Fixing the Java Module System, in Theory and in Practice

Rok Strniša

Computer Laboratory, University of Cambridge
Rok.Strnisa@cl.cam.ac.uk
http://www.cl.cam.ac.uk/~rs456/

Abstract. The proposed Java Module System (JAM) has two major deficiencies, as noted in our previous work: (a) its unintuitive class resolution can often give unexpected results; 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. Since JAM will be a part of Java 7, solving these problems before its release would benefit the majority of Java developers and users.

In this paper, 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 the improved module system, iJAM; (b) prove mechanized type soundness results; and, (c) give a proof-of-concept implementation that closely follows the formalisation; these are in Ott, Isabelle/HOL, and Java, respectively.

The formalisation is itself modular: iJAM is based on our previous formalization of JAM (LJAM), which extends Lightweight Java (LJ). LJ has shown to be 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.

1 Introduction

Currently, Java supports only a very limited form of component-level information hiding and reuse. Two Java Community Processes, JSR-277 [1] and JSR-294 [2], are developing the Java Module System (JAM), which will be a part of Java 7.

In our previous work [3], we analysed, partly developed, and fully formalized the core part of the proposed module system, producing the Lightweight Java Module System (LJAM) [4]. We also discussed the key design decisions related to the development of the core of JAM, and their alternatives, pointed out several deficiencies with module instantiation and class resolution, and gave tentative proposals for solving these problems. In particular, the two main problems are:

- JAM's unintuitive class resolution can easily lead to unexpected results.
- JAM has highly inflexible module instantiations that makes it difficult to reason about module invariants.

Our provisional ideas included reversing the class resolution algorithm, and allowing each repository to hold module instances of any visible module definition.

In this paper, we develop and precisely define clean solutions that make class resolution intuitive and flexible (through class renaming and an adapted resolution algorithm), and allow users to control the sharing of module instances (through user-specified policies). The solutions are modelled on top of LJAM, producing improved JAM (iJAM). We give a precise definition of iJAM's syntax, type system, and operational semantics. Furthermore, we prove in Isabelle/HOL [5] type soundness theorems for Lightweight Java (LJ) [6] (the language LJAM is based on), for LJAM and for 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. More specifically, our contributions are:

- precise and clean solutions to problems previously identified with JAM (§3);
- formalization of the solutions, producing iJAM (§4);
- Isabelle/HOL type soundness proofs for LJ, LJAM, and iJAM (§5); and
- an implementation that can model both LJAM and iJAM (except for sharing through renamed classes, which requires a small change to the JVM) (§7).

The details of our semantics, proofs, and implementation are available online [6, 4, 7].

2 A Short Overview of the Java Module System

The Java Module System (JAM) is a proposal for a module system integrated into Java, aimed at solving the hierarchical information hiding problem and the DLL/JAR-hell problem.

JAM introduces a few new concepts to Java, the most important of which is the *module*, or *superpackage*. A JAM module encapsulates Java packages, making even public *members* (classes or interfaces) of these packages by default invisible outside the module. Specific public members can be made visible by explicitly *exporting* them. A module can *import* other modules, which allows its members to see the (recursively) exported public members of the (recursively) imported modules.

JAM modules are specified by module developers in *module files*. The user syntax (the abstract syntax for what the user writes) of a module file is the following:

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

Here pn is a package name (which can include dots), m and mn are module names, and fqn is a fully-qualified class name. The overbars indicate lists.

A module file is compiled into a *module definition*, which contains class files for its members, and the information specified in the module file. A module definition can be installed into a *repository*, and then instantiated to create a *module instance* (intended to be implemented with a classloader [8]), which is linked up with instances of module definitions it imports. The directed graph of module instances is contained in a structure called the *module hierarchy* (MH).

A repository is used for storing, finding, (un-)installing, and instantiating module definitions. These *actions* are performed by system administrators and passively by the module system itself. Repositories can be composed into a *repository hierarchy* to further control the visibility between modules.

3 The Key Problems and Their Solutions

In this section, we analyse JAM's two key problems: unintuitive 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.

3.1 Unintuitive Class Resolution

The first key problem with JAM concerns its definition of class resolution. The procedure (recursively) searches the imported module definitions (following the order in the module file) before the client module. This is done in order to prevent anyone from overriding the core library classes,[1] and to promote the sharing of static data and types. In JAM, due to an oversimplified relation between modules, the importing module has no control over which exported classes of the imported modules are visible. The combination of the two properties lead to poor support for module interface evolution.

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`. Because the new *XMLParser* module is compatible with the old one, the developers only change its micro version number, which means that the modules that previously imported the old version will now likely import the new one automatically (due to commonly-used flexible version constraints). However, if an importing module, e.g. *XSLT*, already contained a class named "`ParserX`",[2] then any reference to a class `ParserX` in *XSLT* will now incorrectly resolve to `ParserX` in *XMLParser*.

Any changes to the underlying language are highly undesirable due to various compatibility issues. Because of this, we cannot introduce proper namespaces to the source language, which would allow class references such as `XMLParser::ParserX`.

In our previous work, we suggested inverting the class resolution algorithm. However, with that approach the user can override the core library classes. To prevent this, and to obtain a more intuitive semantics, the solution is to search the core library module, then the module itself, and finally the imported modules (recursively). In the exceptional cases where more classes should not be overriden, one can imagine requiring highest administrator privileges to put these classes into the core library module.

Continuing with the above scenario, suppose the developers of `XSLT` (which contains its own `ParserX`) now want to use the new functionality given by the new *XMLParser*'s `ParserX`. In JAM, this is not possible without name refactoring in the members of *XSLT*. To prevent such refactoring, which would likely disrupt the development in a large project, we could use the well-established solution of module-boundary renaming. That is, allow specific classes to be renamed locally as they are imported. We propose boundary renaming for JAM that will allow, for example, the developer of *XSLT* to use *XMLParser*'s `ParserX` under the name `ImportedParser` using:

import XMLParser **with** ParserX **as** ImportedParser ;

Similarly, one can disambiguate classes coming from different imported modules.

---

[1] The *core module* (Java's core library classes) is logically the root of the import graph.

[2] One might argue that due to the class naming conventions this scenario is unlikely, but one also has to realize that the naming conventions arose *because* of the lack of namespace control.

3.2 Inflexible Module Instantiation

In JAM, each module definition can only have a *single* module instance, i.e. JAM's module generators are severely restricted. This means that all clients necessarily have to share the module's static data and types, which is considered desirable, because it saves space, and prevents many possible class casting exceptions. However, suppose we have two module definitions, *XSLT* and *ServletEngine*, both of which depend on a third one, *XMLParser*. 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.

In JAM, 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), in order 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*.

However, the sharing of data or types through a common import is infrequent, especially across different programs. In all such cases, we can replicate the common import as required. This way the users can maintain conflicting invariants on separate instances of a 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. Here, we give an informal overview of the alternatives that allow for any possible sharing scenario:[3]

IMPORT OPTION	SHORT DESCRIPTION
import m	Uses JAM's default sharing policy. Can be overriden by **replicating** — see below.
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.[4]

However, there are cases where the developer of a module 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 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).

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

[4] If another module imports m as amn then they share the same instance. This option is here to cover the general case.

Collecting these alternatives, we have three different types of dependency between the importing and the imported superpackage: *shared*, *own*, and *as*. The following table summarizes the above-described interaction between different annotations: [5]

<table>
<tr><th rowspan="2" colspan="2"></th><th colspan="3">IMPORTED</th></tr>
<tr><th>default</th><th>replicating</th><th>singleton</th></tr>
<tr><th rowspan="4">IMPORTING</th><th>default</th><td>shared</td><td>own</td><td rowspan="4">shared</td></tr>
<tr><th>shared</th><td colspan="2">shared</td></tr>
<tr><th>own</th><td colspan="2">own</td></tr>
<tr><th>as</th><td colspan="2">as</td></tr>
</table>

The '**replicating**' flag says "use this module as shared at your own risk," whereas the '**singleton**' flag says "this module only makes sense if there is a single instance of it," hence we do not allow the importing module to override the latter. The above table shows how we put the intended semantics before safety in a concurrent environment.

4 Formalization

In this section, we describe the formalization of the above-mentioned solutions in improved JAM (iJAM), a modified version of our previously developed LJAM. As LJAM, iJAM, too, was formalized with Ott [9]. Due to the lack of space, we only briefly overview the interesting parts of the formalization. The full definitions (including all the semantic rules) can be found online [6, 4, 7].

The user syntax for iJAM's module files is

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

Comparing this to the user syntax of LJAM's module files shown in §2, the definition is now prefixed with $repl$, and $\overline{imp;}$ has replaced $\overline{\mathbf{import}\,m;}$. The definition of the two new entities (and the import dependency construct imp_dep) is shown in Fig. 1.

The combination of the two statically defined replication policies $repl$ and imp results in the actual replication policy imp_dep used at runtime — see the table in §3.2.

As in LJAM, the module instances are stored in repositories' caches. However, LJAM's caches simply mapped module definitions (md's) to their instances: $md \rightharpoonup mi$. In iJAM, the import dependencies need to be taken into account, so the cache type becomes: $md \rightharpoonup (imp_dep \rightharpoonup mi)$.[6]

The module hierarchy MH stores the connection between module instances. 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 the module hierarchy's structure 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.

The well-formedness relations were updated to reflect the new structures. The semantics of the administration actions, e.g. module initialization, were changed to account for different import dependencies and boundary renaming. Class resolution was updated as described in §3.1. Unfortunately, lack of space prevents any more detail here.

[5] This table might seem more complex than the relation it represents. It is not clear whether there is a simpler relation, which gives the same expressivity and convenience.

[6] Since module instances resulting from $\mathbf{Own}\, mi$ can only be linked against once, we could have excluded them from the cache.

$repl$	$::=$		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
	$\|$		no renaming
	$\|$	$\mathbf{with}\ fqn_1\ \mathbf{as}\ fqn'_1,\ ...,\ fqn_k\ \mathbf{as}\ fqn'_k$	renaming pairs
imp_dep	$::=$		import dependency
	$\|$	$\mathbf{Shared}$	default import
	$\|$	$\mathbf{Own}\ mi$	instance of imported module
	$\|$	$\mathbf{As}\ amn$	ref. to imported module

Fig. 1. Some of the entities introduced by iJAM

5 Mechanically Proving Type-Soundness

By providing Isabelle/HOL [5] homomorphisms for language productions to Ott, the tool is able to generate theorem prover definitions of the meta-types and semantic rules.

5.1 Type-Soundness Theorems

We prove type soundness by proving progress and type preservation properties. The progress property states that: if a configuration $(P, L, H, \overline{s})$ (where P is a program, L a variable mapping, H a heap, and $\overline{s}$ statements to execute) 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 ($\longrightarrow$).

Theorem 1 (Progress).

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

The type preservation property states that: if $config$ is a well-formed configuration in some type environment Γ, and $config$ reduces in one step to $config'$ through either statement reduction ($\longrightarrow$), or administrator action reduction ($\xrightarrow{a}$) in case of LJAM and iJAM, $config'$ is well-formed in some greater ($\subseteq_m$) type environment Γ'.

Theorem 2 (Type Preservation).

$$\begin{aligned} &\Gamma \vdash config \wedge (config \longrightarrow config' \vee config \xrightarrow{a} config') \\ &\quad \implies \exists \Gamma'.\ \Gamma \subseteq_m \Gamma' \ \wedge\ \Gamma' \vdash config' \end{aligned}$$

5.2 Our Experience

The semantic rules are defined as inductively-defined relations. Some of these relations, e.g. the class resolution, are easier to deal with in a theorem prover if expressed as functions. For those, we wrote Isabelle/HOL functions, and proved equivalence between the Ott-generated relation and the corresponding function.

When defining a function with non-primitive recursion in Isabelle/HOL, one also has to prove its termination. Especially in the case of `find_path_f`, the function that finds the inheritance path for a particular class, proving termination was a challenge. In our first attempt, we defined the relation/function so that it failed if 'the path found so far' was greater than the number of classes in the program. This worked fine until we had to prove well-formedness of a program with an extra module instance — the size of the program increased, so we could not prove semantic preservation for this function in general, since the function failed in one case, and not in the other. This forced us to define the acyclicity property among the class definitions, a property that is independent of the size of the program; however, we still had to prove the preservation of this property. Similarly, we had to define an acyclicity property among module instances to prove termination and well-formedness of `find_cld_in_imports_f`, a function finding class definitions among module imports.

All three formalizations also use a non-standard subtyping relation: a type τ is a subtype of τ' *iff* a class definition corresponding to τ is in the inheritance path of τ'. We then derive type reflexivity and transitivity, not vice versa. This definition makes it relatively easy to prove lemmas such as method type preservation.

One of the key lemmas in the LJAM's proof, which iJAM's new class resolution breaks, is `find_cld_same_ctx`. 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:

Lemma 1 (`find_cld_same_ctx`).

$$\begin{aligned}&\text{find_cld_f}\ P\ ctx\ fqn = \text{Some}\,(ctx', cld') \Longrightarrow\\&\text{find_cld_f}\ P\ ctx'\ fqn = \text{Some}\,(ctx', cld')\end{aligned}$$

In iJAM, this lemma did not hold any more, because fqn was not necessarily the same as the fully-qualified name of cld'. We modified the lemma by replacing fqn in the goal with $(\text{full_name_f}\ cld')$. However, the lemma was still false, 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. 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.

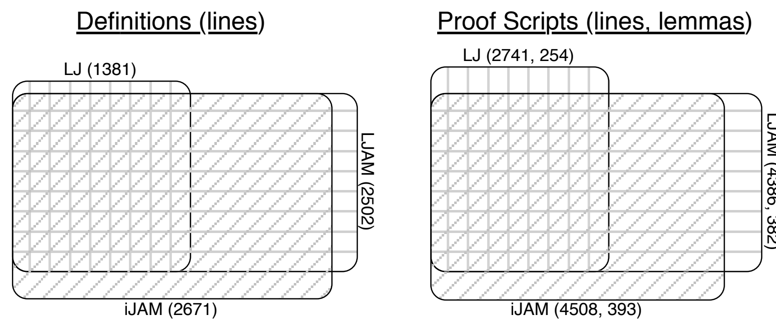
We solved this problem by placing a well-formedness condition on boundary renaming, which prevents module definitions from renaming a class *from* and *to* a name already exported by the core library module. Actually, already the *from* part makes the modified lemma hold again,[7] but we added the *to* part, too, in order to avoid unexpected class resolution results. These 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.

[7] It is still not clear to us whether iJAM is type-sound without this restriction.

6 Reuse in Definitions and Proof Scripts

Since LJAM is an extension of LJ, and iJAM is based on LJAM, they share much of the semantic definitions. In particular, 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. the lemma for the 'method type preservation' property (the lemma is too large to show here). Due to such abstractions, we were able to achieve high reuse in both the definitions and the proof scripts as shown by the following two diagrams (relative area corresponds to relative no. of lines of definition/proof script):



7 The Implementation

Our proof-of-concept implementation runs on top of Java; the module's code can contain any valid Java code. We implemented module files (with JavaCC [10]), repositories, module definitions, module instances, the module initialization mechanism according to iJAM's semantics, and a classloader, which respects iJAM's class resolution. The user can also enter compatibility mode, where the system behaves according to LJAM's semantics instead. We do not implement realistic compilation of module files, an administration console, or eager typechecking as specified by LJAM's and iJAM's semantics — typechecking is done lazily, as is normal in Java.

Each module instance needs to create distinct types, as well as replicate the static state of its classes. This can be achieved by having a classloader per one module instance. Each module object holds references to modules it imports, to which it delegates the class resolution as necessary. Because it is not possible to map two distinct names to the same `Class` object, sharing through renamed classes is not supported; however, we believe that only a small change to the JVM is required to allow this.

Using iJAM's user syntax presented in §4, we here give a simple example:

superpackage XMLParser {**member**; **export** Parser;}
superpackage XSLT {**member**; **import** XMLParser; **export** Config;}
superpackage ServletEngine {**member**; **import** XMLParser; **export** Config;}
superpackage WebCalendar {**member**; **import** XSLT;
import ServletEngine **with** Config **as** SEConfig;}

Both `Config` classes use the *XMLParser*::`Parser`, which tracks the number of its instances. *WebCalendar*::`Main` (not exported) uses *XSLT*::`Config` and `SEConfig` (*ServletEngine*::`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 *XMLParser* instance. One way to achieve multiple instances of the module is to tag it with **replicating**; in that case, the output shows `1. instance` in both cases. The full source is available online [7].

8 Related Work

In this section, we compare iJAM to the other module systems and related language features. In particular, we focus on class resolution, boundary renaming of classes, sharing of static data and types, and separate compilation.

In module systems based on the ML module system, the underlying language usually supports module-aware class references. Examples of such systems are MOBY [11] and CGEN [12]. The classes are looked up in the user module first, only then in the importing ones, unless the user refers to a specific imported module. These systems normally allow type renaming through external names of exported (visible) types.

Jiazzi [13] and ComponentJ [14] are two examples of module systems based on Units [15]. Although renaming is not supported, these systems provide powerful ways of combining components. Jiazzi gives module capabilities to packages, so package names in class references refer to imported modules — this simplifies the class resolution, but changes the semantics of the underlying Java language.

Bauer et al. describe a module system [16], which supports the renaming of both classes and modules, allows module-aware class references, and uses those to guide the class resolution — both of these changes, although useful, change the underlying language, which we are trying to avoid.

MJ [17], a module system similar to JAM, checks the module constraints and sets the appropriate `CLASSPATH` for the standard Java compiler. Its access control mechanism is quite expressive, allowing sophisticated relationships between different modules, such as selective importing and exporting, hiding, and sealing. Surprisingly, no renaming is supported. The problem of ambiguous class references seems to be ignored.

Of the above systems, only MOBY, Jiazzi and CGEN support separate compilation. MOBY achieves this with various restrictions to the language, whereas Jiazzi and CGEN require user-specified module interfaces. SMARTJAVAMOD [18] supports separate compilation through generation and verification of compositional constraints [19].

.NET assemblies [20] represent well-defined boundaries of security, namespaces, and versions. They are similar to JAM module definitions in many respects. Fusion, the

assembly binder for .NET, finds, loads, and binds assemblies before execution. Compilation of an assembly requires all of its imports to be present, so that the created bytecode can contain only explicit references — if there are many possible ways of resolving a class reference, an error is thrown at compile-time.

Currently the most widespread module framework for Java, OSGi [21] is a highly-customizable framework built on top of Java that promotes service-oriented programming. OSGi has a similar class resolution scheme to JAM, which we have argued here is counter-intuitive. Additionally to JAM, it, like MJ, supports *dynamic imports*, where an import dependency is resolved lazily at runtime — this increases the expressive power, but also lowers type safety guarantees, allowing type errors to happen also at runtime. As far as we are aware, it also has no module sharing control, boundary renaming, or namespace hiding, which means that it, too, suffers from the problems described in §3.

All of the systems mentioned so far (except OSGi) use a single copy of static data per application, i.e. modules are used as static libraries. In OSGi and JAM, modules are used as dynamic libraries, where multiple programs share types and static data.

Family polymorphism [22–24] is a flexible and transparent mechanism for code reuse. It gives classes some properties of modules, addresses hierarchical information hiding, and allows improved forms of inheritance. However, it does not address packaging, distribution, deployment, import renaming, or separate compilation.

9 Conclusions and Future Work

In this paper, we have presented and solved some of the most important problems with the JAM. We formalized the solutions in a language called iJAM, an extension of our previously developed LJAM. We defined the syntax, the type system, and the operational semantics of iJAM. In Isabelle/HOL, we proved type soundness for LJ, LJAM, and iJAM. Furthermore, we built a proof-of-concept implementation on top of Java, which can follow the semantics of either LJAM or iJAM. All the definitions, proofs, source code, documentation, and other documents can be found online [3, 4, 6, 7].

A good module system enables the user to restrict herself from making unintended dependencies. To prevent many unintended class name dependencies, JAM allows selective *exporting*. Our work substantially improves on this by providing support for boundary renaming (more expressive than selective *importing*). We also allow multiple module instances, which help with preventing unintended data/type dependencies.

The formalization 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. By mechanizing the meta-theory in Isabelle, it was easy to identify incorrectly formulated judgements and incomplete relations. Through abstractions in the definitions and the proof scripts, we achieved a high-level of reuse in both, which substantially sped up the process.

We plan on trying to use the recently developed Isabelle code generation tools [25] to generate reference implementations of the systems, which could then be compared to existing ones. Formalising OSGi is an interesting option for future work, which would allow detailed comparison of the two systems. This would likely help the industrial attempts to provide clean interoperability between the two systems.

Acknowledgments

We would like to thank Peter Sewell and Matthew Parkinson for their support, ideas, and useful comments on earlier versions of this paper. We acknowledge the support of EPSRC grants DTA-RG44132, EP/C510712, and GR/T11715/01.

References

1. Sun Microsystems, Inc.: JSR-277: Java™ Module System. `http://jcp.org/en/jsr/detail?id=277` (October 2006) Early Draft.
2. Sun Microsystems, Inc.: JSR-294: Improved Modularity Support in the Java™ Programming Language. `http://jcp.org/en/jsr/detail?id=294`
3. Strniša, R., Sewell, P., Parkinson, M.: The Java Module System: Core Design and Semantic Definition. In: Proc. of OOPSLA'07, ACM (October 21-25, 2007) 499–514
4. Strniša, R.: Lightweight Java Module System (LJAM). `http://www.cl.cam.ac.uk/~rs456/ljam/` (March 2007)
5. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic. Volume 2283 of LNCS. Springer (2002)
6. Strniša, R., Parkinson, M.: Lightweight Java (LJ). `http://www.cl.cam.ac.uk/~rs456/lj/` (September 2006)
7. Strniša, R.: improved Java Module System (iJAM). `http://www.cl.cam.ac.uk/~rs456/iJAM/` (November 2007)
8. Liang, S., Bracha, G.: Dynamic Class Loading in the Java Virtual Machine. In: Proc. of OOPSLA'98. (October 18-22, 1998) 36–44
9. Sewell, P., Zappa Nardelli, F., Owens, S., Peskine, G., Ridge, T., Sarkar, S., Strniša, R.: Ott: Effective Tool Support for the Working Semanticist. In: Proc. of ICFP'07, ACM (October 1-3, 2007) 1–12
10. Kodaganallur, V.: Incorporating Language Processing into Java Applications: A JavaCC Tutorial. IEEE Software **21**(4) (2004) 70–77
11. Fisher, K., Reppy, J.: The Design of a Class Mechanism for MOBY. In: Proc. of PLDI'99, New York, NY, USA, ACM (May 1-4, 1999) 37–49
12. Sasitorn, J., Cartwright, R.: Component NextGen: A Sound and Expressive Component Framework for Java. SIGPLAN Not. **42**(10) (2007) 153–170
13. McDirmid, S., Flatt, M., Hsieh, W.: Jiazzi: New Age Components for Old Fashioned Java. In: Proc. of OOPSLA'01. Volume 36. (October 14-18, 2001) 211–222
14. Seco, J.C., Caires, L.: A Basic Model of Typed Components. In Bertino, E., ed.: Proc. of ECOOP'00. Volume 1850 of LNCS., Springer (June 12-16, 2000) 108–128
15. Flatt, M., Felleisen, M.: Units: Cool Modules for HOT Languages. SIGPLAN Not. **33**(5) (1998) 236–248
16. Bauer, L., Appel, A.W., Felten, E.W.: Mechanisms for Secure Modular Programming in Java. Software—Practice and Experience **33**(5) (2003) 461–480
17. Corwin, J., Bacon, D.F., Grove, D., Murthy, C.: MJ: A Rational Module System for Java and its Applications. In Crocker, R., Jr., G.L.S., eds.: Proc. of OOPSLA'03, ACM (October 26-30, 2003) 241–254
18. Ancona, D., Lagorio, G., Zucca, E.: Smart Modules for Java-like Languages. In: Proc. of FTfJP'05. (July 26, 2005)
19. Ancona, D., Damiani, F., Drossopoulou, S., Zucca, E.: Polymorphic Bytecode: Compositional Compilation for Java-like Languages. In Palsberg, J., Abadi, M., eds.: Proc. of POPL'05, ACM (January 12-14, 2005) 26–37

20. DevelopMentor: Assemblies Module - .NET: Building Applications and Components with C# (January 2004)
21. OSGi™ Alliance: About the OSGi Service Platform. 4.1 edn. (November 2005)
22. Ernst, E.: gbeta – a Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance. PhD thesis, Department of Computer Science, University of Aarhus, Århus, Denmark (1999)
23. Odersky, M., Zenger, M.: Scalable Component Abstractions. In: Proc. of OOPSLA '05, New York, NY, USA, ACM (October 16-20, 2005) 41–57
24. Clarke, D., Drossopoulou, S., Noble, J., Wrigstad, T.: Tribe: A Simple Virtual Class Calculus. In: Proc. of AOSD'07. (March 12-16, 2007)
25. Haftmann, F.: Code generation from Isabelle/HOL theories. (November 2007)