\,ctxcld_k,\,f) = \mathbf{null}}$$

FTIP_CONS_TRUE

$$\frac{1.\ \mathbf{class\_fields}\,(cld) = \overline{fd} \quad 2.\ \mathbf{ftype\_in\_fds}\,(P,\,ctx,\,\overline{fd},\,f) = \tau}{\mathbf{ftype\_in\_path}\,(P,\,(ctx,\,cld)\,ctxcld_2\,..\,ctxcld_k,\,f) = \tau}$$

FTIP_CONS_FALSE

$$\frac{1.\ \mathbf{class\_fields}\,(cld) = \overline{fd} \quad 2.\ \mathbf{ftype\_in\_fds}\,(P,\ ctx,\ \overline{fd},\ f) = \mathbf{null} \quad 3.\ \mathbf{ftype\_in\_path}\,(P,\ ctxcld_2\ ..\ ctxcld_k,\ f) = \tau_{opt}}{\mathbf{ftype\_in\_path}\,(P,\ (ctx,\ cld)\ ctxcld_2\ ..\ ctxcld_k,\ f) = \tau_{opt}}$$

Field type (ftype_f)

$\boxed{\mathbf{ftype}\,(P,\ \tau,\ f) = \tau'}$ field type lookup

FTYPE

$$\frac{1.\ \mathbf{find\_path}\,(P,\ \tau) = \overline{ctxcld} \quad 2.\ \mathbf{ftype\_in\_path}\,(P,\ \overline{ctxcld},\ f) = \tau'}{\mathbf{ftype}\,(P,\ \tau,\ f) = \tau'}$$

Method definition in list (find_meth_def_in_list_f)

$\boxed{\mathbf{find\_meth\_def\_in\_list}\,(\overline{meth\_def},\ meth) = meth\_def_{opt}}$ meth. def. lookup (list)

FMDIL_EMPTY

$$\frac{}{\mathbf{find\_meth\_def\_in\_list}\,([\,],\ meth) = \mathbf{null}}$$

FMDIL_CONS_TRUE

$$\frac{1.\ meth\_def\ =\ cl\ meth\,(\overline{vd})\,\{\ meth\_body\ \}}{\mathbf{find\_meth\_def\_in\_list}\,(meth\_def\ meth\_def_2\ ..\ meth\_def_k,\ meth) = meth\_def}$$

FMDIL_CONS_FALSE

$$\frac{1.\ meth\_def\ =\ cl\ meth'\,(\overline{vd})\,\{\ meth\_body\ \} \quad 2.\ meth\ \neq\ meth' \quad 3.\ \mathbf{find\_meth\_def\_in\_list}\,(meth\_def_2\ ..\ meth\_def_k,\ meth) = meth\_def_{opt}}{\mathbf{find\_meth\_def\_in\_list}\,(meth\_def\ meth\_def_2\ ..\ meth\_def_k,\ meth) = meth\_def_{opt}}$$

Method definition in path (find_meth_def_in_path_f)

$\boxed{\textbf{find\_meth\_def\_in\_path}\,(\overline{ctxcld},\ meth) = ctxmeth\_def_{opt}}$ meth. def. lookup (path)

FMDIP_EMPTY

$$\frac{}{\textbf{find\_meth\_def\_in\_path}\,([\,],\ meth) = \textbf{null}}$$

FMDIP_CONS_TRUE

$$\frac{1.\ \textbf{class\_methods}\,(cld) = \overline{meth\_def} \quad 2.\ \textbf{find\_meth\_def\_in\_list}\,(\overline{meth\_def},\ meth) = meth\_def}{\textbf{find\_meth\_def\_in\_path}\,((ctx,\ cld)\ ctxcld_2\ ..\ ctxcld_k,\ meth) = (ctx,\ meth\_def)}$$

FMDIP_CONS_FALSE

$$\frac{1.\ \textbf{class\_methods}\,(cld) = \overline{meth\_def} \quad 2.\ \textbf{find\_meth\_def\_in\_list}\,(\overline{meth\_def},\ meth) = \textbf{null} \quad 3.\ \textbf{find\_meth\_def\_in\_path}\,(ctxcld_2\ ..\ ctxcld_k,\ meth) = ctxmeth\_def_{opt}}{\textbf{find\_meth\_def\_in\_path}\,((ctx,\ cld)\ ctxcld_2\ ..\ ctxcld_k,\ meth) = ctxmeth\_def_{opt}}$$

Method definition (find_meth_def_f)

$\boxed{\textbf{find\_meth\_def}\,(P,\ \tau,\ meth) = ctxmeth\_def_{opt}}$ method def. lookup in a type

FMD_NULL

$$\frac{1.\ \textbf{find\_path}\,(P,\ \tau) = \textbf{null}}{\textbf{find\_meth\_def}\,(P,\ \tau,\ meth) = \textbf{null}}$$

FMD_OPT

$$\frac{1.\ \textbf{find\_path}\,(P,\ \tau) = \overline{ctxcld} \quad 2.\ \textbf{find\_meth\_def\_in\_path}\,(\overline{ctxcld},\ meth) = ctxmeth\_def_{opt}}{\textbf{find\_meth\_def}\,(P,\ \tau,\ meth) = ctxmeth\_def_{opt}}$$

Method type (mtype_f)

$\boxed{\mathbf{mtype}\,(P,\,\tau,\,meth) = \pi}$ method type lookup

MTYPE

$$\frac{\begin{array}{l}1.\,\mathbf{find\_meth\_def}\,(P,\,\tau,\,meth) = (ctx,\,meth\_def)\\ 2.\,meth\_def = cl\;meth\,(\overline{cl_k\;var_k}^{\,k})\,\{\,meth\_body\,\}\\ 3.\,\mathbf{find\_type}\,(P,\,ctx,\,cl) = \tau'\\ 4.\,\overline{\mathbf{find\_type}\,(P,\,ctx,\,cl_k) = \tau_k}^{\,k}\\ 5.\,\pi = \overline{\tau_k}^{\,k} \;\rightarrow\; \tau'\end{array}}{\mathbf{mtype}\,(P,\,\tau,\,meth) = \pi}$$

C.2 LJAM lookup rules

Module definition – recursive part (find_md_rec_f)

$\boxed{\mathbf{find\_md\_rec}\,(RC,\,rn_1,\,mn,\,nn) = rnmd^{\mathsf{c}}_{opt}}$ module def. lookup (recursive part)

FMR_NULL

$$\frac{1.\,RC\,(rn) = \mathbf{null}}{\mathbf{find\_md\_rec}\,(RC,\,rn,\,mn,\,nn) = \mathbf{null}}$$

FMR_BOOTSTRAP_NULL

$$\frac{\begin{array}{l}1.\,RC\,(rn) = \mathbf{bootstrap\;repository}\,\{\,\overline{md^{\mathsf{c}}}\,;\,\phi\,\}\\ 2.\,\mathbf{find\_md\_in\_mds}\,(\overline{md^{\mathsf{c}}},\,mn) = \mathbf{null}\end{array}}{\mathbf{find\_md\_rec}\,(RC,\,rn,\,mn,\,nn) = \mathbf{null}}$$

FMR_BOOTSTRAP

$$\frac{\begin{array}{l}1.\,RC\,(rn) = \mathbf{bootstrap\;repository}\,\{\,\overline{md^{\mathsf{c}}}\,;\,\phi\,\}\\ 2.\,\mathbf{find\_md\_in\_mds}\,(\overline{md^{\mathsf{c}}},\,mn) = md^{\mathsf{c}}\end{array}}{\mathbf{find\_md\_rec}\,(RC,\,rn,\,mn,\,nn) = (rn,\,md^{\mathsf{c}})}$$

FMR_STANDARD_FAIL

$$\frac{\begin{array}{l}1.\,RC\,(rn_1) = \mathbf{repository}\;r\;\mathbf{child\;of}\;rn_2\,\{\,\overline{md^{\mathsf{c}}}\,;\,\phi\,\}\\ 2.\,\mathbf{size}\,(\mathbf{dom}\,RC) \leq nn\end{array}}{\mathbf{find\_md\_rec}\,(RC,\,rn_1,\,mn,\,nn) = \mathbf{null}}$$

FMR_STANDARD_REC

$$\frac{\begin{array}{l}1.\,RC\,(rn_1) = \mathbf{repository}\;r\;\mathbf{child\;of}\;rn_2\,\{\,\overline{md^{\mathsf{c}}}\,;\,\phi\,\}\\ 2.\,\mathbf{size}\,(\mathbf{dom}\,RC) > nn\\ 3.\,\mathbf{find\_md\_rec}\,(RC,\,rn_2,\,mn,\,nn+1) = (rn_3,\,md^{\mathsf{c}})\end{array}}{\mathbf{find\_md\_rec}\,(RC,\,rn_1,\,mn,\,nn) = (rn_3,\,md^{\mathsf{c}})}$$

FMR_STANDARD_SELF

$$\frac{\begin{array}{l}1.\ RC\,(rn_1) = \textbf{repository}\ r\ \textbf{child of}\ rn_2\,\{\,\overline{md^{\mathsf{c}}}\,;\,\phi\,\}\\ 2.\ \textbf{size}\,(\textbf{dom}\ RC) > nn\\ 3.\ \textbf{find\_md\_rec}\,(RC,\ rn_2,\ mn,\ nn+1) = \textbf{null}\\ 4.\ \textbf{find\_md\_in\_mds}\,(\overline{md^{\mathsf{c}}},\ mn) = md^{\mathsf{c}}\end{array}}{\textbf{find\_md\_rec}\,(RC,\ rn_1,\ mn,\ nn) = (rn_1,\ md^{\mathsf{c}})}$$

FMR_STANDARD_NULL

$$\frac{\begin{array}{l}1.\ RC\,(rn_1) = \textbf{repository}\ r\ \textbf{child of}\ rn_2\,\{\,\overline{md^{\mathsf{c}}}\,;\,\phi\,\}\\ 2.\ \textbf{size}\,(\textbf{dom}\ RC) > nn\\ 3.\ \textbf{find\_md\_rec}\,(RC,\ rn_2,\ mn,\ nn+1) = \textbf{null}\\ 4.\ \textbf{find\_md\_in\_mds}\,(\overline{md^{\mathsf{c}}},\ mn) = \textbf{null}\end{array}}{\textbf{find\_md\_rec}\,(RC,\ rn_1,\ mn,\ nn) = \textbf{null}}$$

Module definition (find_md_f)

$\boxed{\textbf{find\_md}\,(RC,\ rn,\ mn) = rnmd^{\mathsf{c}}_{opt}}$ module def. lookup

FM_DEF

$$\frac{1.\ \textbf{find\_md\_rec}\,(RC,\ rn,\ mn,\ 0) = rnmd^{\mathsf{c}}_{opt}}{\textbf{find\_md}\,(RC,\ rn,\ mn) = rnmd^{\mathsf{c}}_{opt}}$$

Class – core (find_cld_in_core_f)

$\boxed{\textbf{find\_cld\_in\_core}\,(P,\ fqn) = ctxcld_{opt}}$ class lookup in the core library module

FCIC_NO_REP_EX

$$\frac{1.\ RC\,(\textbf{bootstrap\_r}) = \textbf{null}}{\textbf{find\_cld\_in\_core}\,((RC,\ MH),\ fqn) = \textbf{null}}$$

FCIC_NOT_BOOTSTRAP_EX

$$\frac{1.\ RC\,(\textbf{bootstrap\_r}) = \textbf{repository}\ r\ \textbf{child of}\ rn\,\{\,\overline{md^{\mathsf{c}}}\,;\,\phi\,\}}{\textbf{find\_cld\_in\_core}\,((RC,\ MH),\ fqn) = \textbf{null}}$$

FCIC_NO_CORE_EX

$$\frac{\begin{array}{l}1.\ RC\,(\textbf{bootstrap\_r}) = \textbf{bootstrap repository}\,\{\,\overline{md^{\mathsf{c}}}\,;\,\phi\,\}\\ 2.\ \textbf{find\_md\_in\_mds}\,(\overline{md^{\mathsf{c}}},\ \textbf{core\_m}) = \textbf{null}\end{array}}{\textbf{find\_cld\_in\_core}\,((RC,\ MH),\ fqn) = \textbf{null}}$$

FCIC_NO_CORE_MI_EX

$$\frac{1.\ RC\ (\mathbf{bootstrap\_r}) = \mathbf{bootstrap\,repository}\ \{\ \overline{md^{\mathsf{c}}}\ ;\ \phi\ \} \quad 2.\ \mathbf{find\_md\_in\_mds}\ (\overline{md^{\mathsf{c}}},\ \mathbf{core\_m}) = md^{\mathsf{c}} \quad 3.\ \phi\ (md^{\mathsf{c}}) = \mathbf{null}}{\mathbf{find\_cld\_in\_core}\ ((RC,\ MH),\ fqn) = \mathbf{null}}$$

FCIC_NO_MDMIS_EX

$$\frac{1.\ RC\ (\mathbf{bootstrap\_r}) = \mathbf{bootstrap\,repository}\ \{\ \overline{md^{\mathsf{c}}}\ ;\ \phi\ \} \quad 2.\ \mathbf{find\_md\_in\_mds}\ (\overline{md^{\mathsf{c}}},\ \mathbf{core\_m}) = md^{\mathsf{c}} \quad 3.\ \phi\ (md^{\mathsf{c}}) = mi \quad 4.\ MH\ (mi) = \mathbf{null}}{\mathbf{find\_cld\_in\_core}\ ((RC,\ MH),\ fqn) = \mathbf{null}}$$

FCIC_FALSE

$$\frac{1.\ RC\ (\mathbf{bootstrap\_r}) = \mathbf{bootstrap\,repository}\ \{\ \overline{md^{\mathsf{c}}}\ ;\ \phi\ \} \quad 2.\ \mathbf{find\_md\_in\_mds}\ (\overline{md^{\mathsf{c}}},\ \mathbf{core\_m}) = md^{\mathsf{c}} \quad 3.\ \phi\ (md^{\mathsf{c}}) = mi \quad 4.\ MH\ (mi) = (\mathbf{module}\ mn\ \{\ \overline{cld}\ \overline{m}\ \overline{fqn}\ \},\ \overline{mi}) \quad 5.\ \mathbf{find\_cld\_in\_module}\ (\overline{cld},\ fqn) = \mathbf{null}}{\mathbf{find\_cld\_in\_core}\ ((RC,\ MH),\ fqn) = \mathbf{null}}$$

FCIC_TRUE

$$\frac{1.\ RC\ (\mathbf{bootstrap\_r}) = \mathbf{bootstrap\,repository}\ \{\ \overline{md^{\mathsf{c}}}\ ;\ \phi\ \} \quad 2.\ \mathbf{find\_md\_in\_mds}\ (\overline{md^{\mathsf{c}}},\ \mathbf{core\_m}) = md^{\mathsf{c}} \quad 3.\ \phi\ (md^{\mathsf{c}}) = mi \quad 4.\ MH\ (mi) = (\mathbf{module}\ mn\ \{\ \overline{cld}\ \overline{m}\ \overline{fqn}\ \},\ \overline{mi}) \quad 5.\ \mathbf{find\_cld\_in\_module}\ (\overline{cld},\ fqn) = cld \quad 6.\ \mathbf{package\_name}\ (cld) = pn}{\mathbf{find\_cld\_in\_core}\ ((RC,\ MH),\ fqn) = (mi.pn,\ cld)}$$

Class – module (find_cld_in_module_f)

$\boxed{\mathbf{find\_cld\_in\_module}\ (\overline{cld},\ fqn) = cld_{opt}}$ class lookup in an import

FCIM_EMPTY

$$\frac{}{\mathbf{find\_cld\_in\_module}\ ([\,],\ fqn) = \mathbf{null}}$$

FCIM_NULL

$$\frac{1.\ \neg\mathbf{distinct\_fqns}\ (cld\ cld_2\ ..\ cld_k)}{\mathbf{find\_cld\_in\_module}\ (cld\ cld_2\ ..\ cld_k,\ fqn) = \mathbf{null}}$$

FCIM_CONS_TRUE

$$\frac{1.\ \mathbf{distinct\_fqns}\,(cld\ cld_2\ ..\ cld_k) \quad 2.\ cld = \mathbf{package}\,pn\,;\ \mathbf{public\,class}\,dcl\,\mathbf{extends}\,cl\,\{\,\overline{fd}\ \overline{meth\_def}\,\}}{\mathbf{find\_cld\_in\_module}\,(cld\ cld_2\ ..\ cld_k,\ pn.dcl) = cld}$$

FCIM_CONS_FALSE

$$\frac{1.\ \mathbf{distinct\_fqns}\,(cld\ cld_2\ ..\ cld_k) \quad 2.\ cld = \mathbf{package}\,pn'\,;\ am\,\mathbf{class}\,dcl'\,\mathbf{extends}\,cl\,\{\,\overline{fd}\ \overline{meth\_def}\,\} \quad 3.\ pn \neq pn' \ \vee\ am \neq \mathbf{public} \ \vee\ dcl \neq dcl' \quad 4.\ \mathbf{find\_cld\_in\_module}\,(cld_2\ ..\ cld_k,\ pn.dcl) = cld_{opt}}{\mathbf{find\_cld\_in\_module}\,(cld\ cld_2\ ..\ cld_k,\ pn.dcl) = cld_{opt}}$$

Class – imports (find_cld_in_imports_f)

$\boxed{\mathbf{find\_cld\_in\_imports}\,(MH,\ \overline{mi},\ fqn) = ctxcld_{opt}}$ class lookup in imports

FCII_EMPTY

$$\frac{}{\mathbf{find\_cld\_in\_imports}\,(MH,\ [\,],\ fqn) = \mathbf{null}}$$

FCII_NULL

$$\frac{1.\ \neg\,(\mathbf{acyclic\_mh}\,MH\ \wedge\ mi \in \mathbf{dom}\,(MH)\ \wedge\ mi_2\ ..\ mi_k\ \subseteq\ \mathbf{dom}\,(MH))}{\mathbf{find\_cld\_in\_imports}\,(MH,\ mi\ mi_2\ ..\ mi_k,\ fqn) = \mathbf{null}}$$

FCII_SKIP

$$\frac{1.\ \mathbf{acyclic\_mh}\,MH\ \wedge\ MH\,(mi) = (md,\ \overline{mi})\ \wedge\ mi_2\ ..\ mi_k\ \subseteq\ \mathbf{dom}\,(MH) \quad 2.\ md = \mathbf{module}\,mn\,\{\,\overline{cld}\,\overline{m}\,\overline{fqn}\,\}\ \wedge\ fqn \notin \overline{fqn} \quad 3.\ \mathbf{find\_cld\_in\_imports}\,(MH,\ mi_2\ ..\ mi_k,\ fqn) = ctxcld_{opt}}{\mathbf{find\_cld\_in\_imports}\,(MH,\ mi\ mi_2\ ..\ mi_k,\ fqn) = ctxcld_{opt}}$$

FCII_REC

$$\frac{1.\ \mathbf{acyclic\_mh}\,MH\ \wedge\ MH\,(mi) = (md,\ \overline{mi})\ \wedge\ mi_2\ ..\ mi_k\ \subseteq\ \mathbf{dom}\,(MH) \quad 2.\ md = \mathbf{module}\,mn\,\{\,\overline{cld}\,\overline{m}\,\overline{fqn}\,\}\ \wedge\ fqn \in \overline{fqn} \quad 3.\ \mathbf{find\_cld\_in\_imports}\,(MH,\ \overline{mi},\ fqn) = ctxcld}{\mathbf{find\_cld\_in\_imports}\,(MH,\ mi\ mi_2\ ..\ mi_k,\ fqn) = ctxcld}$$

FCII_SELF

$$\frac{1.\ \mathbf{acyclic\_mh}\,MH\ \wedge\ MH\,(mi) = (md,\ \overline{mi})\ \wedge\ mi_2\ ..\ mi_k\ \subseteq\ \mathbf{dom}\,(MH) \quad 2.\ md = \mathbf{module}\,mn\,\{\,\overline{cld}\,\overline{m}\,\overline{fqn}\,\}\ \wedge\ fqn \in \overline{fqn} \quad 3.\ \mathbf{find\_cld\_in\_imports}\,(MH,\ \overline{mi},\ fqn) = \mathbf{null} \quad 4.\ \mathbf{find\_cld\_in\_module}\,(\overline{cld},\ fqn) = cld\ \wedge\ \mathbf{package\_name}\,(cld) = pn}{\mathbf{find\_cld\_in\_imports}\,(MH,\ mi\ mi_2\ ..\ mi_k,\ fqn) = (mi.pn,\ cld)}$$

FCII_NEXT

$$\frac{1.\ \mathbf{acyclic\_mh}\ MH\ \wedge\ MH\,(mi) = (md,\ \overline{mi})\ \wedge\ mi_2\,..\,mi_k\ \subseteq\ \mathbf{dom}\,(MH)\quad 2.\ md = \mathbf{module}\ mn\,\{\,\overline{cld}\,\overline{m}\,\overline{fqn}\,\}\ \wedge\ fqn \in \overline{fqn}\quad 3.\ \mathbf{find\_cld\_in\_imports}\,(MH,\ \overline{mi},\ fqn) = \mathbf{null}\quad 4.\ \mathbf{find\_cld\_in\_module}\,(\overline{cld},\ fqn) = \mathbf{null}\quad 5.\ \mathbf{find\_cld\_in\_imports}\,(MH,\ mi_2\,..\,mi_k,\ fqn) = ctxcld_{opt}}{\mathbf{find\_cld\_in\_imports}\,(MH,\ mi\ mi_2\,..\,mi_k,\ fqn) = ctxcld_{opt}}$$

Class – self (find_cld_in_self_f)

$\boxed{\mathbf{find\_cld\_in\_self}\,(\overline{cld},\ pn,\ fqn) = cld_{opt}}$ class lookup in the same module

FCIS_EMPTY

$$\frac{}{\mathbf{find\_cld\_in\_self}\,([\,],\ pn,\ fqn) = \mathbf{null}}$$

FCIS_NULL

$$\frac{1.\ \neg\mathbf{distinct\_fqns}\,(cld\ cld_2\,..\,cld_k)}{\mathbf{find\_cld\_in\_self}\,(cld\ cld_2\,..\,cld_k,\ pn,\ fqn) = \mathbf{null}}$$

FCIS_CONS_TRUE

$$\frac{1.\ \mathbf{distinct\_fqns}\,(cld\ cld_2\,..\,cld_k)\quad 2.\ cld = \mathbf{package}\ pn'\,;\ am\ \mathbf{class}\ dcl\ \mathbf{extends}\ cl\,\{\,\overline{fd}\ \overline{meth\_def}\,\}\quad 3.\ pn = pn'\ \vee\ am = \mathbf{public}}{\mathbf{find\_cld\_in\_self}\,(cld\ cld_2\,..\,cld_k,\ pn,\ pn'.dcl) = cld}$$

FCIS_CONS_FALSE

$$\frac{1.\ \mathbf{distinct\_fqns}\,(cld\ cld_2\,..\,cld_k)\quad 2.\ cld = \mathbf{package}\ pn''\,;\ am\ \mathbf{class}\ dcl'\ \mathbf{extends}\ cl\,\{\,\overline{fd}\ \overline{meth\_def}\,\}\quad 3.\ (pn \neq pn'\ \wedge\ am \neq \mathbf{public})\ \vee\ pn' \neq pn''\ \vee\ dcl \neq dcl'\quad 4.\ \mathbf{find\_cld\_in\_self}\,(cld_2\,..\,cld_k,\ pn,\ pn'.dcl) = cld_{opt}}{\mathbf{find\_cld\_in\_self}\,(cld\ cld_2\,..\,cld_k,\ pn,\ pn'.dcl) = cld_{opt}}$$

Class (find_cld_f)

$\boxed{\textbf{find\_cld}\,(P,\ ctx,\ fqn) = ctxcld_{opt}}$ class lookup

FC_CORE

$$\frac{1.\ \textbf{find\_cld\_in\_core}\,(P,\ fqn) = ctxcld}{\textbf{find\_cld}\,(P,\ ctx,\ fqn) = ctxcld}$$

FC_NULL

$$\frac{\begin{array}{l}1.\ \textbf{find\_cld\_in\_core}\,((RC,\ MH),\ fqn) = \textbf{null}\\ 2.\ MH\ (mi) = \textbf{null}\end{array}}{\textbf{find\_cld}\,((RC,\ MH),\ mi.pn,\ fqn) = \textbf{null}}$$

FC_IMPORTS

$$\frac{\begin{array}{l}1.\ \textbf{find\_cld\_in\_core}\,((RC,\ MH),\ fqn) = \textbf{null}\\ 2.\ MH\ (mi) = (md,\ \overline{mi})\\ 3.\ md = \textbf{module}\ mn\ \{\ \overline{cld}\ \overline{m}\ \overline{fqn}\ \}\\ 4.\ \textbf{find\_cld\_in\_imports}\,(MH,\ \overline{mi},\ fqn) = ctxcld\end{array}}{\textbf{find\_cld}\,((RC,\ MH),\ mi.pn,\ fqn) = ctxcld}$$

FC_SELF

$$\frac{\begin{array}{l}1.\ \textbf{find\_cld\_in\_core}\,((RC,\ MH),\ fqn) = \textbf{null}\\ 2.\ MH\ (mi) = (md,\ \overline{mi})\\ 3.\ md = \textbf{module}\ mn\ \{\ \overline{cld}\ \overline{m}\ \overline{fqn}\ \}\\ 4.\ \textbf{find\_cld\_in\_imports}\,(MH,\ \overline{mi},\ fqn) = \textbf{null}\\ 5.\ \textbf{find\_cld\_in\_self}\,(\overline{cld},\ pn,\ fqn) = cld\\ 6.\ \textbf{package\_name}\,(cld) = pn'\end{array}}{\textbf{find\_cld}\,((RC,\ MH),\ mi.pn,\ fqn) = (mi.pn',\ cld)}$$

FC_FAIL

$$\frac{\begin{array}{l}1.\ \textbf{find\_cld\_in\_core}\,((RC,\ MH),\ fqn) = \textbf{null}\\ 2.\ MH\ (mi) = (md,\ \overline{mi})\\ 3.\ md = \textbf{module}\ mn\ \{\ \overline{cld}\ \overline{m}\ \overline{fqn}\ \}\\ 4.\ \textbf{find\_cld\_in\_imports}\,(MH,\ \overline{mi},\ fqn) = \textbf{null}\\ 5.\ \textbf{find\_cld\_in\_self}\,(\overline{cld},\ pn,\ fqn) = \textbf{null}\end{array}}{\textbf{find\_cld}\,((RC,\ MH),\ mi.pn,\ fqn) = \textbf{null}}$$

C.3 LJAM context insertion rules

Context insertion for a statement

$\boxed{\vdash_{ctx} s^{\mathsf{c}} \rightsquigarrow s}$ context insertion for a statement

$$\frac{1.\ \overline{\vdash_{ctx} s_k^{\mathsf{c}} \rightsquigarrow s_k}^{\,k}}{\vdash_{ctx} \{\ \overline{s_k^{\mathsf{c}}}^{\,k}\ \} \rightsquigarrow \{\ \overline{s_k}^{\,k}\ \}}\ \text{CI\_S\_BLOCK} \qquad \frac{}{\vdash_{ctx} var = x\,; \rightsquigarrow var = x\,;}\ \text{CI\_S\_VAR\_ASSIGN}$$

$$\text{CI\_S\_FIELD\_READ}\quad \overline{\vdash_{ctx} var = x\,.\,f\,; \rightsquigarrow var = x\,.\,f\,;} \qquad \overline{\vdash_{ctx} x\,.\,f = y\,; \rightsquigarrow x\,.\,f = y\,;}\ \text{CI\_S\_FIELD\_WRITE}$$

$$\frac{1.\ \vdash_{ctx} s_1^{\mathsf{c}} \rightsquigarrow s_1 \qquad 2.\ \vdash_{ctx} s_2^{\mathsf{c}} \rightsquigarrow s_2}{\vdash_{ctx} \textbf{if}\ (x == y)\ s_1^{\mathsf{c}}\ \textbf{else}\ s_2^{\mathsf{c}} \rightsquigarrow \textbf{if}\ (x == y)\ s_1\ \textbf{else}\ s_2}\ \text{CI\_S\_IF}$$

$$\frac{}{\vdash_{ctx} var = x\,.\,meth\,(\overline{y_k}^{\,k})\,; \rightsquigarrow var = x\,.\,meth\,(\overline{y_k}^{\,k})\,;}\ \text{CI\_S\_MCALL}$$

$$\frac{}{\vdash_{ctx} var = \textbf{new}\ cl\,(); \rightsquigarrow var = \textbf{new}_{ctx}\,cl();}\ \text{CI\_S\_NEW}$$

Context insertion for a method definition

$\boxed{\vdash_{ctx} meth\_def^{\mathsf{c}} \rightsquigarrow meth\_def}$ context insertion for method def.'s

$$\frac{1.\ \overline{\vdash_{ctx} s^{\mathsf{c}} \rightsquigarrow s}^{\,k}}{\vdash_{ctx} cl\ meth\,(\overline{vd})\ \{\ \overline{s_k^{\mathsf{c}}}^{\,k}\ \textbf{return}\ y\,;\ \} \rightsquigarrow cl\ meth\,(\overline{vd})\ \{\ \overline{s}^{\,k}\ \textbf{return}\ y\,;\ \}}\ \text{CI\_METH\_DEF}$$

Context insertion for a class definition

$\boxed{\vdash_{mi} cld^{\mathsf{c}} \rightsquigarrow cld}$ context insertion for class def.'s

$$\frac{\begin{array}{l}1.\ cld^{\mathsf{c}} = \textbf{package}\ pn\,;\ am\ \textbf{class}\ dcl\ \textbf{extends}\ cl\ \{\ \overline{fd}\ \overline{meth\_def_k^{\mathsf{c}}}^{\,k}\ \} \\ 2.\ \overline{\vdash_{mi.pn} meth\_def_k^{\mathsf{c}} \rightsquigarrow meth\_def_k}^{\,k} \\ 3.\ cld = \textbf{package}\ pn\,;\ am\ \textbf{class}\ dcl\ \textbf{extends}\ cl\ \{\ \overline{fd}\ \overline{meth\_def_k}^{\,k}\ \}\end{array}}{\vdash_{mi} cld^{\mathsf{c}} \rightsquigarrow cld}\ \text{CI\_CLD}$$

Context insertion for a module definition

$\boxed{\vdash_{mi} md^{\mathsf{c}} \rightsquigarrow md}$ module def. translation

CI_MODULE

$$\frac{1.\ \overline{\vdash_{mi} cld_k^{\mathsf{c}} \rightsquigarrow cld_k}^{\,k}}{\vdash_{mi} \mathbf{module}\ mn\ \{\ \overline{cld_k^{\mathsf{c}}}^{\,k}\ \overline{m}\ \overline{fqn}\ \} \rightsquigarrow \mathbf{module}\ mn\ \{\ \overline{cld_k}^{\,k}\ \overline{m}\ \overline{fqn}\ \}}$$

C.4 iJAM lookup rules

Class (find_cld_f)

$\boxed{\mathbf{find\_cld}\ (P,\ ctx,\ fqn) = ctxcld_{opt}}$ class lookup

FC_ERR

$$\frac{1.\ \neg\mathbf{no\_core\_renaming}\ P}{\mathbf{find\_cld}\ (P,\ ctx,\ fqn) = \mathbf{null}}$$

FC_CORE

$$\frac{\begin{array}{l}1.\ \mathbf{no\_core\_renaming}\ P\\ 2.\ \mathbf{find\_cld\_in\_core}\ (P,\ fqn) = ctxcld\end{array}}{\mathbf{find\_cld}\ (P,\ ctx,\ fqn) = ctxcld}$$

FC_NULL

$$\frac{\begin{array}{l}1.\ \mathbf{no\_core\_renaming}\ (RC,\ MH)\\ 2.\ \mathbf{find\_cld\_in\_core}\ ((RC,\ MH),\ fqn) = \mathbf{null}\\ 3.\ MH\ (mi) = \mathbf{null}\end{array}}{\mathbf{find\_cld}\ ((RC,\ MH),\ mi.pn,\ fqn) = \mathbf{null}}$$

FC_SELF

$$\frac{\begin{array}{l}1.\ \mathbf{no\_core\_renaming}\ (RC,\ MH)\\ 2.\ \mathbf{find\_cld\_in\_core}\ ((RC,\ MH),\ fqn) = \mathbf{null}\\ 3.\ MH\ (mi) = (md,\ \overline{mibr})\\ 4.\ md = repl\ \mathbf{module}\ mn\ \{\ \overline{cld}\ \overline{imp_k}^{\,k}\ \overline{fqn}\ \}\\ 5.\ \mathbf{find\_cld\_in\_self}\ (\overline{cld},\ pn,\ fqn) = cld\\ 6.\ \mathbf{package\_name}\ (cld) = pn'\end{array}}{\mathbf{find\_cld}\ ((RC,\ MH),\ mi.pn,\ fqn) = (mi.pn',\ cld)}$$

FC_IMPORTS

$$\frac{\begin{array}{l}1.\ \mathbf{no\_core\_renaming}\,(RC,\ MH)\\ 2.\ \mathbf{find\_cld\_in\_core}\,((RC,\ MH),\ fqn) = \mathbf{null}\\ 3.\ MH\ (mi) = (md,\ \overline{mibr})\\ 4.\ md = repl\ \mathbf{module}\ mn\ \{\ \overline{cld}\ \overline{imp_k}^{\,k}\ \overline{fqn}\ \}\\ 5.\ \mathbf{find\_cld\_in\_self}\ (\overline{cld},\ pn,\ fqn) = \mathbf{null}\\ 6.\ \mathbf{find\_cld\_in\_imports}\,(MH,\ \overline{mibr},\ fqn) = ctxcld_{opt}\end{array}}{\mathbf{find\_cld}\,((RC,\ MH),\ mi.pn,\ fqn) = ctxcld_{opt}}$$

Class – imports (find_cld_in_imports_f)

$\boxed{\mathbf{find\_cld\_in\_imports}\,(MH,\ \overline{mibr},\ fqn) = ctxcld_{opt}}$ class lookup in imports

FCII_EMPTY

$$\frac{}{\mathbf{find\_cld\_in\_imports}\,(MH,\ [\,],\ fqn) = \mathbf{null}}$$

FCII_NULL

$$\frac{1.\ \neg\left(\begin{array}{l}\mathbf{acyclic\_mh}\ MH\ \ \wedge\ \ mi \in \mathbf{dom}\,(MH)\ \wedge\\ \mathbf{mis\_of}\ (mibr_2\ ..\ mibr_k)\ \subseteq\ \mathbf{dom}\,(MH)\end{array}\right)}{\mathbf{find\_cld\_in\_imports}\,(MH,\ mi\ br\ mibr_2\ ..\ mibr_k,\ fqn) = \mathbf{null}}$$

FCII_SKIP

$$\frac{\begin{array}{l}1.\left(\begin{array}{l}\mathbf{acyclic\_mh}\ MH\ \ \wedge\ \ MH\ (mi) = (md,\ \overline{mibr})\ \wedge\\ \mathbf{mis\_of}\ (mibr_2\ ..\ mibr_k)\ \subseteq\ \mathbf{dom}\,(MH)\end{array}\right)\\ 2.\left(\begin{array}{l}\left(md = repl\ \mathbf{module}\ mn\ \{\ \overline{cld}\ \overline{imp_j}^{\,j}\ \overline{fqn}\ \}\ \ \wedge\ \ br[fqn] \notin \overline{fqn}\right)\ \vee\\ (fqn \notin \mathbf{dom}\,(br)\ \ \wedge\ \ fqn \in \mathbf{ran}\,(br))\end{array}\right)\\ 3.\ \mathbf{find\_cld\_in\_imports}\,(MH,\ mibr_2\ ..\ mibr_k,\ fqn) = ctxcld_{opt}\end{array}}{\mathbf{find\_cld\_in\_imports}\,(MH,\ mi\ br\ mibr_2\ ..\ mibr_k,\ fqn) = ctxcld_{opt}}$$

FCII_SELF

$$\frac{\begin{array}{l}1.\left(\begin{array}{l}\mathbf{acyclic\_mh}\ MH\ \ \wedge\ \ MH\ (mi) = (md,\ \overline{mibr})\ \wedge\\ \mathbf{mis\_of}\ (mibr_2\ ..\ mibr_k)\ \subseteq\ \mathbf{dom}\,(MH)\end{array}\right)\\ 2.\left(\begin{array}{l}\left(md = repl\ \mathbf{module}\ mn\ \{\ \overline{cld}\ \overline{imp_j}^{\,j}\ \overline{fqn}\ \}\ \ \wedge\ \ br[fqn] \in \overline{fqn}\right)\ \wedge\\ (fqn \in \mathbf{dom}\,(br)\ \ \vee\ \ fqn \notin \mathbf{ran}\,(br))\end{array}\right)\\ 3.\ \mathbf{find\_cld\_in\_module}\,(\overline{cld},\ br[fqn]) = cld\ \ \wedge\ \ \mathbf{package\_name}\,(cld) = pn\end{array}}{\mathbf{find\_cld\_in\_imports}\,(MH,\ mi\ br\ mibr_2\ ..\ mibr_k,\ fqn) = (mi.pn,\ cld)}$$

FCII_REC

$$\frac{\begin{array}{ll}1. & \left(\begin{array}{l}\mathbf{acyclic\_mh}\ MH\ \wedge\ MH\,(mi) = (md,\ \overline{mibr})\ \wedge \\ \mathbf{mis\_of}\,(mibr_2\,..\,mibr_k)\ \subseteq\ \mathbf{dom}\,(MH)\end{array}\right) \\ 2. & \left(\begin{array}{l}\left(md = repl\ \mathbf{module}\ mn\ \{\,\overline{cld}\ \overline{imp_j}^{\,j}\ \overline{fqn}\,\}\ \wedge\ br[fqn] \in \overline{fqn}\right)\ \wedge \\ (fqn \in \mathbf{dom}\,(br)\ \vee\ fqn \notin \mathbf{ran}\,(br))\end{array}\right) \\ 3. & \mathbf{find\_cld\_in\_module}\,(\overline{cld},\ br[fqn]) = \mathbf{null} \\ 4. & \mathbf{find\_cld\_in\_imports}\,(MH,\ \overline{mibr},\ br[fqn]) = ctxcld\end{array}}{\mathbf{find\_cld\_in\_imports}\,(MH,\ mi\ br\ mibr_2\,..\,mibr_k,\ fqn) = ctxcld}$$

FCII_NEXT

$$\frac{\begin{array}{ll}1. & \left(\begin{array}{l}\mathbf{acyclic\_mh}\ MH\ \wedge\ MH\,(mi) = (md,\ \overline{mibr})\ \wedge \\ \mathbf{mis\_of}\,(mibr_2\,..\,mibr_k)\ \subseteq\ \mathbf{dom}\,(MH)\end{array}\right) \\ 2. & \left(\begin{array}{l}\left(md = repl\ \mathbf{module}\ mn\ \{\,\overline{cld}\ \overline{imp_j}^{\,j}\ \overline{fqn}\,\}\ \wedge\ br[fqn] \in \overline{fqn}\right)\ \wedge \\ (fqn \in \mathbf{dom}\,(br)\ \vee\ fqn \notin \mathbf{ran}\,(br))\end{array}\right) \\ 3. & \mathbf{find\_cld\_in\_module}\,(\overline{cld},\ br[fqn]) = \mathbf{null} \\ 4. & \mathbf{find\_cld\_in\_imports}\,(MH,\ \overline{mibr},\ br[fqn]) = \mathbf{null} \\ 5. & \mathbf{find\_cld\_in\_imports}\,(MH,\ mibr_2\,..\,mibr_k,\ fqn) = ctxcld_{opt}\end{array}}{\mathbf{find\_cld\_in\_imports}\,(MH,\ mi\ br\ mibr_2\,..\,mibr_k,\ fqn) = ctxcld_{opt}}$$

D

iJAM example Java source code

D.1 XMLParser.Parser

```
public class Parser {
  private static int instances = 0;
  public final int instance;

  public Parser() {
    instance = ++instances;
  }
}
```

D.2 XSLT.Config

```
public class Config {
  public void edit() {
    System.out.println("XSLT::Config using " + new Parser().instance
        + ". instance of Parser.");
  }
}
```

D.3 ServletEngine.Config

```
public class Config {
  public void edit() {
    System.out.println("ServletEngine::Config using "
        + new Parser().instance + ". instance of Parser.");
  }
}
```

D.4 ServletEngine.UnitTest

```
public class UnitTest {
  public void run() {
    System.out.println("ServletEngine::UnitTest complete.");
  }
}
```

D.5 WebCalendar.UnitTest

```
public class UnitTest {
  public void run() {
    System.out.println("WebCalendar::UnitTest complete.");
  }
}
```

D.6 WebCalendar.Main

```
public class Main {
  public static void main(String[] args) {
    new UnitTest().run();
    new XSLT_Config().edit();
    new Config().edit();
  }
}
```

Bibliography

[1] The Coq Proof Assistant. http://coq.inria.fr/. (Cited on pages 34 and 38.)

[2] HOL-4. http://hol.sourceforge.net. (Cited on pages 34 and 38.)

[3] Isabelle 2008. http://isabelle.in.tum.de/. (Cited on pages 34, 35, 37, 38, 39, and 58.)

[4] Twelf. http://www.cs.cmu.edu/~twelf/. (Cited on page 34.)

[5] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer, 1996. (Cited on page 24.)

[6] Gavin Bierman, Matthew J. Parkinson, and Andrew Pitts. MJ: An Imperative Core Calculus for Java and Java with Effects. Technical Report 563, Computer Laboratory, University of Cambridge, April 2003. (Cited on page 57.)

[7] John Billings, Peter Sewell, Mark Shinwell, and Rok Strniša. Type-Safe Distributed Programming for OCaml. In *Proceedings of Workshop on ML*, pages 20–31. ACM Press, September 2006. (Cited on page 147.)

[8] Bard Bloom, John Field, Nathaniel Nystrom, Johan Östlund, Gregor Richards, Rok Strniša, Jan Vitek, and Tobias Wrigstad. Thorn—Robust, Concurrent, Extensible Scripting on the JVM. In *Proceedings of OOPSLA*, volume 42(10) of *ACM SIGPLAN Notices*, pages 499–514. ACM Press, October 2007. (Cited on pages 37 and 139.)

[9] Gilad Bracha and William R. Cook. Mixin-based Inheritance. In *Proceedings of OOPSLA*, volume 25(10) of *ACM SIGPLAN Notices*, pages 303–311. ACM Press, October 1990. (Cited on page 57.)

[10] Manfred Broy and Ernst Denert, editors. *Software Pioneers: Contributions to Software Engineering*. Springer, 2002. (Cited on page 23.)

[11] Alex Buckley. *Flexible Dynamic Linking*. PhD thesis, Imperial College London, February 2007. (Cited on page 49.)

[12] John Corwin, David F. Bacon, David Grove, and Chet Murthy. MJ: A Rational Module System for Java and its Applications. In *Proceedings of OOPSLA*, volume 38(11) of *ACM SIGPLAN Notices*, pages 241–254. ACM Press, November 2003. (Cited on page 84.)

[13] Karl Crary, Robert Harper, and Sidd Puri. What is a Recursive Module? In *Proceedings of PLDI*, volume 34(5) of *ACM SIGPLAN Notices*, pages 50–63. ACM Press, May 1999. (Cited on page 24.)

[14] Benjamin Delaware, William R. Cook, and Don Batory. A Machine-Checked Model of Safe Composition. http://www.cs.utexas.edu/~bendy/featurejava.php, October 2008. (Cited on pages 35, 41, 58, 84, and 154.)

[15] MSDN's .NET Framework Developer's Guide. Assemblies in the Common Language Runtime. http://msdn.microsoft.com/, 2008. (Cited on page 47.)

[16] Sophia Drossopoulou and Susan Eisenbach. The Java Type System is Sound - Probably. In *Proceedings of ECOOP*, volume 1241 of *LNCS*. Springer, June 1997. (Cited on page 36.)

[17] Len Erlikh. Leveraging Legacy System Dollars for E-Business. *IT Professional*, 2 (3):17–23, 2000. (Cited on page 34.)

[18] Matthew Flatt and Matthias Felleisen. Units: Cool Modules for HOT Languages. In *Proceedings of PLDI*, volume 33(5) of *ACM SIGPLAN Notices*, pages 236–248. ACM Press, May 1998. (Cited on page 50.)

[19] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java™ Language Specification*. Sun Microsystems, Inc., Third edition, May 2005. (Cited on pages 24, 25, 41, and 57.)

[20] Florian Haftmann. *Code generation from Isabelle/HOL theories*, November 2007. (Cited on page 84.)

[21] Richard S Hall. OSGi and Gravity Service Binder Tutorial. http://oscar-osgi.sourceforge.net/tutorial/, 2003. (Cited on page 43.)

[22] Robert Harper and Benjamin C Pierce. Design Considerations for ML-Style Module Systems. In Benjamin C Pierce, editor, *Advanced Topic in Types and Programming Languages*, pages 293–346. MIT Press, 2005. (Cited on page 24.)

[23] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A Minimal Core Calculus for Java and GJ. In *Proceedings of OOPSLA*, volume 34(10)

of *ACM SIGPLAN Notices*, pages 132–146. ACM Press, October 1999. (Cited on pages 36, 41, 57, and 69.)

[24] INRIA. Objective Caml. http://caml.inria.fr/ocaml/, 2008. Version 3.11. (Cited on pages 24, 37, and 49.)

[25] Gerwin Klein and Tobias Nipkow. A Machine-Checked Model for a Java-Like Language, Virtual Machine and Compiler. *TOPLAS*, 28(4):619–695, July 2006. (Cited on page 41.)

[26] Viswanathan Kodaganallur. Incorporating Language Processing into Java Applications: A JavaCC Tutorial. *IEEE Software*, 21(4):70–77, 2004. (Cited on page 131.)

[27] Daniel K. Lee, Karl Crary, and Robert Harper. Towards a Mechanized Metatheory of Standard ML. In *Proceedings of POPL*, volume 42(1) of *ACM SIGPLAN Notices*, pages 173–184. ACM Press, January 2007. (Cited on page 41.)

[28] Xavier Leroy. Java bytecode verification: algorithms and formalizations. *Journal of Automated Reasoning*, 30(3–4):235–269, 2003. (Cited on page 24.)

[29] Sheng Liang and Gilad Bracha. Dynamic Class Loading in the Java Virtual Machine. In *Proceedings of OOPSLA*, pages 36–44, October 1998. (Cited on page 28.)

[30] David MacQueen. Modules for Standard ML. In *Proceedings of LFP*, pages 198–207. ACM Press, August 1984. (Cited on pages 24 and 144.)

[31] Jacob Matthews and Robert Bruce Findler. An operational semantics for R^5RS Scheme. In *Proceedings of SFP*, ACM SIGPLAN Notices, pages 41–54. ACM Press, September 2005. (Cited on pages 41 and 50.)

[32] Sean McDirmid, Matthew Flatt, and Wilson Hsieh. Jiazzi: New Age Components for Old Fashioned Java. In *Proceedings of OOPSLA*, volume 36(11) of *ACM SIGPLAN Notices*, pages 211–222. ACM Press, November 2001. (Cited on page 50.)

[33] *$C^{\#}$ Specification*. Microsoft, 2.0 edition, September 2005. (Cited on pages 24 and 41.)

[34] Robin Milner, Mads Tofte, and Robert Harper. *The definition of Standard ML*. MIT Press, 1990. (Cited on pages 24 and 41.)

[35] Tobias Nipkow and David von Oheimb. $Java_{\ell ight}$ is Type-Safe — Definitely. In *Proceedings of POPL*, ACM SIGPLAN Notices, pages 161–170. ACM Press, January 1998. (Cited on page 41.)

[36] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, June 2008. (Cited on page 63.)

[37] Michael Norrish. *C Formalized in HOL*. PhD thesis, University of Cambridge, 1998. (Cited on page 41.)

[38] *About the OSGi Service Platform*. OSGi™ Alliance, 4.1 edition, November 2005. (Cited on page 42.)

[39] Scott Owens. A Sound Semantics for OCaml$_{light}$. In *Proceedings of ESOP*, volume 4960 of *LNCS*, pages 1–15. Springer, March 2008. (Cited on page 41.)

[40] David L. Parnas. On the Criteria to be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12), December 1972. (Cited on page 23.)

[41] Simon Peyton Jones. Special Issue: Haskell 98 Language and Libraries. *Journal of Functional Programming*, 13, January 2003. (Cited on page 24.)

[42] The Apache Jakarta Project. Byte Code Engineering Library, June 2006. Version 5.2. (Cited on page 137.)

[43] John C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proceedings of LICS*, pages 55–74. IEEE Computer Society, July 2002. (Cited on page 57.)

[44] Dennis Ritchie. The Development of the C Language. In *HOPL Preprints*, pages 201–208, 1993. (Cited on page 24.)

[45] Peter Sewell, James J. Leifer, Keith Wansbrough, Francesco Zappa Nardelli, Mair Allen-Williams, Pierre Habouzit, and Viktor Vafeiadis. Acute: High-level programming language design for distributed computation. In *Proceedings of ICFP*, volume 40(9) of *ACM SIGPLAN Notices*, pages 15–26. ACM Press, September 2005. (Cited on page 147.)

[46] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strniša. Ott: Effective Tool Support for the Working Semanticist. In *Proceedings of ICFP*, volume 42(9) of *ACM SIGPLAN Notices*, pages 1–12. ACM Press, October 2007. (Cited on pages 34, 35, 37, 38, and 58.)

[47] TIOBE Software. TIOBE Programming Community Index. http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html, March 2009. (Cited on page 25.)

[48] Rok Strniša. Fixing the Java Module System, in Theory and in Practice. In *Proceedings of FTfJP*, pages 88–99, July 2008. (Cited on page 37.)

[49] Rok Strniša and Matthew J. Parkinson. Lightweight Java (LJ). http://www.cl.cam.ac.uk/research/pls/javasem/lj/, September 2006. (Cited on page 57.)

[50] Rok Strniša, Peter Sewell, and Matthew J. Parkinson. The Java Module System: Core Design and Semantic Definition. In *Proceedings of OOPSLA*, volume 42(10) of *ACM SIGPLAN Notices*, pages 499–514. ACM Press, October 2007. (Cited on page 37.)

[51] Sun Microsystems, Inc. JSR-277: Java™ Module System. http://jcp.org/en/jsr/detail?id=277, October 2006. Early Draft. (Cited on pages 25, 35, 85, 86, 89, 90, 109, and 118.)

[52] Sun Microsystems, Inc. JSR-294: Improved Modularity Support in the Java™ Programming Language. http://jcp.org/en/jsr/detail?id=294, 2007. (Cited on pages 25, 35, 86, and 109.)

[53] Sun Microsystems, Inc. Java™ SE 7. https://jdk7.dev.java.net/, 2009. In development. (Cited on page 25.)

[54] Sun Microsystems, Inc. OpenJDK: Modules project. http://openjdk.java.net/projects/modules/, 2009. (Cited on page 25.)

[55] Don Syme. Proving Java Type Soundness. In *Formal Syntax and Semantics of Java*, pages 83–118. Springer, 1999. (Cited on page 41.)

[56] Macin Zalewski. *Generic Programming with Concepts*. PhD thesis, Chalmers University of Technology, November 2008. (Cited on page 41.)

Index