The Java Module System: Core Design and Semantic Definition

Rok Strniša Peter Sewell Matthew Parkinson
Computer Laboratory, University of Cambridge
{Rok.Strnisa,Peter.Sewell,Matthew.Parkinson}@cl.cam.ac.uk

Abstract

Java has no module system. Its *packages* only subdivide the class namespace, allowing only a very limited form of component-level information hiding and reuse. Two Java Community Processes have started addressing this problem: one describes the runtime system and has reached an early draft stage, while the other considers the developer's view and only has a straw-man proposal. Both are natural language documents, which inevitably contain ambiguities.

In this work we design and formalize a core module system for Java. Where the JCP documents are complete, we follow them closely; elsewhere we make reasonable choices. We define the syntax, the type system, and the operational semantics of an *LJAM* language, defining these rigorously in the Isabelle/HOL automated proof assistant. Using this formalization, we identify various issues with the module system. We highlight the underlying design decisions, and discuss several alternatives and their benefits. Our Isabelle/HOL definitions should provide a basis for further consideration of the design alternatives, for reference implementations, and for proofs of soundness.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Design, Languages, Theory

Keywords Java, JAM, LJAM, module, superpackage

1. Introduction

Large applications are typically built in several parts, which are more tightly connected to each other than to the outside. Therefore, one would like to give these parts more access to each other than the outside world has. One might also want these parts reused at many points in the system without worrying about the synchronization of shared data.

OOPSLA'07, October 21–25, 2007, Montréal, Québec, Canada.

In the functional world, this problem is solved with modules, but popular OO-languages, such as Java [15] and C# [20], have no direct language support for the above scenario. In Java, you have two alternatives: either (1) to put all parts of an application in a single package, and make them all non-public, which results in one giant, unwieldy package; or (2) to put each part in its own package, but then you also have to make their interfaces public, globally visible [10].

To address this, two Java Community Processes are developing *Java Module System (JAM*[1]*)* for the next version of Java [28]. One of them, JSR-277 [31], describes the runtime. The other, JSR-294 [29], is intended to cover the developer's view and the access policies. Currently, it only has a straw-man proposal [11], consisting of a couple of examples. Moreover, the JSR-277 draft introduces many concepts and is implementation-oriented, making it is hard to grasp the essential ideas. For example, there are: hierarchical structures of repositories, which can change at runtime; module definitions that can be instantiated and linked to other instances in remote repositories; and lookup functions for module and class definitions with unexpected behaviours. Both JSR's and this paper are mostly prose; however, our paper is based on a fully-formalized, mathematical definition.

In this paper we present an integrated design for a Java Module System, *LJAM*, covering the core parts of both JSR-277 and JSR-294. For those aspects in the scope of the former, we follow the informal description closely; for the remainder, we try to capture what we believe is the intended semantics. Our design is expressed rigorously: we define the syntax, type system and operational semantics for LJAM, producing (from a single source) both human-readable typeset rules and machine-processed mathematics for the Isabelle/HOL proof assistant. More specifically, our contributions are:

- a lightweight introduction to the Java Module System;
- development of the JSR-294 material on the programmer's view and the access model (§3.2);
- discussion of the core of JSR-277 (§3.3, §3.5–7); and, most significantly,
- a rigorous formalization of the above (§4), available while the JAM design is ongoing.

[1] JAM comes from JAva Module system, and not JAva with Mixins [6].

2. Background

The Java Module System (JAM) [30] is aimed at bringing hierarchical, component-level information hiding to Java, while at the same time removing the problem of DLL/JAR hell. In this section, we introduce the problems that JAM is trying to solve, and give an overview of the proposed solutions.

2.1 Hierarchical Information Hiding

Suppose we want to run a servlet that provides calendar services on our web server in existing Java. We write the *WebCalendar* software unit, which depends on two third-party software units (*XSLT* and *ServletEngine*); both of these require an *XMLParser*: the first to create HTML from XML calendar data, and the second for its configuration files. Furthermore, *ServletEngine* uses a *Cache* software unit to avoid overhead.

Figure 1 shows the dependency relation between these software units, and the result of putting them into their own Java packages. Figure 2 provides the key for this and all other figures in this paper.

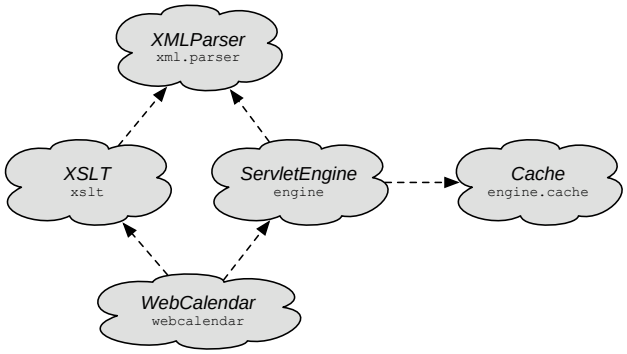

Figure 1. Software components in our example, and dependency among them

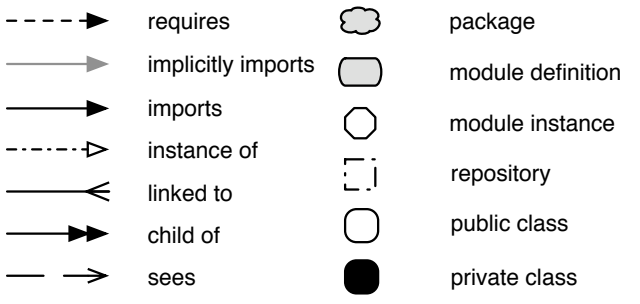

Figure 2. The key for all figures in this paper

At the moment, all classes in our system can see the public classes in `engine.cache`. Since *Cache* is (in our application) logically part of *ServletEngine*, we would like to make `engine.cache`'s public interface (its public classes) visible only to `engine`, i.e. to classes in package `engine`, and not also to the rest of the system.

A way of tackling this problem is to combine the contents of both `engine` and `engine.cache` into a single package, and make `engine.cache`'s public interface package-private instead; however, this approach not only loses structure, it also makes the package-private interfaces of the original packages visible to all classes in the new, combined package.

A better approach is to put both packages in a bigger structure, which acts as a black box, so that `webcalendar`, which is outside this structure, cannot see any of its contents. We refer to this kind of a bigger structure as a *module definition* (or *superpackage*; the terms are used interchangeably in the literature), and henceforth to the specific one holding both `engine` and `engine.cache` packages as `ServletEngine`.

To solve the information-hiding problem in a *hierarchical* way, we need to allow module definitions to define their own interfaces, i.e. to selectively leak parts of the interfaces of their contents. (If module definitions were simply black boxes, there would be no point in putting them in a bigger black box, since none of them could see each other's contents anyway.) For example, suppose there is a public class `engine.Servlet` that we would like to make visible to `webcalendar.CalendarServlet` (which is outside of `engine.Servlet`'s module definition `ServletEngine`). To achieve this, `ServletEngine` has to explicitly *export* `engine.Servlet`.

Since `webcalendar` is a standard Java package it is unaware of module definitions; therefore, we put it inside its own module definition `WebCalendar`, which then *imports* `ServletEngine`, making `engine.Servlet` visible to `webcalendar`. See Figure 3 for the software design, now with module definitions. Note that no module definition ever contains another[2]; this allows, for example, that both `XSLT` and `ServletEngine` import the same `XMLParser`.

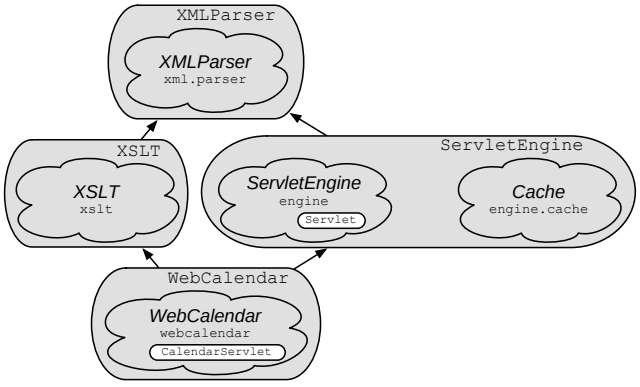

Figure 3. The example with module definitions

[2] The newest proposal of JSR-294 defines the concept of a *submodule definition*, which is a module definition that is *encapsulated* by another module definition. We do not formalize submodules.

2.2 DLL/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 filesystem, but it can also be loaded from somewhere else, or generated by reflection, and then possibly post-processed/checked before use [18]. A classloader keeps a map from fully-qualified names of classes[3] to their definitions, which means 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 static state. By default, classes are loaded by the system classloader.

JAR hell appears when two software components require different versions of a third component. 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 due to the above-mentioned restrictions. When a classloader is told to load a class whose name matches an already loaded class, it simply ignores the command. This results in undesirable behaviour either for *XSLT* or *ServletEngine*, depending on which version of the *XMLParser* is loaded first.

There are various solutions to this problem using complex systems of classloaders. The most common arguments against their use is that they are ad-hoc, not built into the language, and/or non-standardized (we discuss some of them later in §5). JAM allows same-named classes in the system as long as they are in different module definitions: Figure 4 shows this solution to the problem.

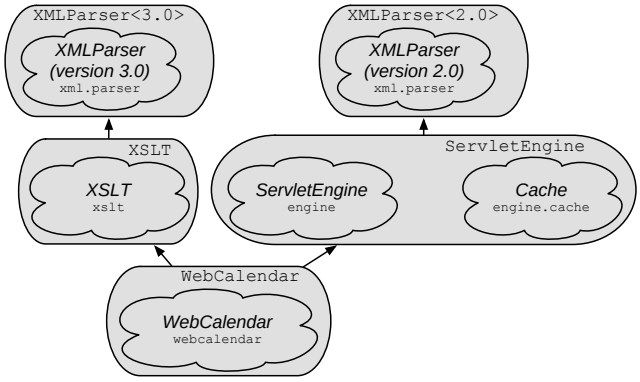

Figure 4. JAM's solution to DLL/JAR hell

3. The Design, Informally

This section presents the Java Module System and LJAM, informally. Subsections that are largely formalizations of the existing early draft of JSR-277 are annotated with "(JSR-277)"; those that should be covered by JSR-294, but are here designed by us, are annotated with "(JSR-294)". Subsections with no annotations contain our material not directly related to either of the two JSR's.

In each section, we identify all major design decisions made, either by the Java's EG-277 or EG-294[4], or by us, and then discuss reasons for them and possible alternatives at the end of the same subsection.

We begin with a fragment of Java, *Lightweight Java*, on which we base our design and formalization.

3.1 Lightweight Java with Standard Packages

Lightweight Java (LJ) is an imperative fragment of Java. It is intended to be as simple as possible while still retaining the feel of Java. LJ includes fields, methods, single inheritance, and dynamic method dispatch, but does not include any of the more advanced language features, e.g. field hiding, method overloading, interfaces, inner classes, or generics. A major difference w.r.t. Featherweight Java (FJ) [16] is that LJ also models state, whereas FJ is purely functional. In this sense, LJ is similar to ClassicJava [14], but, unlike ClassicJava, LJ is a proper subset of Java, i.e. every LJ program is a valid Java program. LJ is also formalized rigorously: using the Ott tool [25], we obtain the typeset rules *and* its formal definition (in Isabelle/HOL [21]) from a single point of truth — its Ott source files. All documents regarding LJ are available online [27].

Most of LJ's syntax is shown in Figure 5. Meta-variables dcl, f, var, $meth$ range over identifier strings,[5] whereas k is an index meta-variable. Overbars indicate lists, e.g. $\overline{s}$ is a list of statements, and $\overline{y_k}^{k \in 1..n}$ stands for $y_1 \ldots y_n$ (we omit $\in 1..n$ when the length is not important). The ctx in the object creation statement is a runtime annotation representing the context of the statement: it is always empty in LJ, but not in its extensions; it will be explained in more detail later. The rest is fairly standard.

By adding the concept of standard Java packages to LJ, we already obtain the core part of the syntax for our Lightweight Java Module System (LJAM): the syntactic changes and additions required to support Java packages are shown in Figure 6. Note that the fully-qualified name (of a class) now starts with pn, which is a meta-variable ranging over valid package names (which can include full stops, i.e. '.').

Now we have an appropriate base language to add the concepts described in §2.

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

[4] Expert Group (EG) is a group of individuals/companies, which contributes to particular Java Community Process (JCP).

[5] Our meta-variable naming is very specific. This is because Ott creates a meta-type for each meta-variable, which allows it to typecheck the semantic definitions: we only allow a meta-variable to appear at certain places in a construct. Therefore, we use short names, which closely correspond to standard notation wherever possible. For example, the meta-type of meta-variable dcl (name of *d*erived *cl*ass) is a subtype of the meta-type of meta-variable cl (name of *cl*ass; either dcl or $\mathbf{Object}$).

fqn	::=		fully qualified name
	|	dcl	def.
cl	::=		class name
	|	**Object**	top class
	|	fqn	fully qualified name
fd	::=		field declaration
	|	$cl\, f\,;$	def.
vd	::=		variable declaration
	|	$cl\, var$	def.
$x,\ y$	::=		term variable
	|	var	normal variable
	|	**this**	keyword
s	::=		statement
	|	$\{\, \overline{s_k}^{\,k} \,\}$	block
	|	$var = x\,;$	variable assignment
	|	$var = x\,.\,f\,;$	field read
	|	$x\,.\,f = y\,;$	field write
	|	$\textbf{if}\ (x == y)\ s\ \textbf{else}\ s'$	conditional branch
	|	$var = x\,.\,meth\,(\overline{y})\,;$	method call
	|	$var = \textbf{new}_{ctx}\, cl();$	object creation
$meth\_sig$	::=		method sig.
	|	$cl\, meth\,(\overline{vd})$	def.
$meth\_body$	::=		method body
	|	$s_1\,..\,s_k\ \textbf{return}\ y\,;$	def.
$meth\_def$	::=		method def.
	|	$meth\_sig\,\{\, meth\_body \,\}$	def.
cld	::=		class
	|	$\textbf{class}\ dcl\ \textbf{extends}\ cl\,\{\, \overline{fd}\,\overline{meth\_def} \,\}$	def.

Figure 5. Syntax of LJ

fqn	::=		fully qualified name
	|	$pn.dcl$	def.
am	::=		access modifier
	|		default
	|	**public**	public
pd	::=		package declaration
	|	$\textbf{package}\ pn\,;$	def.
cld	::=		class
	|	$pd\ am\ \textbf{class}\ dcl\ \textbf{extends}\ cl\,\{\, \overline{fd}\,\overline{meth\_def} \,\}$	def.

Figure 6. Syntax of LJ with Packages (modifications)

3.2 Module Files and New `public` (JSR-294)

The source file describing a module definition is known as a *module file*, with syntax as below.

$$mf ::= \textbf{super package}\ mn\,\{\, \overline{\textbf{export}\, fqn_j;}^{\,j}\ \overline{pn_k;}^{\,k}\ \overline{mn_l;}^{\,l} \,\}$$

A module file specifies the module definition's name mn, its *contents* as a list of package names $\overline{pn_k;}^{\,k}$, its *exports* as a list of fully-qualified class names $\overline{\textbf{export}\, fqn_j;}^{\,j}$, and its *imports* as a list of module definition names $\overline{mn_l;}^{\,l}$.

DESIGN DECISION 1 (Module Membership). *Module files specify their contents with a list of package names (not class names).*

The meaning of *import* here is similar to the standard meaning of `import` in Java: in both cases the visibility of the contents of the imported entity is controlled by the entity itself. The main difference, apart from operating on a different scale, is that module definitions can also control what is visible outside them, which makes hierarchical information hiding possible. We avoid the need for Java's import statements by specifying all class references with fully-qualified names. Note that JSR's are against changing the syntax of class source files in order to remain backward-compatible, which is why referring to individual modules from within the source is not possible.

```
package xml.parser; public class Parser {...}
package engine; class Config {...}
package engine; public class Servlet {...}
package engine.cache; public class Cache {...}
package webcalendar; public class
                          CalendarServlet {...}

super package XMLParser {
 export xml.parser.Parser;
 xml.parser;
}
super package ServletEngine {
 export engine.Servlet;
 engine; engine.cache;
 XMLParser;
}
super package WebCalendar {
 webcalendar;
 ServletEngine;
}
```

Figure 7. Access control example (code)

As outlined in §2, module definitions limit the scope of `public`. To give the intuition of how the new access policy works, we take the code in Figure 7 as an example: it shows the contents of five class *files* and three module *files*. Figure 8, shows the visibility between classes when the module files have been compiled into module definitions (a.k.a. superpackages).

From the figure, you can see that `xml.parser.Parser` is visible to all classes in `ServletEngine`, since it is public and exported by `XMLParser`. However, the class is not visible to `WebCalendar`, since it is not *re-exported* by `ServletEngine`. No class in `WebCalendar` can see `engine.cache.Cache` or `engine.Config` because they are, respectively, not exported or private.

Discussion of the Design Decisions

Module Membership (1)

Module files specify their contents with a list of package names (not class names).

This design choice has been made recently in JSR-294. It is the only reasonable choice to make here, since this

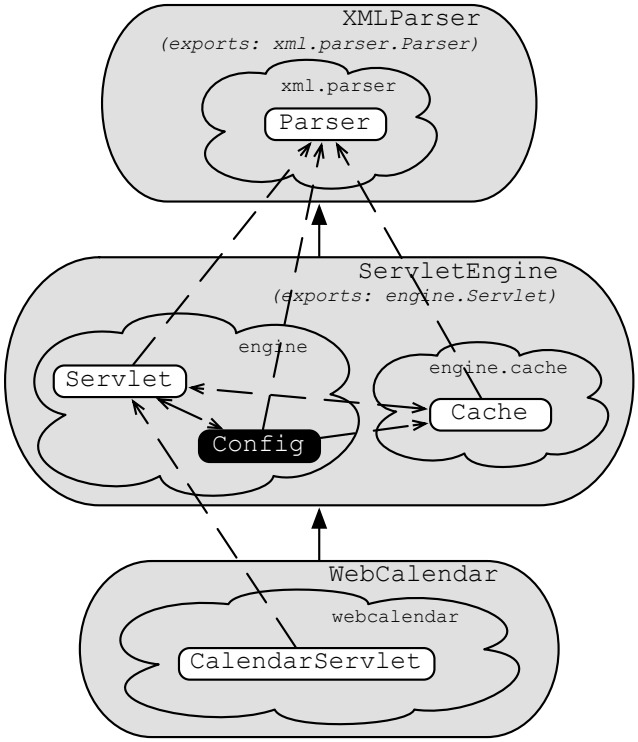

Figure 8. Access control example (visibility)

makes packages sub-members of module definitions, which in turn preserves the existing inter-package visibility semantics, i.e. the new access control will not change the visibility semantics within a single module definition.

Also, within a module definition, the previous access control semantics is preserved, e.g. `engine.cache.Cache` in Figure 8 cannot see package-private class `engine.Config`. If we had a single module definition that contained all the packages in the system, we would get Java-like semantics.

Note, however, that module membership is internally specified with individual classes [31, §3]. If one were to somehow ignore JSR-294's interface, one could have only a part of a Java package in a module definition. The inter-package visibility semantics would break, since a class reference that before resolved to a class in the same package could now fail completely, or resolve to a different class.

3.3 Modules and Repositories (JSR-277)

The compiled version of a module file is known as a *module definition*: the compilation replaces each package name in a module file with the corresponding class definitions. Module definitions must be installed in a *repository*, a *runtime* concept, for their classes to be usable. There can be many repositories at runtime to further control the dependency and isolation between different module definitions.

Repositories are organized into hierarchies with the **bootstrap repository** as its root [31, §7.1]. The **bootstrap repository** always contains the `core platform` module definition [31, §7.1], which in turn contains all the classes in the core SE platform [31, §2.17]; however, we do not formalize these classes explicitly. All modules definitions implicitly import the `core platform`; this is because of the way the class lookup function works (explained in §3.7).

For example, we might have the following repository-module structure in the runtime (see Figure 9):

- the runtime system contains an **XML parsers** repository, and two **Codebase** repositories;
- each of the **Codebase** repositories contains a `Main` module definition, which imports its own `Configuration`;
- both of the `Configuration` module definitions *import* the `XMLParser<3.0>` module definition from the **XML parsers** repository; and

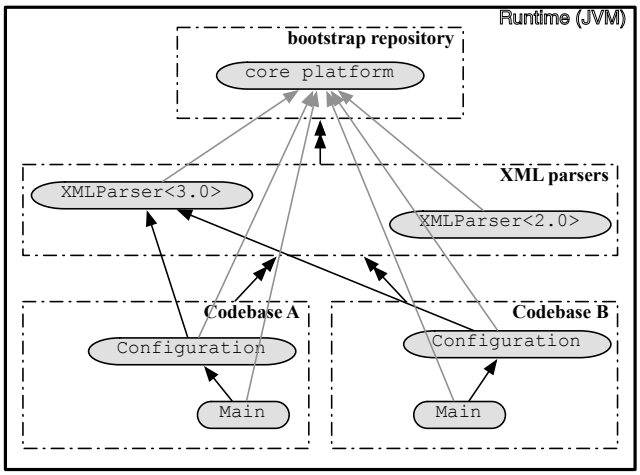

Figure 9. An example repository-module structure

By putting each codebase in its own repository, we made them almost completely isolated from each other — this is due to the way the module definition lookup function works (explained later in §3.6). Having a common repository parent (**XML parsers**), allows them to share types (and their static data) defined in the module definition `XMLParser<3.0>`.

We define the *abstract* syntax of module definitions and repositories, in Figure 10, which gives us a way of expressing and manipulating them in our rules. The structure of a module definition md consists of its name mn, its exports $\overline{fqn}$ (fully-qualified class names), its member class definitions $\overline{cld}$ (obtained from the package names $\overline{pn}$ of the corresponding module file), and its imports $\overline{mn}$ (module names).

A repository, R, is either a **bootstrap repository**, which cannot have a parent, or a *standard* repository, which always has a parent. We also distinguish between module definitions and *module instances*. The former specify modules' properties, i.e. their members, imports and exports, whereas the latter are executable, runtime instances of module definitions. As objects are runtime instances of classes, module instances

$$
\begin{array}{lcll}
md & ::= & & \text{module def.} \\
 & | & \mathbf{module}\ mn\ \{\ \overline{fqn}\ \overline{cld}\ \overline{mn}\ \} & \text{def.} \\
rn & ::= & & \text{repository name} \\
 & | & \mathbf{bootstrap\_r} & \text{bootstrap} \\
 & | & r & \text{standard} \\
R & ::= & & \text{repository} \\
 & | & \mathbf{bootstrap\ repository}\ \{\ \overline{md};\ \phi\ \} & \text{bootstrap} \\
 & | & \mathbf{repository}\ r\ \mathbf{child\ of}\ rn\ \{\ \overline{md};\ \phi\ \} & \text{standard}
\end{array}
$$

Figure 10. The *abstract* syntax for module definitions and repositories

are runtime instances of module definitions. Both types of repositories contain both module definitions $\overline{md}$ and their instances, the latter of which are stored in a module instance map ϕ.

The module instance map (ϕ) is a partial map, which maps module definitions md to module instance identifiers mi, i.e. $\phi : md \rightharpoonup mi$. It is, therefore, repository's heap for module instances of its module definitions.

Before any code in a module definition can be executed, all module definitions that the module definition (recursively) depends on must be initialized, and the module instances linked together. If we wanted to run the code in `Main` of **Codebase A** in Figure 9, `Main` has to be initialized. This triggers a recursive initialization procedure, which initializes `Configuration` (in **Codebase A**), `XMLParser<3.0>` (in **XML parsers**), and the `core platform` (in **bootstrap repository**). If we then initialize `Main` of **Codebase B**, it will trigger initialization of `Configuration` (in **Codebase B**), which will *not* initialize `XMLParser<3.0>` again, but will simply link against the existing instance.

Figure 11 shows the resulting runtime structure, and the actual location of module instances. Octagons represent module instances, dotted arrows point from module instances to their module definitions, and three-headed arrows represent the linking between the module instances. For clarity, we removed the lines indicating the implicit imports. Discussion of Design Decision 2 covers various alternatives to this picture.

DESIGN DECISION 2 (Creation of Module Instances). *Any repository can instantiate only a* single *module instance of each module definition installed in it.*

Discussion of the Design Decisions

Creation of Module Instances (2)

Any repository can instantiate only a single *module instance of each module definition installed in it.*

This design decision gives module instances a very limited usage. The only use they have in our formalization (and probably also in the upcoming system) is the fact that we can uninstall module definitions, and not need to worry about

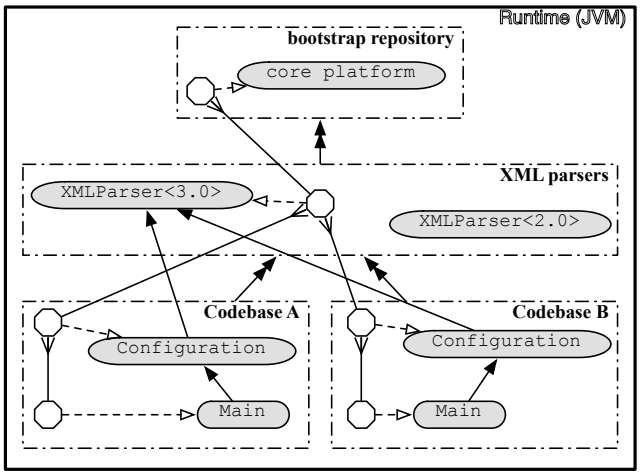

Figure 11. The example, now with module instances

currently executing code. One might want to create many module instances of a single module definition in order to replicate the static data and so avoid various concurrency problems; however, there can be only a single module instance of a module definition.

The only way to create two instances of a module definition is to install the same module definition in two different repositories, and instantiate both. However, because of the single-parent rule for repositories (see Figure 10), and because of the way module definition lookup function works (explained later in §3.6), it is not possible to refer to both of them from a single context.

Going back to the example in Figure 3, we have both `XSLT` and `ServletEngine` module definitions importing the same `XMLParser`. Suppose `XSLT` and `ServletEngine` can execute in parallel, and `XMLParser` is not thread-safe. The easiest solution would be to make two module instances of `XMLParser`, one for each user. Because there can be only one instance per repository, we place `XSLT` and `ServletEngine` in separate repositories, both of which also contain identical copies of `XMLParser`. The problem is that `WebCalendar` needs to use module definitions from both repositories, which is not possible due to the single-parent rule. Therefore, JAM does not have a solution for this *high-level separation* problem.

An alternative would be to allow each repository to create its own instance of any module definition it has access to, i.e. the module definition might originate from a different repository. Then each module instance could choose whether it wants to link against a remote instance, and share data (and types) with others, or a local instance, and have its own copy of module definition's state. Indeed, the early draft is not very clear about the creation of module instances, and we first thought that module instances were created only locally. We believe, however, that the above described approach can be simulated by installing the same module definition lo-

cally, and then using a *user-defined import policy*[6] to let a module instance choose which one of the two to link against.

A more object-oriented alternative would be to have a central registry for module instances, and allow any number of module instances per module definition in a single repository. Each module instance would then have to be put in the registry under a unique name (for that module definition), so that module instances could specify which ones they want to import. One can then also imagine a single instance importing multiple instances of the same module definition; for this, however, you would also need a way of disambiguating class references w.r.t. module instances, i.e. classes would have to be module-aware.

3.4 Representing the Runtime State

Apart from the usual state required for Java-like programs, we need a structure that holds the information about the network of repositories, and about the network of module instances. We store this information with two partial maps.

The first one, a *repository context* RC, maps repository names rn to repository structures R (defined in Figure 10), i.e. $RC : rn \rightharpoonup R$. It is used when looking up class and module definitions.

The second, a *module hierarchy* MH, maps module instance identifiers mi to *copies* of their corresponding module definitions md and module instances $\overline{mis}$, which mi is directly linked to through the import relation. It is *copies* of module definitions, and not links to them, since original definitions can be uninstalled. Therefore, $MH : mi \rightharpoonup (md * \overline{mis})$. The module hierarchy is used for resolving class references.

We refer to the pair of the two maps (RC, MH) as a *program* P.

3.5 Module Definition (Un)Installation/Initialization (JSR-277)

Module definitions can be installed and uninstalled from a repository. To use an installed module definition, we have to instantiate it, and link the instance in. We show these *administrator actions* in Figure 12.

a	$::=$		administator action
	$\mid$	$rn\,.\,\mathbf{install}\,(md)\,;$	install
	$\mid$	$rn\,.\,\mathbf{uninstall}\,(mn)\,;$	uninstall
	$\mid$	$\overline{init}$	initialize
$init$	$::=$		initialization
	$\mid$	$mi = rn\,.\,\mathbf{get\_instance}\,(mn)\,;$	def.

Figure 12. Administrator actions

Module installation inserts a given module definition into a specified repository. The only check that we perform during the installation is that module definition's name is unique in the repository.

[6] We do not formalize it in our system at present.

Module uninstallation simply removes the module definition from the specified repository, making subsequent searches for that module definition or its instance fail. Since module definitions are only searched at initialization time, and since module instances hold their own copies of the corresponding definitions, un-installing a module definition should not affect already executing code — in §4.5, we discuss this property in detail.

If a module definition has been initialized previously, then the initialization command simply returns the existing instance. Otherwise, we recursively initialize its imports, link an instance of the module definition with them, and update the program structures. Typechecking happens at initialization time, since this is the earliest that we know what a particular class reference resolves to. We discuss the details of typechecking in §4.4.

3.6 Module Definition Lookup (JSR-277)

Module definitions are looked up during module initialization in order to determine which module definition to initialize and link up with. Given the current repository context RC, the name rn_1 of the repository where we are starting the search, and the name mn of the module definition we are searching for, the module definition lookup function returns the corresponding module definition md, and the name rn_2 of the repository that has it installed.[7]

The search proceeds as follows: rn_1 first delegates the search to its parent repository recursively, only then searching itself; the result is the first match found. By definition, the **bootstrap repository** is at the top of the repository hierarchy.

Classloaders search for class definitions in the same manner in order to prevent library classes from getting overridden/hidden. Similarly, searching for module definitions in this fashion we prevent the `core platform` module definition from getting overridden/hidden.

3.7 Class Definition Lookup (JSR-277)

In order to execute an object creation statement $var = \mathbf{new}\ cl\,()$;, we need to resolve a class reference cl. For this, we need to know where the statement is located, i.e. its module instance mi and its standard Java package pn, which we refer to as statement's context ctx ($\equiv mi.pn$). We solve this problem in the semantics by annotating each object creation statement with its context — this *context insertion* happens at module initialization time, when we already know what module instance a certain statement belongs to. An implementation would carry the context around in the stack, but modelling a stack would unnecessarily complicate our semantics.

Since modules can import other modules, which can potentially be in a different repository, the class definition we

[7] It returns the owning repository as well, because in case of a recursive initialization that is the repository used to lookup module definitions for uninitialized imports.

are looking for can be in a different module (possibly in an ancestor repository) w.r.t. currently executing code. To search the runtime for class definitions, we need a representation of the runtime relation between modules, i.e. the module hierarchy MH introduced in §3.4.

Having access to the current repository structure RC and the current module hierarchy structure MH, and to the module instance mi we are currently executing in, we can resolve a class reference cl.

The search first checks the `core platform` module definition $\mathbf{core\_m}$, then all of the module instance's imports recursively, and finally the module instance itself, returning the first match found. Note that the order in which the imports are searched is determined by the order they are specified in the module file.

You can see an example class definition search in Figure 13, where the search is started at mi_1, which imports mi_2 and mi_3 (in this order), etc. The numbers below module instance names represents the order in which they are searched: note that this is *not* depth-first search, but rather a *reverse depth-first search*. For clarity, we do not show the repository structure or the corresponding module definitions.

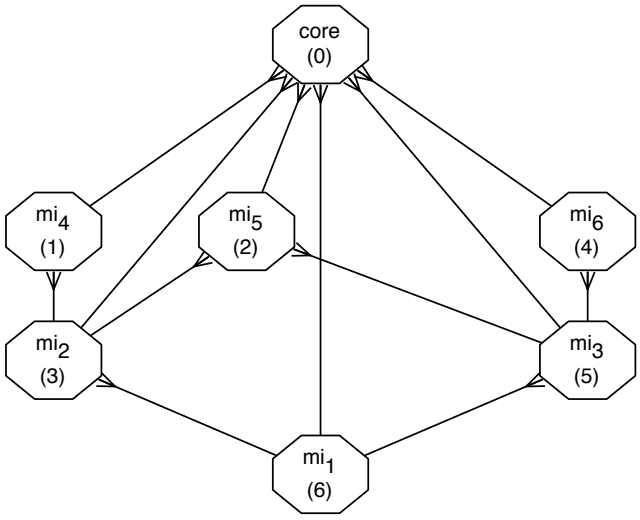

Figure 13. An example showing class search order

Each module instance will have a classloader associated with it, and each imported module instance will be represented with a delegate classloader. Therefore, the class search will be implemented through *multiple-siblings delegation* [31, §8.3.1].

DESIGN DECISION 3 (Class Search Order). *To find a class, look first in the imports, and only after in the module instance itself.*

There is a further restriction on the class names that is related to the class search order [31, §8.2.2.1]:

DESIGN DECISION 4 (Class Name Restriction). *The fully-qualified name of a class within a module instance must not clash with any of the fully-qualified names of the classes that are exported from the imported module instances.*

Discussion of the Design Decisions

Class Search Order (3)

To find a class, look first in the imports, and only after in the module instance itself.

Suppose we have a module definition `A`, which defines and exports a class `T`, and module definitions `B` and `C`, both of which import `A`, and define their own class `T`. `B` also defines a class `S`, which extends a class named `T`. Figure 14 shows the situation described (we do not show packages, assuming that all `T`'s and the `S` are fully-qualified class names). It is important to note here that even if the definitions of `T`'s are equal, their types are incompatible, because they are loaded by different classloaders (see §2.2 for clarification).

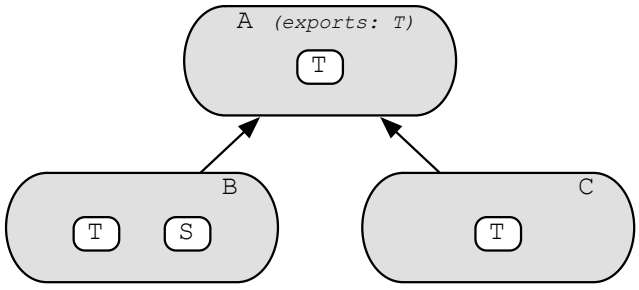

Figure 14. Type sharing example

Design Decision 3 implies that any reference to type `T` in any of the three module definitions will resolve to class `T` in module definition `A`. The argument for this is that if the module definitions were to somehow exchange objects of type `T` there would be no type clashes.

However, this also means that the `T`'s of both `B` and `C` are completely ignored, and there is no way the user can prevent that without changing `A`. The implementers of `B` and `C` probably had a good reason to include their own definition of `T` — for example, `C` could be a local patch of `A`. All the classes in `B` that depend on the included definition, e.g. `S` that extends `T`, might have their invariants broken, making the module definition behave in an unexpected manner.

This is similar to the issue of field-shadowing for classes, i.e. looking at Figure 14, we analogously have a *class* `A` with a public *field* `T`, and two *subclasses* `B` and `C`, which also define the *field* `T` (`B` also defines a *field* `S`).

Any reference in class `B` to field `S` will resolve to the field `S` in class `B`. If an update of class `A`, the superclass of `B`, was to add a field named `S`, any reference in `B` to field `S` should *still* resolve to `S` in `B`. This is why Java has field-shadowing. If we wanted references in `B` to field `S` to instead resolve to `S` in `A`, we would simply not define `S` in `B`.

Therefore, looking up classes in a *depth-first fashion* (self, then imports) rather than in a *reverse depth-first fashion* (imports, then self) would give more intuitive semantics. It would also give the developer of the module definition an option whether to use its own class, or the class defined in the imported module definition, and hence provide the ability to locally patch module definitions.

Class Name Restriction (4)

The fully-qualified name of a class within a module instance must not clash with any of the fully-qualified names of the classes that are exported from the imported module instances.

Shallow validation is an operation that guarantees the above property. It is performed by default during installation of a module definition, but can also be turned off. *Deep validation* (not performed by default) performs shallow validation and additionally validates the dependencies of classes in the module definition. We say validation to mean shallow validation.

Let's consider the example in Figure 14 again. If no validation happens, all references to type `T` refer to the definition in `A` — this is what we assumed in Design Decision 3. However, if validation *is* performed in our example, it fails, since there is a name-clash, and prevents execution.

If the developer is aware of the name-clash and wants to use the top-most definition of `T` (to allow for sharing between the module definitions), she has to disable the validation for `B` and `C` (validation can be disabled per module definition).

On the other hand, if the developer wants `S` to use the `T` defined in the same module definition (`B`), she can turn on the validation. This will tell her 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 `T` would still resolve to the top-most `T`.

Therefore, the only guarantee validation gives is that, if it succeeds, there will be no unexpected behaviour due to the unnatural class lookup function.[8]

4. Formalization of the System

Our informal discussion in the previous section is based on a fully rigorous formal definition of LJAM. As for LJ, this is produced using the Ott tool [25], generating a human-readable typeset version of the definition and an Isabelle/HOL formalization from a single source file. The Ott and Isabelle typechecking guarantees that the meta-types of the semantic definitions and rules are well-formed. The definition is available online [26]. It is a relatively large formal document, with around 140 semantic rules. This section discusses some of the most interesting aspects of the definition.

Note that our focus is upon simplicity and accuracy of the semantics, not runtime efficiency. While optimizations are possible, it is important that they preserve observable behaviour.

4.1 Judgements

Our formalization involves a number of different judgements — relations over the abstract syntax of the language (and auxiliary semantic entities) — defined by rules. These (62) judgement forms fall into five groups: execution of statements (3); execution of actions (7); typechecking and well-formedness (19); lookup functions (22); and various helper functions (11). In Figure 15 we show a sample of ten of these judgement forms.

The first one shows the variable translation: we do not formalize a stack, but rather flatten the method calls, which requires the variables in method bodies to be fresh — variable translation renames original names to fresh ones. In particular, variables in statement s are translated using the variable map θ; the result is statement s'.

The second is the main statement reduction judgement, with a configuration executing one step (for the next statement in the given configuration) to evolve into the subsequent state.

The third line shows the administrative action reduction judgement, for an administrator action a, again producing the subsequent state.

The next two are direct subtyping and transitive subtyping judgements. They have the usual meaning, i.e. τ is a direct (resp. normal) subtype of τ'.

Each of our structures has its own well-formedness judgement, checking that its contents are valid and correspond to its context. We show the judgements for well-formedness of a statement (in the context of its current program state and typing environment Γ, which holds the types of the containing method's parameters and $\mathbf{this}$), a method definition (in the context of the class type τ), and a module definition (corresponding to the copy being held by module instance mi).

Following are three lookup functions: $\mathbf{find\_type}$ returns the primary type (explained in §4.4) of the given class reference, starting the search in ctx; $\mathbf{find\_cld}$ finds the class definition (and where it originates from) for a fully-qualified name, starting the search in ctx; and $\mathbf{mtype}$ finds the primary method type for the given method name and the primary type of the owner.

4.2 Configuration and Statement Reduction

Statement reduction is basically straightforward. We define its semantics in small-step, operational style. To avoid the complexities of a stack, we freshly generate new variable names at every method call, renaming the method body's variables appropriately. We use a variable state ($L : x \rightharpoonup v$)

[8] JSR-277 states that names clashes should be avoided, and are prevented by validation; however, they are sometimes required as in the case of module definition patches or specializations shown in Design Decision 3. As name clashes are prevented by default, the specification of class search order in JSR-277 is in most cases redundant.

$\theta \vdash s \rightsquigarrow s'$	variable translation
$config \longrightarrow config'$	statement reduction
$config \xrightarrow{a} config'$	action reduction
$P \vdash \tau \prec \tau'$	subtyping relation
$P, \Gamma \vdash s$	well-formed statement
$P \vdash_\tau meth\_def$	well-formed method def.
$P \vdash_{mi} md$	well-formed module def.
$\mathbf{find\_type}(P,\ ctx,\ cl) = \tau$	get type for cl
$\mathbf{find\_cld}(P,\ ctx,\ fqn) = (ctx,\ cld)$	get class def. for fqn
$\mathbf{mtype}(P,\ \tau,\ meth) = \pi$	get $meth$'s type

Figure 15. Selected judgement forms

to store the value (**null** or reference oid) that a variable holds, and a heap ($H : oid \rightharpoonup (\tau * (f \rightharpoonup v))$) to store the type and values of fields (belonging to that type) that a reference points to.

Statement reduction is expressed as single steps from one valid configuration to another, where a configuration is a four-tuple of the program state P (described in §3.4), a variable state L, a heap H, and a list of statements yet to be executed $\overline{s_k}^{\,k}$, i.e. $config \equiv (P,\ L,\ H,\ \overline{s_k}^{\,k})$. Figure 16 shows rules for two of the more complicated statement reductions, i.e. reduction of object creations and method calls.[9]

4.3 Semantics of (Un)Installation/Initialization

The intuitive semantics of module definition (un)installation and initialization were explained in §3.5. We formally define the reduction of administrator actions in a similar manner to normal statement reduction: $config \xrightarrow{a} config'$, where a is an administrative action as in Figure 12. This says that the $config$ reduces to $config'$ if action a was successfully executed; if the action for some reason fails, the initial configuration is left unchanged. Our rules for these actions are shown in Figure 19. Note that the uninstallation action can only remove module definitions with name m, which by definition cannot refer to the `core platform` module definition.

We explain the initialization rule (R_NEW_INSTANCE) premise-by-premise as an example:

1. find the module definition in the program's runtime structure;
2. identify the repository that owns this module definition;
3. inspect the repository;
4. check that no instance exists for it already;
5. inspect the module definition;
6. (recursively) get instances for its imports;
7. create a fresh module instance;
8. annotate the module definition;
9. update the module hierarchy appropriately;
10. typecheck the annotated module definition in new context;
11. update the repository's module instance map;
12. update the repository context.

As you can see from step 10, typechecking of a module definition is done at initialization-time, but before any of the module definition's code can be executed. Typechecking must be done at initialization time, since only then do we know which class definitions do class references resolve to, i.e. they might resolve to class definitions present in a different module definition, possibly in a different repository.

The module definitions' import relation could potentially contain cycles, which should be detected by the module system to guarantee termination of the lookup functions [31, §8.3.4]. This is straightforward, but is not yet formalized.

4.4 Typechecking

Typechecking happens at initialization-time, as explained in §4.3. The typing and well-formedness rules are fairly standard — for an example, see Figure 17 for the rules that check for well-formedness of an object creation statement, a method call statement, and a method. For most language constructs, including statements, these rules are identical in LJ and LJAM. This is possible because, by and large, only the lookup functions differ. For example, for the rules in Figure 17, the definitions of $\mathbf{mtype}$ and $\mathbf{find\_type}$ differ in LJ and LJAM.

Types must uniquely identify the structure they refer to. To uniquely refer to a class definition in LJAM, it is enough to have a triple consisting of a module instance name mi, a package name pn, and a class name dcl; therefore, we use this triple to represent our type τ, i.e. $\tau \equiv mi.pn.dcl$. Since our context ctx is $mi.pn$ (see §3.7), and since our fully-qualified name fqn is $pn.dcl$ (see Figure 6), we also have the following equivalences: $\tau \equiv ctx.dcl \equiv mi.fqn$.

Note that by including the name of the module instance that the type belongs to in the type itself we automatically distinguish between the (otherwise same-named) types loaded by different classloaders — see the start of §2.2 and the end of §3.7 for more on this.

Two types referring to the same structure are not necessarily syntactically identical. Looking at Figure 13, suppose only mi_5 defines a class with name $pn.dcl$. Then both types $mi_2.pn.dcl$ and $mi_1.pn.dcl$ refer to $pn.dcl$ in md_5. In other words, the definition of types takes the class lookup function into account. We call $mi_5.pn.dcl$ the *primary type* of $pn.dcl$ (in md_5), since it exactly specifies where the class is located without having to search the runtime.

We cannot know whether a type is primary just by looking at it; however, in our formalization, all our lookup functions for types, e.g. $\mathbf{find\_type}$, always return primary types. All type references used in our subtyping judgements are

[9] All statement reduction rules are identical to those in LJ; the differences in semantics are hidden in the lookup functions, e.g. $\mathbf{find\_type}$.

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

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

Figure 16. Small-step operational semantics rules for two statements

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

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

Figure 17. Selected well-formedness rules

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

Figure 18. Selected subtyping rules

resolved with these lookup functions, so the judgements can assume they are dealing with primary types. The well-formedness of the object creation statements (the second rule in Figure 17) is an example how a type is looked up before it is compared to another.

The subtyping rules are fairly standard as well, except for the two direct subtyping rules shown in Figure 18. The lookup function used here, **find_cld**, takes a program state P, a context ctx ($\equiv mi.pn$), and a fully-qualified class name fqn ($\equiv pn.dcl$), and returns the appropriate class definition cld along with its context. Therefore, the first rule simply checks that the supertype of the given type is an **Object**, whereas the second rule also looks up the definition of the superclass, and checks that its type matches the primary type given. Other rules can be found in the full specification of LJAM [6].

Our typechecking is slightly different to that of Java, since we typecheck all classes within a module instance as soon as it has been initialized. In Java, however, most typechecking across module instances will happen during execution. Also, at the moment, our formalization looks up class definitions at runtime — we know those definitions will be found, since the typechecking already checked that they exist, and since the network of module instances doesn't

$\boxed{\mathit{config} \xrightarrow{a} \mathit{config}'}$ reduction of an administrative action

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

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

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

$$\frac{\begin{array}{l}1.\ \mathbf{find\_md}\,(RC,\ rn_1,\ mn) = (rn_2,\ md) \qquad 2.\ RC\,(rn_2) = R_2 \\ 3.\ \mathbf{R\_body}\,(R_2) = (\overline{md},\ \phi_2) \qquad 4.\ \phi_2\,(md) = \mathbf{null} \\ 5.\ md = \mathbf{module}\ mn\ \{\ \overline{fqn}\ \overline{cld}\ \overline{mn_k}^{\,k}\ \} \\ 6.\ ((RC,\ MH),\ L,\ H,\ \overline{s_l}^{\,l}) \xrightarrow{\overline{mi_k = rn_2\,.\,\mathbf{get\_instance}\,(mn_k)\,;}^{\,k}} ((RC',\ MH'),\ L,\ H,\ \overline{s_l}^{\,l}) \\ 7.\ mi \notin \mathbf{dom}\,(MH') \qquad 8.\ \vdash_{mi} md \rightsquigarrow md' \\ 9.\ MH'' = MH'\,[mi \mapsto (md',\ \overline{mi_k}^{\,k})] \\ 10.\ (RC',\ MH'') \vdash_{mi} md' \\ 11.\ \mathbf{R\_update}\,(R_2,\ \overline{md},\ \phi_2\,[md \mapsto mi]) = R_2' \\ 12.\ RC'' = RC'\,[rn_2 \mapsto R_2']\end{array}}{((RC,\ MH),\ L,\ H,\ \overline{s_l}^{\,l}) \xrightarrow{mi = rn_1\,.\,\mathbf{get\_instance}\,(mn)\,;} ((RC'',\ MH''),\ L,\ H,\ \overline{s_l}^{\,l})}\ \text{R\_NEW\_INSTANCE}$$

$$\frac{}{(P,\ L,\ H,\ \overline{s_l}^{\,l}) \to (P,\ L,\ H,\ \overline{s_l}^{\,l})}\ \text{R\_INITS\_EMPTY}$$

$$\frac{\begin{array}{l}1.\ (P,\ L,\ H,\ \overline{s_l}^{\,l}) \xrightarrow{mi = rn\,.\,\mathbf{get\_instance}\,(mn)\,;} (P',\ L,\ H,\ \overline{s_l}^{\,l}) \\ 2.\ (P',\ L,\ H,\ \overline{s_l}^{\,l}) \xrightarrow{init_2\,..\,init_k} (P'',\ L,\ H,\ \overline{s_l}^{\,l})\end{array}}{(P,\ L,\ H,\ \overline{s_l}^{\,l}) \xrightarrow{mi = rn\,.\,\mathbf{get\_instance}\,(mn)\,;\,init_2\,..\,init_k} (P'',\ L,\ H,\ \overline{s_l}^{\,l})}\ \text{R\_INITS\_CONS}$$

Figure 19. Module definition installation/uninstallation/initialization

change once fixed. Therefore, we could have cached class definitions found during typechecking, and used them during execution.

4.5 Lookup Functions and Concurrency

The execution of normal statements accesses and changes the heap H, the variable state L, and the list of statements $\overline{s_k}^k$ of the configuration. It does *not* change the program state P, however. In fact, only the class lookup function accesses the module hierarchy MH part of the program state, using the repository context RC only to locate the instance of the `core platform` module definition, which by definition cannot be removed.

The administrator actions, on the other hand, either add/remove module definitions, or create/fetch module instances, i.e. they change both parts of the program state, but only add new mappings to the module hierarchy — see step 9 in the module definition initialization rule. The new mappings cannot override old ones, because all module instances are created freshly, and the old mappings never change with execution once fixed by the module definition initialization step.[10]

From the above it follows that administrator actions do not interfere with normal execution, and can therefore be done in parallel. This is also the reason why we made the two reduction relations appear similar.

As long as intermediate states of structures that these actions operate on are invisible, or are read-only during action execution, actions can be done in parallel either on different repositories or on the same repository. There can be no circular dependency because of the hierarchical relation among repositories.

4.6 Modularization of the Semantics

We were able to reuse more than 80% of LJ's definition in the definition of LJAM. This was possible due to some software engineering techniques on the level of language design.

In LJ, we introduced an annotated version of the object creation statement: $var = \mathbf{new}_{ctx}\, cl();$. We also defined the statement reduction semantics without assumptions about the structure of the program state P or the current context ctx. The program state and current context are interpreted by lookup functions. Therefore, all the statement reduction rules remain identical; and only the lookup functions had to be adapted for different structures of program states and contexts. We applied the same principles for the typing and well-formedness rules. The benefits of this are demonstrated by the rule for well-formedness of a method in Figure 17.

The key concepts we abstracted away from to achieve this are shown in the following table:

Concept	Def. in LJ	Def. in LJAM
fully-qualified name fqn	dcl	$pn.dcl$
context ctx	(empty)	$mi.pn$
program state P	$\overline{cld}$	$(RC,\ MH)$

5. Related Work

Classloaders Currently the most common practice used to control the visibility and access policies of classes in Java is with specialized classloaders that delegate to each other in various patterns. These patterns can be quite complex, making the system's structure non-obvious and error-prone. A good example of the possible complexity is WebSphere's system of classloaders [23].

Open Source Gateway initiative OSGi [22] is a highly-customizable framework built on top of Java which promotes service-oriented programming, where services, specified in bundles (components), are registered into a global registry (Service Registry) with specific properties (Service description), which are used for service-lookup. Each bundle, a JAR file containing some metadata, classes, native code, and resources, is simulated with a classloader. The framework allows their installation, de/activation, update, and removal.

OSGi is at the moment the most widespread framework for a Java module system [2, 3, 1]. Building a module system into the language, however, will probably achieve an even larger user base, and hopefully also increase compatibility of module interfaces across software houses, although for backward-compatibility purposes JAM does allow the use of custom classloaders through the reflective API [31, §9.4].

Component systems Probably the most similar solutions to JAM are JavaMod [8] and Jiazzi [19]. They both have concepts of importing and exporting Java classes, where classes can link to each other across component boundaries. The user of a component has to define the interface of every imported class manually, but then gets separate compilation; compatibility between specified interfaces and actual classes is checked by the linker at initialization-time. Neither, however, has any notion of module instances or repositories, so state replication and location independence is not as transparent as in JAM.

MJ [12][11] is a module system built on top of Java, where each module corresponds to a classloader, and its (static) relationships to other modules are represented by import and export relationships amongst classloaders. Its access control mechanism is quite expressive, and so allows sophisticated relationships between different modules — see Figure 20.

MJ's compiler is used for module-aware compilation of classes, which basically checks the module constraints and sets the appropriate `CLASSPATH` for the standard Java compiler (`javac`). The module system only checks inter-module dependencies at runtime, even though many of them could be checked at installation/start time. There is no support

[10] In JAM, a module instance will be garbage collected once nothing refers to it any more.

[11] Not to be confused with MJ [9].

```
provides ‘‘catalina.jar’’;

import * from xerces;
import com.sun.tools.* from tools;

hide * in *;
export org.apache.catalina.* to webapp;
forbid org.apache.catalina.* in *;

module catalina {
  public static void load() {...}
  public static void main(String[] args) {...}
}
```

Figure 20. MJ's access control example

for versions of modules or classes. MJ does not work with custom classloaders; however, it is supposed to be easy to rewrite them as MJ modules.

By having a more expressive access control language such as MJ's, JAM could avoid a few of its problems regarding needless name clashes (described in §3.7).

There are a variety of theoretical proposals for rather different modularity features: first-class components [24] and mixin modules [7, 32].

Windows Fusion .NET assemblies [13] represent well-defined boundaries of security, namespaces, and versions, i.e. an assembly is the smallest versionable unit. In many respects, they are very similar to Java module definitions. However, .NET assemblies are still very much filesystem based, loaded into runtime by a loading system called Fusion [4]. Even though they are, in this respect, less expressive than JAM, .NET assemblies can have *strong names*, which are basically an author's public keys that can be checked for validity at load-time, providing an easy way to increase security of the system. This functionality can be simulated in JAM by putting public keys into *extensible metadata* [31, §2.5], and defining a *custom import policy* [31, §2.7.2] that checks them (neither of which we formalize).

6. Conclusion and Further Work

In this paper, we have designed and formalized a core module system for Java. We have defined the syntax, the type system, and the operational semantics of an *LJAM* language rigorously in the Isabelle/HOL automated proof assistant. With the help of our formalization, we have identified various issues with the module system, highlighted the underlying design decisions, and discussed various alternatives and their benefits. Two of the most significant issues we found are:

1. The creation of module instances is too limited. Put together with a single-parent repository structure, and we cannot solve a relatively simple software engineering problem, which we refer to as *high-level separation*: having access to more than a single instance of a particular module definition from a single point in runtime;
2. The current class definition lookup function has unintuitive behaviour, and gives the developer no control for disambiguating class name clashes across module definitions. We proposed an alternative, which gives a field-hiding-like behaviour, fixes the disambiguation problem, and allows module definitions to be locally patched.

The formalization is available online [26]. Our work is by no means the first to cover some aspect of Java formally — see, for example, that of Klein and Nipkow [17], and references therein. It is, however, very rare (for any language) to have a fully rigorous definition of a proposed language change available during the design process. We hope it will provide a useful basis for precise discussion.

The full Isabelle/HOL definition of our system is crucial step towards our future work: we aim (1) to obtain a reference implementation of our system, tightly conforming to the definition; and (2) to rigorously prove properties such as type preservation, progress, and non-interference of administrator actions with normal execution.

Currently, we are also extending this work to cover other features of the JAM, which includes versioning, user-defined import policy, native libraries, resources, and legacy JAR files.

As mentioned in §4.3, *all* typechecking is currently done at module definition initialization time. An alternative, and more efficient, approach would be to typecheck each module definition as much as possible in isolation at installation time (or even before), generate the linking conditions, and check those at linking time. The idea of *compositional compilation* [5] already solves a similar problem for class-level compilation, and should be investigated in the JAM setting.

Acknowledgments

We thank Jan Vitek, Doug Lea, Alex Buckley and Stanley Ho, for introducing us to the topic and for important discussion. We also thank Sophia Drossopoulou, Tom Ridge, Alisdair Wren, Nobuko Yoshida, Viktor Vafeiadis, John Billings, and Sam Staton for useful comments on earlier versions of this paper. We acknowledge the support of two EPSRC grants, DTA-RG44132 and GR/T11715/01, a Royal Society University Research Fellowship (Sewell), and a Royal Academy of Engineering/EPSRC Research Fellowship (Parkinson).

References

[1] Apache Felix. `http://cwiki.apache.org/felix/`.

[2] Eclipse Callisto. `http://www.eclipse.org/callisto/`.

[3] Equinox. `http://www.eclipse.org/equinox/`.

[4] Fusion. `http://en.wikipedia.org/wiki/.NET_assembly#Fusion`.

[5] ANCONA, D., DAMIANI, F., DROSSOPOULOU, S., AND ZUCCA, E. Polymorphic Bytecode: Compositional Compilation for Java-like Languages. In *Proc. 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'05* (Long Beach, CA, USA, Jan. 12-14, 2005), J. Palsberg and M. Abadi, Eds., ACM, pp. 26–37.

[6] ANCONA, D., LAGORIO, G., AND ZUCCA, E. Jam - a smooth extension of java with mixins. In *Proc. European Conference on Object-Oriented Programming, ECOOP'00* (Sophia Antipolis and Cannes, France, June 12-16, 2000), E. Bertino, Ed., vol. 1850 of *Lecture Notes in Computer Science*, Springer, pp. 154–178.

[7] ANCONA, D., LAGORIO, G., AND ZUCCA, E. Smart Modules for Java-like Languages. In *Proc. 7th Workshop on Formal Techniques for Java-like Programs, FTfJP'05* (Glasgow, Scotland, July 26, 2005).

[8] ANCONA, D., AND ZUCCA, E. True Modules for Java-like Languages. In *Proc. European Conference on Object-Oriented Programming, ECOOP'01* (Budapest, Hungary, June 18-22, 2001), J. L. Knudsen, Ed., vol. 2072 of *Lecture Notes in Computer Science*, Springer, pp. 354–380.

[9] BIERMAN, G., PARKINSON, M., AND PITTS, A. MJ: An imperative core calculus for Java and Java with Effects. Tech. Rep. 563, Cambridge University Computer Laboratory, Apr. 2003.

[10] BRACHA, G. Developing Modules for Development. `http://blogs.sun.com/gbracha/`, Mar. 2006.

[11] BRACHA, G. Superpackages: Development Modules in Dolphin. In *Proc. JavaOne℠ Conference* (2006), Sun Microsystems, Inc.

[12] CORWIN, J., BACON, D. F., GROVE, D., AND MURTHY, C. MJ: a rational module system for Java and its applications. In *Proc. ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA'03* (Anaheim, CA, USA, Oct. 26-30, 2003), R. Crocker and G. L. S. Jr., Eds., ACM, pp. 241–254.

[13] DEVELOPMENTOR. Assemblies Module - .NET: Building Applications and Components with C#, Jan. 2004.

[14] FLATT, M., KRISHNAMURTHI, S., AND FELLEISEN, M. A Programmer's reduction semantics for classes and mixins. Tech. Rep. TR-97-293, Rice University, 1997. Corrected June, 1999.

[15] GOSLING, J., JOY, B., STELLE, G., AND BRACHA, G. *The Java™ Language Specification*, Third ed. Sun Microsystems, Inc., May 2005.

[16] IGARASHI, A., PIERCE, B. C., AND WADLER, P. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst. 23*, 3 (2001), 396–450.

[17] KLEIN, G., AND NIPKOW, T. A Machine-Checked Model for a Java-Like Language, Virtual Machine and Compiler. *ACM Transactions on Programming Languages and Systems, TOPLAS'06 28*, 4 (July 2006), 619–695.

[18] LIANG, S., AND BRACHA, G. Dynamic Class Loading in the Java Virtual Machine. In *Proc. ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA'98* (Vancouver, British Columbia, Canada, Oct. 18-22, 1998), pp. 36–44.

[19] MCDIRMID, S., FLATT, M., AND HSIEH, W. Jiazzi: New Age Components for Old Fashioned Java. In *Proc. ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA'01* (Tampa, Florida, USA, Nov. 2001), vol. 36, pp. 211–222.

[20] MICROSOFT. *C# Specification*, 2.0 ed., Sept. 2005.

[21] NIPKOW, T., PAULSON, L. C., AND WENZEL, M. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, vol. 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[22] OSGI™ ALLIANCE. *About the OSGi Service Platform*, 4.1 ed., Nov. 2005.

[23] ROBINSON, R. Developing and Deploying Modular J2EE Applications with WebSphere Studio Application Developer and WebSphere Application Server. `http://www-128.ibm.com/developerworks/websphere/library/techarticles/0206_robinson/robinson.html`, June 2002.

[24] SECO, J. C., AND CAIRES, L. A Basic Model of Typed Components. In *Proc. European Conference on Object-Oriented Programming, ECOOP'00* (Sophia Antipolis and Cannes, France, June 12-16, 2000), E. Bertino, Ed., vol. 1850 of *Lecture Notes in Computer Science*, Springer, pp. 108–128.

[25] SEWELL, P., ZAPPA NARDELLI, F., OWENS, S., PESKINE, G., RIDGE, T., SARKAR, S., AND STRNIŠA, R. Ott: Effective Tool Support for the Working Semanticist. In *Proc. ICFP* (Freiburg, Germany, Oct. 2007). To appear in ICFP'07.

[26] STRNIŠA, R. Lightweight Java Module System. `http://www.cl.cam.ac.uk/~rs456/ljam`, Feb. 2007.

[27] STRNIŠA, R., AND PARKINSON, M. Lightweight Java. `http://www.cl.cam.ac.uk/~rs456/lj`, Sept. 2006.

[28] SUN MICROSYSTEMS, INC. Java™ SE 7. `https://jdk7.dev.java.net/`. In development.

[29] SUN MICROSYSTEMS, INC. JSR-294: Improved Modularity Support in the Java™ Programming Language. `http://jcp.org/en/jsr/detail?id=294`.

[30] SUN MICROSYSTEMS, INC. OpenJDK: Modules project. `http://openjdk.java.net/projects/modules/`.

[31] SUN MICROSYSTEMS, INC. JSR-277: Java™ Module System. `http://jcp.org/en/jsr/detail?id=277`, Oct. 2006. Early Draft.

[32] ZENGER, M. *Programming Language Abstractions for Extensible Software Components*. PhD thesis, University of Lausanne, EPFL, 2003.