Proving Java Type Soundness

Don Syme*
email: `drs1004@cl.cam.ac.uk`

June 17, 1997

Abstract

This technical report describes a machine checked proof of the type soundness of a subset of the Java language called Java$_S$. A formal semantics for this subset has been developed by Drossopoulou and Eisenbach, and they have sketched an outline of the type soundness proof. The formulation developed here complements their written semantics and proof by correcting and clarifying significant details; and it demonstrates the utility of formal, machine checking when exploring a large and detailed proof based on operational semantics. The development also serves as a case study in the application of 'declarative' proof techniques to a major property of an operational system.

Contents

*This work was carried out under the PhD. program at the Computer Laboratory, University of Cambridge, UK.

1 Introduction

This technical report describes a machine checked proof of the type soundness of a subset of the Java[1] language called Java$_S$. A formal semantics for this subset has been developed by Drossopoulou and Eisenbach, and they have sketched an outline of the type soundness proof [DE97b]. The formulation we develop here serves two roles: it complements their written semantics and proof by correcting and clarifying significant details; and it demonstrates the utility

[1]Java is a trademark of Sun Microsystems, Inc.

of formal, machine checking when exploring a large and detailed proof based on operational semantics[2].

This work contributes to three distinct fields of formal reasoning:

- It contributes to our understanding of Java from a formal perspective, acting as a highly detailed analysis of a significant property of the language, and providing proofs and corrections to existing proofs that are interesting in their own right.
- It contributes to the development of proof tools for formal methods by being a major case study in 'declarative' proof techniques.
- It contributes to the study of the detailed formalization of language theory, and in particular highlights some of the tools and methodology that can be applied to this task.

A familiarity with Drossopoulou and Eisenbach's work may be required to understand all the technical details in this report. However most of the report should be clear to readers with a simple understanding of operational semantics and formal specification.

1.1 Java

Java is a rapidly spreading programming language developed by Sun Microsystems that aims to support safe distributed programming. Its developers describe it is follows [GJS96]:

> "Java is a simple robust O-O platform independent multi-threaded dynamic general purpose programming environment. It's best for creating applets and applications for the Internet, intranets and any other complex, distributed network."

Java is notable for its attractiveness for the existing base of C/C++ programmers, while avoiding the features that make C/C++ unsafe (e.g. pointers); its portability (e.g. all implementations implement IEEE floating point numbers); its large standardized library; its memory management with garbage collection; its dynamic linking and its precise language definition. Even before the first complete language description was available, use of the language was extremely widespread and the rate of increase in usage is still steep.

The main language features of Java are primitive types (characters, integers, booleans, IEEE floats), strings, classes with inheritance, instance/class variables and methods, interfaces for class signatures, shadowing of instance variables, dynamic method binding, statically resolvable overloading of methods, exceptions, arrays, subtyping through arrays, dynamic type checking of casts and array assignments, class modifiers (**private, protected, public** etc.), final/abstract classes and methods, nested scopes, separate compilation, dynamic linking, extensible security management, constructors and finalizers.

The Java subset we consider here is that covered in version 2.01 of Drossopoulou and Eisenbach's paper. It includes primitive types, classes with inheritance, instance variables and instance methods, interfaces, shadowing of instance variables, dynamic method binding, statically resolvable overloading of methods, object creation, null pointers, arrays and a minimal treatment of exceptions.

The subset excludes initializers, constructors, finalizers, class variables and class methods, local variables, class modifiers, final/abstract classes and methods, **super**, strings, numeric promotions and widening, concurrency, the handling of exceptions, dynamic linking, packages and

[2]The latest version of the proofs and specifications described in this document are available on the World Wide Web at `http://www.cl.cam.ac.uk/users/drs1004/java-proofs.html`. This will be updated to reflect further work on the formalization.

separate compilation. An advantage of the approach to formalization we take in this report is that as new features of the language are treated it will be possible to incrementally adjust existing definitions and proofs.

1.2 What is Type Soundness for Java?

Several studies have uncovered flaws in the security of the Java system, including its type system, and have pointed out the need for a formal semantics to complement the existing language definition [GJS96]. A formal treatment of many important aspects of the language (e.g. dynamic linking) has yet to be performed.

Type soundness states that a welltyped Java program will not 'go wrong' at runtime, in the sense that it will never reach a state that violates conditions implied by the typing rules. To illustrate, one aspect of type soundness is captured in the following statement that is taken directly from the Java Language Specification [GJS96]:

> The type [of a variable or expression] limits the possible values that the variable can hold or the expression can produce at run time. If a run-time value is a reference that is not `null`, it refers to an object or array that has a class ... that will necessarily be compatible with the compile-time type.

Type soundness is a property of particular interest for the Java language, because the type system is the key mechanism used to ensure that Java bytecodes downloaded from untrusted sites cannot breach the integrity of the user's machine, while still being executed without expensive runtime checks.

In this report we are concerned with the Java language itself, rather than the Java Virtual Machine (JVM). The two are very closely related but the difference is non-trivial: for example there are JVM bytecodes that do not correspond to any Java text. Thus it remains a challenge to formalize and verify the corresponding type soundness property for the JVM. However, the type systems of Java and the JVM are closely related, and a comprehensive study of the former is a useful precursor to the study of the latter. Of course, even if an abstract model of Java and/or the JVM is verified, this does not guarantee the soundness of any particular implementation, just of the 'ideal' case.

How should we formulate type soundness? The first question to ask is what observable effects we would expect from a language with an unsound type system. In the case of C, we expect protection violations, often in the form of 'core dumps'. With Java, we tend to be concerned with breaches of the system security policy, e.g. the transmission of private data by some Java program. Ideally we would like to relate our notion of type soundness to the absence of such effects. However, in practice the first step toward doing this is to look *inside* the runtime mechanisms of the language, and to prove that a certain *type soundness invariant* is maintained during the execution of the machine. This is the approach taken by Drossopoulou and Eisenbach, which we also use in this report.

Thus, type soundness must initially be expressed in terms of the *inner* workings of the runtime model we develop for Java$_S$. It is beyond the scope of this work to demonstrate that this ensures that no security breaches occur. The precise formulation of type soundness we use is described in Section 6.1.

Our main focus in this work is on issues associated with using a computer to help reason about properties of programming languages. We demonstrate that with the right tools a mechanically checked formalization of a significant problem in language research can be achieved. Although our main aim is not to find errors, several significant errors in the formulation adopted by Drossopoulou and Eisenbach have been discovered. We outline this error in Section 8, and discuss the aspects of the tools and methodology that allowed its discovery. In addition to this error, we have found other interesting mistakes and omissions, and we will note these as we proceed. We have also independently rediscovered one fairly significant omission from the Java Language Specification, described in Section 8.

1.3 The Tool: DECLARE

The work described in this document exists as part of a larger project to help develop effective tools for tackling language research problems. Similar work has been attempted by researchers for other languages, including Syme and Van Inwegen's work on Standard ML [Sym93, Van93], Van Inwegen's work towards a proof of type soundness for Standard ML [Van97], Nipkow et al's proofs on Mini ML [NN96] and Norrish's semantics for C in HOL [Nor97]. Much has been learnt from these efforts, particularly with regard to representational issues and the utility of certain kinds of automated reasoning tools. In the past researchers have generally tackled these problems with tools designed for other purposes (e.g. HOL, which was designed for structured hardware verification). The time seems ripe to reassess what tools are really most appropriate to assist with reasoning about operational semantics.

The need for better tools can really only be appreciated if we consider the problems with tools already available, and the difficulty of the task at hand. The following factors all affect the utility of a tool when applied to tasks of the kind we are considering:

- What underlying logic is used, and what is its expressiveness?
- How much automated reasoning support is provided to avoid tedious reasoning?
- What language is used for specifications, and can specifications be written in a natural style?
- How are proofs expressed, and can arguments be formulated succinctly and naturally?
- What assistance is given the construction of proofs?
- How does the tool support the maintenance of specifications and proofs under incremental changes?
- Are the documents produced readable? Can they be validated by researchers unfamiliar with the tool?
- Does the tool support exploratory modes of work?

Existing tools each fall short in a number of categories, e.g. proofs developed with HOL and PVS are typically unreadable, and in many provers the automated support provided is weak. The importance of the readability and clarity of proofs and specification is an important feature for the work we do in this report. Existing tools consistently force the user into expressing proofs in a manner that is awkward, obscure and unmaintainable (the exception is where the automation of those tools is sufficient to solve the problems in question).

The tool we use is called DECLARE [Sym97], and has been developed by the author over the last year. An introduction to DECLARE is given in Appendix A.

$$\begin{tikzcd}
\mathrm{Java} \arrow[r, phantom, "\supset"] & \mathrm{Java}_S \arrow[r, squiggly, "_{compiles}"] \arrow[d] & \mathrm{Java}_A \arrow[r] \arrow[d] & \mathrm{Java}_R \times \mathit{state} \arrow[r, squiggly, "_{(\Gamma,p)}"] \arrow[d] & \mathrm{Java}_R \times \mathit{state} \arrow[d] \\
 & \mathit{type} \arrow[r, phantom, "="] & \mathit{type} \arrow[r, phantom, "="] & \mathit{type} \arrow[r, phantom, "\geq_{wdn}"] & \mathit{type}
\end{tikzcd}$$

Figure 1: Components of the Semantics and their Relationships

1.4 Outline

The remainder of this report is organised as follows. In Section 2 we outline the formal semantics of Java$_S$ that we will use as the basis for the rest of our work. This is based heavily on the semantics developed by Drossopoulou and Eisenbach but is reproduced here for completeness and because it contains important corrections. Section 3 outlines the six steps we need to take to actually complete this task, and describes the tool (called DECLARE) that we used to create and check a formal proof of the type soundness property for Java$_S$. In Sections 4 to 7 we describe these steps in detail. In Section 8 the two major errors discovered in Drossopoulou and Eisenbach's proof are described, as well as the independent rediscovery of an error in the Java language specification. Finally we summarize and discuss related work in Section 9.

2 The Semantics of Java$_S$

In this section we present an operational semantics for Java$_S$, based heavily on that developed by Drossopoulou and Eisenbach in version 2.01 of their paper [DE97b]. The specification we present is the result of several iterations through the waterfall model of formal development that we will outline in the next section. The description of the semantics will necessarily be brief in places: for more details consult [DE97b].

We define:

- A subset of Java containing the features listed in Section 1.1;
- A small-step term rewriting system to describe the dynamic execution of Java$_S$ programs;
- A type inference system to describe compile-time type checking.

A picture of the components of the semantics is shown in FIgure 1. Our main concern will be with the 'annotated' language Java$_A$ and the Java$_R$ terms of the 'runtime machine'. The main differences between our semantics for Java$_S$ and version 2.01 of Drossopoulou and Eisenbach's are:

- We correct minor mistakes, such as missing rules for null pointers, some definitions that were not well-founded (e.g. those for MSigs, FDecs and FDec), some typing mistakes and some misleading/ambiguous definitions (e.g. the definition of MethBody, and the incorrect assertion that any primitive type widens to the null type).
- We choose different representations for environments, based on tables (partial functions) rather than lists of declarations.
- We carefully differentiate between the Java$_S$ source language and 'runtime terms'. The latter are used to model execution and have subtly different typing rules.

- We adopt a suggestion by von Oheimb that well-formedness for environments be specified without reference to a declaration order.

- We allow the primitive class `Object` to have an arbitrary set of methods (Drossopoulou and Eisenbach restrict `Object` to have no methods).

- We do not use substitution during typing, as it turns out to be unnecessary given our representation of environments.

- At runtime we do not choose arbitrary new names for local variables when calling a procedure, but use a system of 'frames' of local variables that makes reasoning about substitution much easier (and is also closer to a real implementation based on stacks and offsets).

- We are careful to identify the types that judgments and relations operate over, as a clear picture of this is needed when writing the machine model.

- The modelling of multi-dimensional arrays in version 2.01 of Drossopoulou and Eisenbach's paper was not faithful to Java, where sub-array dimensions are not constant.

- Arrays in Java support methods supported by the class `Object`. We include this in our model (with a non-trivial consequences for the model). However our model of arrays is still incomplete, as in Java arrays support certain array-specific methods and fields, whereas in our treatment they do not.

- We are less stringent in our use of 'well-formedness' conditions for types, as it turns out that the widening and typing relations implicitly ensure well-formedness when required.

We are grateful to Drossopoulou and Eisenbach for the opportunity to discuss these points, and they have incorporated many suggestions into their latest version [DE97a].

2.1 Syntax of Java$_S$

Java$_S$ programs consist of a sequence of classes (see Figure 2). Each class has a name, a super-class, a set of super-interfaces, a sequence of field declarations and a sequence of method bodies. Fields have a name and a type. Methods have a return type (possibly void), a list of parameters with types, and a sequence of statements followed by an optional return expression. The statements considered are conditionals, assignments, blocks and expressions. Statements always type to `void`. Java distinguishes between variables (which are akin to C lvalues, and eventually correspond to locations in memory) and expressions. Variables can occur on the left of an assignment statement, expressions cannot. Expressions include primitive values, variable dereferencing, method calls and class and array allocation. Variables include identifiers, field lookup and array lookup. The primitive values are the obvious literals for the primitive types `bool`, `char`, `byte`, `short`, `int`, `long`, `float` and `double`. Component types are types that form the basis of arrays, i.e. class types, interface types and primitive types, and array types are component types raised to some number of dimensions (unlike C, array sizes are never part of Java array types). Reference types are classes, interfaces or arrays, and regular types are just primitive types or reference types. Expression types may also be void (unlike Standard ML, `void` is not a primitive type, e.g. an array of `voids` is not possible). Argument types and method types give signatures for argument lists and method declarations. The special reference type `nullT` is added to assign to `null` values in the source text.

prog	=	$class_1; \ldots; class_n;$	(programs)
class	=	C `extends` C_{sup} `implements` $\mathrm{I}_1, \ldots, \mathrm{I}_n$ {	(class declaration)
		$field_1; \ldots; field_n;$	
		$method_1; \ldots; method_m;$	
		}	
field	=	*type field-name*	(field declaration)
method	=	*expr-type* *method* (*type* $\mathrm{x}_1, \ldots,$ *type* x_n) {	(method declarations)
		$stmt_1; \ldots; stmt_m$	
		`return` *expr*?	
		}	
stmt	=	`if` *expr* `then` *expr* `else` *expr*	(conditionals)
	|	*var* := *expr*	(assignment)
	|	{ $stmt_1; \ldots; stmt_n;$ }	(blocks)
	|	*expr*	(evaluation)

Figure 2: Programs, Classes, Fields, Methods

var	=	*id*	(local variable)
	|	*expr*.field-name	(object field)
	|	*expr*[*expr*]	(array element)
expr	=	*prim*	(literal value)
	|	*var*	(dereferencing)
	|	*expr*.*method-name*(*expr*+)	(method call)
	|	`new` C	(object creation)
	|	`new` *comptype*[*expr*]+[]*	(array creation)

Figure 3: Expressions and Variables

primitive-type	=	`bool` | `char` | `short` | `int` |	
		`long` | `float` | `double`	
simple-reference-type	=	*class-name* | *interface-name*	
component-type	=	*simple-reference-type* | *primitive-type*	
array-type	=	*component-type*$[]^n$	$(n > 0)$
reference-type	=	*simple-reference-type* | *array-type* | `nullT`	
type	=	*primitive-type* | *reference-type*	
expr-type	=	*type* | `void`	
arg-type	=	list of *type*	
method-type	=	*arg-type* $\rightarrow$ *expr-type*	

Figure 4: Types

$$
\begin{array}{lcl}
\textit{class-env} & = & \textit{class-names} \overset{table}{\longmapsto} \textit{class-dec} \\
\textit{interface-env} & = & \textit{interface-names} \overset{table}{\longmapsto} \textit{interface-dec} \\
\textit{variable-env} & = & \textit{variable-names} \overset{table}{\longmapsto} \textit{type} \\
\textit{class-dec} & = & \langle \begin{array}[t]{ll} \text{super:} & \textit{class-name}, \\ \text{interfaces:} & \text{set of } \textit{interface-names}\ , \\ \text{fields:} & \textit{field-names} \overset{table}{\longmapsto} \textit{type}, \\ \text{methods:} & \textit{method-names} \times \textit{arg-types} \overset{table}{\longmapsto} \textit{expr-type}\rangle \end{array} \\
\textit{interface-dec} & = & \langle \begin{array}[t]{ll} \text{superinterfaces:} & \text{set of } \textit{interface-names}\ , \\ \text{methods:} & \textit{method-names} \times \textit{arg-types} \overset{table}{\longmapsto} \textit{type}\rangle \end{array}
\end{array}
$$

Figure 5: Type checking environments

2.2 The Static Semantics

The static semantics are more complex than a simple set of inference rules. The complicating factors are:

- Java allows the use of classes before they are defined. There are no restrictions on this, except that a non-circular class and interface hierarchy must result.
- Java implementations disambiguate key pieces of information at compile-time. Method calls may be statically overloaded (not to be confused with the object oriented late-binding mechanism), and fields may be hidden by superclasses. The resolution of field and method references is done at compile-time. Also, widening between primitive types (not covered here) is also decided at compile time, by inserting coercions.

To accommodate the first of these difficulties, Drossopoulou and Eisenbach define the notion of type-checking environment, extracted from entire programs, along with a well-formedness condition for these. An environment contains the class and interface hierarchies, and the subclass, subinterface and widening (subtyping) relations can be derived from it. Well-formedness excludes circular class and interface hierarchies, and imposes other constraints. After this, the rules for static typing and the compile-time disambiguation of constructs can be developed.

2.2.1 Type Checking Environments

Type checking environments contain several components (Figure 5). Always present are tables of class and interface declarations. These contain the type information extracted from the definition of these constructs: we write these as records, and omit record tag names when it is obvious from the context what is being referred to. When typechecking inside method bodies the environment also contains a table of variable declarations. We use Γ for a composite environment, Γ^V, Γ^C and Γ^I its respective components, and $\Gamma(x)$ for the lookup of x in the appropriate table. In general it should be understood that Γ only contains those component environments that are necessary for the construct at hand to make sense. We also use $x \in \Gamma$ to indicate that x is defined in the relevant table in Γ.

Component types, array types, reference types and regular types are said to be well-formed, written $\Gamma \vdash \textit{object} \diamond_{syntax-category}$, if all classes and interfaces are in scope. We do not give the details here, but note that the special class `Object` is always in scope.

2.2.2 Subclasses, Subinterfaces and Widening

Next we define the subclass ($\sqsubseteq_{class}$), subinterface ($\sqsubseteq_{int}$) and implements ($:_{imp}$) relations as shown below. All classes are a subclass of the special class `Object`, though we do not have to mention this explicitly as the well-formedness conditions for environments will ensure it.

$$\frac{\Gamma \vdash C \diamond_{class-name}}{\Gamma \vdash C \sqsubseteq_{class} C} \ (\mathit{reflC}) \qquad \frac{\Gamma(C).\text{super} = C_{sup} \quad \Gamma \vdash C_{sup} \sqsubseteq_{class} C'}{\Gamma \vdash C \sqsubseteq_{class} C'} \ (\mathit{stepC})$$

$$\frac{I \in \Gamma}{\Gamma \vdash I \sqsubseteq_{int} I} \ (\mathit{reflI}) \qquad \frac{I_k \in \Gamma(I).\text{interfaces} \quad \Gamma \vdash I_k \sqsubseteq_{int} I'}{\Gamma \vdash I \sqsubseteq_{int} I'} \ (\mathit{stepC})$$

$$\frac{I_k \in \Gamma(C).\text{interfaces}}{\Gamma \vdash C :_{imp} I_k} \ (\mathit{implements})$$

Subtyping in Java is the combination of the subclass, subinterface and implements relations, and is called *widening*. Defining widening accurately turns out to be a tedious but instructive process: we define it incrementally over the different kinds of types, i.e. over simple reference types ($\leq_{sref}$) then component types ($\leq_{comp}$) then array types ($\leq_{arr}$) and so on through to regular types ($\leq_{wdn}$). We have to be careful about this to avoid errors that creep in by other approaches: e.g. in Drossopoulou and Eisenbach's presentation it appears that all primitive types are narrower then `Object`, when in fact only reference types are. All reference types are subtypes of `Object` (classes by virtue of the subclassing rule) and **`nullT`** (the type given to null-pointers) is narrower than all reference types.

The full rules for widening are shown in Appendix B. Two important rules are those for arrays: the co-variant rule eventually leads to the need for runtime typechecking.

$$\frac{n > 0}{\Gamma \vdash ty\,\texttt{[]}^n \leq_{arr} \texttt{Object}} \ (\mathit{array\text{-}object}) \qquad \frac{n > 0 \quad \Gamma \vdash ty \leq_{comp} ty'}{\Gamma \vdash ty\,\texttt{[]}^n \leq_{arr} ty'\,\texttt{[]}^n} \ (\mathit{array})$$

2.2.3 Traversing the Class and Interface Hierarchies

The functions FDec, FDecs and MSigs traverse the subclass/subinterface graphs, starting at a particular class/interface, to collect:

- FDec: The 'first visible' definition of a field starting at a particular class. A set is returned, which will have one element for well-formed environments.
- FDecs: All the fields, including hidden ones, in the given class and its super-classes.
- MSigs: All the methods visible from a reference type. Methods with identical argument descriptors hide methods further up the hierarchy, though return types may be different. All `Object` methods are visible from all interfaces and arrays.

In Drossopoulou and Eisenbach's formulation these definitions are given as recursive functions. Their definitions only make sense for well-formed environments, as the search will not terminate for circular class and interface hierarchies. However the constructs are themselves used in the definition of well-formedness below. The functions are better formulated using inductively defined sets. The definitions of these rules are given in Appendix C.

MSigs is defined by first defining MSigs$_C$ MSigs$_I$ and MSigs$_A$ for the visible methods from the three different reference types. The methods visible from arrays and interfaces include all methods found in the type `Object`. Whether this should be the case for interfaces is the subject of discussion in Section 8.3.

2.2.4 Well-formedness for Type Checking Environments

We now turn to well-formedness for type checking environments. Drossopoulou and Eisenbach originally formulated this by an incremental process, where a sequence of definitions built up the entire environment, thus somewhat mimicking the process of separate compilation. We originally followed this formulation, but von Oheimb has pointed out that this is not necessary, since the definition is independent of any ordering constraints (though a finiteness constraint is needed).

To be well formed ($\vdash TE \diamond_{tyenv}$), every class declaration in an environment must satisfy the following constraints, based on those in [DE97b]:

- The class `Object` must be defined and have no superclass, superinterfaces or fields.
- Its superclass and implemented interfaces must be defined and no circularities can occur in the hierarchy;
- No two methods can have the same name and argument types (ensured by construction);
- Any methods that override inherited methods (by having the same name and argument types) must have a narrower return type;
- All interfaces must be implemented by methods that have narrower return types.

These are written as:

```
if $\Gamma(C) = \langle C_{sup}, Is, fields, methods \rangle$ then
        $\Gamma \vdash C_{sup} \diamond_{class-name}$
        $\neg(\Gamma \vdash C_{sup} \sqsubseteq_{class} C)$
        $\forall I \in Is.\ I \in \Gamma$
        $\forall meth, at, rt.$
            $methods(meth, at) = rt_1 \rightarrow$
            if $\Gamma \vdash ((meth, at), rt_2) \in \mathsf{MSigs}(C_{sup})$
            then $\Gamma \vdash rt_1 \leq_{wdn} rt_2$
  and   $\forall I \in Is.$
            if $\Gamma \vdash ((meth, at), rt_1) \in \mathsf{MSigs}(I)$
            then $\exists rt_2 . \Gamma \vdash ((meth, at), rt_2) \in \mathsf{MSigs}(C) \land \Gamma \vdash rt_2 \leq_{wdn} rt_1$
```

Likewise every interface declaration must satisfy the following conditions:

- All inherited interfaces must be defined and no circularities can occur in the hierarchy;
- No two methods can have the same name and argument types (ensured by construction);
- Any methods that override inherited methods must have a narrower return type;

These are written as:

```
if $\Gamma(I) = \langle Is, methods \rangle$ then
        $\forall I' \in Is.\ \ I' \in \Gamma$
        $\forall I' \in Is.\ \neg(\Gamma \vdash I' \sqsubseteq_{int} I)$
  and   $\forall I' \in Is, meth, at, rt_1.$
            $methods(meth, at) = rt_1 \rightarrow$
            if $\Gamma \vdash ((meth, at), rt_2) \in \mathsf{MSigs}(I')$
            then $\Gamma \vdash rt_1 \leq_{wdn} rt_2$
```

2.2.5 Static Typing and Compilation Rules

We can now define the static typing system for the input language. As mentioned before, information is disambiguated in the typing process. Thus, there are conceptually two languages and two type systems involved. No harm comes from conflating the two, though later we prove that the compilation process preserves types.

We do not give the full details of the typing rules here, since they follow the rules given by Drossopoulou and Eisenbach very closely. As an example, the typing rule for references to local (stack) variables in both the unannotated and annotated languages is:

$$\frac{VE(id) = type}{\Gamma \vdash id : type}$$

The typing rule for method calls in the unannotated language are:

$$\frac{\begin{array}{l}\Gamma \vdash obj : C \\ \forall i.\ 1 \leq i \leq n \rightarrow \Gamma \vdash arg_i : at_i \\ \Gamma \vdash \mathsf{MostSpec}(C, m, at) = \{(at' \rightarrow rt)\}\end{array}}{\Gamma \vdash obj.m(arg_1, \ldots, arg_n) : rt}$$

The definition of MostSpec can be found in [DE97b]: it determines the set of 'most special' applicable methods given the static types of the arguments. Only one such method should exist, otherwise a typing error occurs. Note that the resolution of methods based on static argument types means that unique typing is essential in Java.

Now we give the typing rule for the same construct in the annotated language. Here an exact method descriptor is given, so we simply check the argument types must match up to widening. The corresponding rule for annotated procedure calls is:

$$\frac{\begin{array}{l}\Gamma \vdash obj : C \\ \Gamma \vdash ((method, at), rt) \in \mathsf{MSigs}_C(C) \\ \mathsf{size}(at) = n \\ \forall i.\ 1 \leq i \leq n \rightarrow \exists ty.\ \Gamma \vdash arg_i : ty \wedge \Gamma \vdash ty \leq_{wdn} at_i\end{array}}{\Gamma \vdash obj.method[at](arg_1, \ldots, arg_n) : rt}$$

This completes our presentation of the static checks performed for the Java$_S$ language. We now move onto the runtime model of execution.

2.3 The Runtime Semantics

Drossopoulou and Eisenbach model execution as a small step rewrite system, i.e. a configuration represents the expressions yet to be evaluated and also the partial results of steps executed so far. This configuration is progressively modified by making reductions. The rewrite system is best thought of as specifying an abstract machine, and can be considered an inefficient but simple interpreter for Java$_S$.

A small-step system is chosen over a big-step since we later want to model non-determinism and concurrency, and we also want to reason about non-terminating programs. Unfortunately using a small-step system imposes significant overheads during the type soundness proof (e.g. with a big-step rewrite system no intermediary configurations need be considered), but this seems unavoidable.

$$\begin{array}{lcll}
\mathit{rval} & = & \mathit{prim} & \text{(primitive value)} \\
 & \mid & \mathit{addr\ option} & \text{(addresses} = \mathit{addr} \mid \texttt{null}\text{)} \\
 & \mid & \mathit{exn\text{-}packet} & (= \mathit{id}) \\
 & & & \\
\mathit{rvar} & = & \mathit{frame\text{-}num} \times \mathit{id} & \text{(local variable reference)} \\
 & \mid & \mathit{rexp}\texttt{.[}\mathrm{C}\texttt{]}\mathrm{F} & \text{(lookup } F \text{ at class } C\text{)} \\
 & \mid & \mathit{rexp}\texttt{[}\mathit{rexp}\texttt{]} & \text{(array access)} \\
 & & & \\
\mathit{rexp} & = & \mathit{rval} & \text{(simple value)} \\
 & \mid & \mathit{rvar} & \text{(dereference)} \\
 & \mid & \mathit{rexp}\texttt{.}\mathrm{M}\texttt{[}\mathit{at}\texttt{]}\texttt{(}\mathit{rexp}\texttt{+)} & \text{(method call)} \\
 & \mid & \texttt{new}\ \mathrm{C} & \text{(object creation)} \\
 & \mid & \texttt{new}\ \mathit{type}\texttt{[}\mathit{rexp}\texttt{]}+\texttt{[]*} & \text{(array creation)} \\
 & \mid & \{\mathit{rstmt}; \mathit{rexp}\} & \text{(statements with return expression)} \\
 & & & \\
\mathit{rstmt} & = & \texttt{if}\ \mathit{rexp}\ \texttt{then}\ \mathit{rexp}\ \texttt{else}\ \mathit{rexp} & \\
 & \mid & \mathit{rvar}\texttt{:=}\mathit{rexp} & \\
 & \mid & \{\mathit{rstmt}_1; \ldots; \mathit{rstmt}_n; \} & \\
 & \mid & \mathit{rexp} &
\end{array}$$

Figure 6: The syntax of rterms

2.3.1 Configurations and Runtime Terms

A *configuration* (s, t) of the runtime system is a *state* s and a *runtime term* (rterm) t. Runtime terms (of the language Java$_R$) contain artifacts not found in the source language, notably addresses, exception packets and the bodies of methods that have been called. There are three kinds of rterms: expressions, variables and statements, and thus there are really three different kinds of configurations. The top level configuration always contains an expression, since Java programs begin with the execution of a `main` static method from a given class (though how the machine gets started is not really important).

The syntax for rterms is shown in Figure 6. All terminating expressions eventually become values.

2.3.2 Program State

The program state consists of two components: a *list of frames* of local variables and a *heap* containing objects and arrays. The components have quite different properties and we will distinguish them from here on. Neither frames nor heap objects are garbage collected[3]. Heap objects are objects or arrays, and both are annotated with types (in the case of arrays this is the type of values stored in the array). We use the symbol $\oplus$ to indicate adding a new frame at the next available frame index, $s(\mathit{id})$ and $s(\mathit{addr})$ for looking up local variables and objects, and $s(\mathit{id}) \leftarrow \mathit{val}$ and $s(\mathit{addr}) \leftarrow \mathit{heap\text{-}obj}$ for assigning things into the respective components of the state.

[3]In future versions of the semantics a garbage collection rule that allows the collection of any inaccessible items at any time may be added. Garbage collection is semantically visible in Java because of the presence of 'finally' methods that get called before an object is deallocated.

$$\begin{array}{llll} state & = \langle & \text{frames: list of } (id \overset{table}{\longmapsto} val), & \\ & & \text{heap: } addr \overset{table}{\longmapsto} heap\text{-}object\rangle & \\ & & & \\ heap\text{-}object & = & \ll fld_1 \mapsto val_1, \ldots, fld_n \mapsto val_n \gg^C & (object) \\ & | & \texttt{[[}val_1, \ldots, val_n\texttt{]]}^{type} & (array) \end{array}$$

Figure 7: State

2.3.3 The rewrite system

The reduction of rterms is specified by three relations, one for each syntax category: $\overset{exp}{\leadsto}_{(\Gamma,p)}$, $\overset{var}{\leadsto}_{(\Gamma,p)}$ and $\overset{stmt}{\leadsto}_{(\Gamma,p)}$. Global parameters are an environment Γ (containing the class and interface hierarchies, needed for runtime typechecking) and the program p being executed.

A term is *ground* if it is in normal form, i.e. no further reduction can be made. Groundedness is actually a syntactic test that can depend on the syntax category from which a term is viewed. For example a local variable lookup *id* is ground if *id* is a variable, but not ground if it is an expression. This is because variables represent locations in memory, and when treated as expressions represent the values at those locations. Formally, groundedness is defined as follows:

- A value is ground *iff* it is a primitive value or an address.
- An expression is ground *iff* it is a ground value.
- A variable is ground *iff* all its component expressions are ground.
- A statement is ground *iff* it is an empty block of statements or a ground expression.

There are 36 rules in the rewrite system. 15 of them are "redex" rules that specify the reduction of expressions in the cases where sub-expressions have reductions. A sample is:

$$\frac{stmt_0, s_0 \ \overset{stmt}{\leadsto}_{(\Gamma,p)} \ stmt_1, s_1}{\{stmt_0; stmts\}, s_0 \ \overset{stmt}{\leadsto}_{(\Gamma,p)} \ \{stmt_1; stmts\}, s_1}$$

11 of the rules specify the generation of exceptions: 5 for null pointer dereferences, 4 for bad array index bounds, one for a bad size when creating a new array and one for runtime type checking when assigning to arrays. A simple example is:

$$\frac{\mathsf{ground}(exp) \quad \mathsf{ground}(val)}{\texttt{null[}exp\texttt{]} := val, s_0 \ \overset{stmt}{\leadsto}_{(\Gamma,p)} \ \texttt{NullPointExc}, s_0}$$

We cover the rules for array assignment and method call below. We omit the rules for field dereferencing, variable lookup, class creation, field assignment, local variable assignment and conditionals as they are straight forward and are covered in [DE97b]. As an example the array access rule for the case where the index is in-bounds is:

$$\frac{\begin{array}{l} s_0.\text{heap}(addr) = \texttt{[[}val_0 \ldots val_{n-1}\texttt{]]}^{type} \\ 0 \leq k < n \end{array}}{addr\texttt{[}k\texttt{]}, s_0 \ \overset{exp}{\leadsto}_{(\Gamma,p)} \ val_k, s_0}$$

The array creation rule is:

$$\frac{\begin{array}{l}\text{each } \mathit{dim}_i \text{ is some ground non-negative integer value } k_i \ (1 \leq i \leq n) \\ (\mathit{val}, \mathit{heap}_1) = \mathsf{alloc}(s_0.\text{heap}, \mathit{type}, \mathbf{k}, m)\end{array}}{\texttt{new}\ \mathit{type}\texttt{[}\mathit{dim}_1\texttt{]} \ldots \texttt{[}\mathit{dim}_n\texttt{]}\ \texttt{[]}^m, s_0 \overset{exp}{\leadsto}_{(\Gamma,p)} \mathit{val}, (s_0.\text{frames}, \mathit{heap}_1)}$$

Here alloc represents the process of allocating $k_1 \times \ldots k_{n-1}$ arrays containing pointers to arrays, and eventually pointing to arrays containing initial values appropriate for the type *type*. This process is described in detail in [GJS96][4].

2.3.4 Runtime typechecking

Java performs runtime typechecks at just two places: during array assignment, and when casting reference values. Runtime typechecking is needed for array assignment because of the well-known problem with a co-variant array typing rule. Casts are not covered in this report.

Runtime typechecking is performed by simply checking that the real (i.e. runtime) type of any reference object, as stored in the state, is narrower than the real type of the array cell it is being assigned to. This means the runtime system must have access to the program class/interfaces hierarchies (as the JVM does). An aside: Drossopoulou and Eisenbach's notion of runtime type checking (weak conformance) is a little too strong, as it allows the runtime machine to check the conformance of primitive values to primitive types: no realistic implementation of Java checks at runtime that a primitive type such as `int` fits in a given slot. We have not yet addressed this in our model, though we plan to in the near future.

The array assignment rules that utilise runtime type-checking are:

$$\frac{\begin{array}{l}\mathit{sval} \text{ is ground} \\ s_0.\text{heap}(\mathit{addr}) = \texttt{[[}\mathit{val}_0 \ldots \mathit{val}_{n-1}\texttt{]]}^{\mathit{type}} \\ 0 \leq k < n \\ \mathsf{typecheck}(\mathit{sval}, \mathit{type}) \text{ fails}\end{array}}{\texttt{addr[}k\texttt{] :=}\ \mathit{sval}, s_0 \overset{exp}{\leadsto}_{(\Gamma,p)} \texttt{ArrStoreExc}, s_0}$$

$$\frac{\begin{array}{l}\mathit{sval} \text{ is ground} \\ s_0.\text{heap}(\mathit{addr}) = \texttt{[[}\mathit{val}_0 \ldots \mathit{val}_{n-1}\texttt{]]}^{\mathit{type}} \\ 0 \leq k < n \\ \mathsf{typecheck}(\mathit{sval}, \mathit{type}) \text{ succeeds} \\ s_1 = s_0.\text{heap}(\mathit{addr}) \leftarrow \texttt{[[}\mathit{val}_1 \ldots \mathit{val}_{k-1}, \mathit{sval}, \mathit{val}_{k+1}, \ldots \mathit{val}_{n-1}\texttt{]]}^{\mathit{type}}\end{array}}{\texttt{addr[}k\texttt{] :=}\ \mathit{sval}, s_0 \overset{exp}{\leadsto}_{(\Gamma,p)} \texttt{void}, s_1}$$

The function typecheck checks that the stored type is compatible with the given type. It succeeds for an address *addr*, a type *ty* in a heap *h* if:

- $h(\mathit{addr}) = \ll \ldots \gg^C$ and *ty* is wider than C
- or $h(\mathit{addr}) = \texttt{[[...]]}^{ty'}$ and *ty* is wider than $ty'\texttt{[]}$

In future versions of the semantics this will not perform compatibility checks for primitive or `null` values.

[4]This model of array creation would need to be modified if threads or constructors are considered, as array creation is not atomic with respect to thread execution. It may also involve executing constructors (and thus may not even terminate), and may raise an out-of-memory exception.

2.3.5 Local Variables and Method Calls

In rterms references to local stack variables are annotated with a *frame number* that indicates which instance of the variable is being referred to (this is reminiscent of de Bruijn indices, though the lack of higher order features makes things simpler). This makes reasoning about substitution during method call easier.

Aside from the initial *rexp*, rterms are created only when a method is called. The body of the method is fetched from the program and translated to an rterm, while annotating local variables with a fresh frame number. The rule is:

$$\frac{\begin{array}{l}\text{each } arg_i \text{ is some ground value } v_i \ (1 \le i \le n)\\ s_0(addr) = \ll \ldots \gg^C\\ \mathsf{MethBody}(meth, at, C, p) = \lambda x_1 \ldots x_n.\ body\\ s_1 = s_0.\text{frames} \oplus \{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}\end{array}}{addr.meth\,\texttt{[}at\texttt{]}\,\texttt{(}arg_1, \ldots, arg_n\texttt{)}, s_0 \overset{exp}{\leadsto}_{(\Gamma,p)} \texttt{[}body\texttt{]}_{s_0}, s_1}$$

where $\texttt{[}body\texttt{]}_{s_0}$ is the process of translating input syntax *body* into an Java$_R$ term, annotating local variables with the next available frame number in s_0.

3 Six Steps to a Formalized, Machine Checked, Human Readable Proof of Java Type Soundness

In the following sections we present the process by which we develop a formal proof of the type soundness of Java$_S$. The methodology can be applied to any exploratory theorem proving exercise. The steps are as follows:

- Step 1: Understand the Problem
- Step 2: Develop a Machine Acceptable Model
- Step 3: Validate the Model by Generating an Interpreter
- Step 4: Formulate All Key Properties
- Step 5: Sketch an Outline of the Proof
- Step 6: Convince the Machine

The methodology is somewhat like the 'waterfall' methodology of software development: each step can require a return to previous steps, and we iterate until the task is complete. Some steps (e.g. validation) can be highly automated or skipped in later iterations.

Step 1 The first step is to develop a strong understanding of the problem. The tool we use, called DECLARE [Sym97], is designed to help users express proofs and specifications only when they already have a fairly good mental picture as to how the proof should proceed. The stronger the understanding that the user has, the more effective their use of the tool will be.

Step 2 In the second step, we use our rough understanding of the system in question to specify that system within higher order logic. Normally for any system that has previously only been specified on paper, this process uncovers many significant errors and simplifications, as well as unexpected complications that arise from the use of an insufficiently expressive logic in the mechanised tool. We specify the entire system before proceeding.

Step 3 The third step involves applying some techniques to check that the logical specifications we have written represent a valid model of the Java$_S$ language. Validation of such a specification is non-trivial, and is a topic that has been often ignored by the theorem proving community. For instance, researchers will often rely on the process of *proof* to debug simple translation mistakes in their specifications. This tends to be slow and reduces the value of the proof performed.

In our case we use three techniques for validation: eye-balling, typechecking, and the automatic generation of an interpreter for Java$_S$, based directly on the specification. It is not possible to remove all mistakes in the specification via this technique, but are surprising number are caught.

Step 4 We now, hopefully, have a valid model of the Java$_S$ language in a form that the computer can accept. The fourth step is to formulate, in the terms of the logic, the properties that we expect to hold of the specification. Typically this involves writing and typechecking propositions that relate various parts of the semantics. Though this may seem simple, we typically learn a lot by doing a thorough job of this: writing properties in terms of the logic forces us to state the problem in a logically clear and precise fashion, which is always the first step toward a successful proof.

Step 5 This step involves writing a rough (i.e. not machine acceptable) proof outline in a format close to that accepted by DECLARE. DECLARE supports the expression of proofs in a language that resembles that used by mathematicians, and thus allows a migratory path from a rough outline to a machine acceptable outline.

Surprisingly, this was the most valuable stage in the whole process, through which a major flaw in the original proof was discovered. An important by-product of this stage is *lemma discovery*, where we identify the key facts about component constructs that under-pin the argument. This is something often ignored by the theorem proving community: unless you are formalising a well-established corpus of mathematics, the necessary lemmas are not at all obvious *a priori*, and thus support for top-down proof development is essential.

Step 6 The sixth step is lengthy but completes our ultimate aim: we fill in the details of the rough proof outlines until the point where DECLARE's automated proof support takes over. Again surprisingly, this process turned up many unforeseen difficulties in the specifications and proofs, and contributed further to the process of lemma discovery. The end result is a proof outline that is machine checkable, human readable and, we claim, maintainable as further features are added to our language.

4 Developing A Machine Acceptable Model For Java$_S$

In Section 2 we sketched the runtime and typing semantics for the Java$_S$ language developed by Drossopoulou and Eisenbach. In this section we describe the formalization of this within simple type theory. We have used the DECLARE system to do this and here we give examples of the documents that have been written and machine checked. The documents described here are *abstracts*, i.e. summaries of theories that are checked to be consistent extensions of higher order logic.

Beside the model itself, what makes formalization interesting is that it uncovers many ambiguities in the formal definition. The results of this disambiguation process have already been presented in the modified semantics of Section 2. Here we will simply present some small examples of the process of formalization for the benefit of readers who are unfamiliar with these techniques.

4.1 Comments on the process of machine formalization

Specification within a fully formal framework requires attention to detail that is normally glossed over in written mathematics. In particular, many notational conveniences traditionally used by mathematicians are not available in most formal systems. The use of such conveniences is essential to the readability of a journal presentation of a proof, but they must be discarded in a machine presentation.

In addition, a journal presentation of operational semantics will frequently rely on a concrete means of describing a construct. A good example is the representation of environments in Java$_S$: in the journal presentation environments are represented as a list of declarations where each declaration is for a class, interface or variable. Logically this is fine, though at the expense of having to explicitly disallow repeated declarations of a class or variable. In the previous section we used a more abstract representation in terms of partial functions. This representation automatically excludes many ill-formed environments, and clarifies our mechanized presentation.

4.2 The Specification in DECLARE

DECLARE specifications act as high-level specifications that can be interpreted as axioms in an appropriate logic, or as a specification of an interpreter, if the rules are executable. The declarative forms available are simple (non-recursive) definitions, recursive datatype definitions (mutually recursive and recursive through positive type functions like *list*), inductive relations (again mutually recursive, with any monotonic operators), and recursive functions with a well-founded measure.[5]

The syntax classes described in Section 2 are easily defined in DECLARE as datatypes - we will not give an example here. Inductive relations are formulated by specifying a set of rules, and giving a name to each. When treated as a logical specification, DECLARE generates the appropriate axioms for the least fixed point of the set of rules (these axioms could be derived conservatively by well-known techniques [CM92, Pau94]).

For example, the $\diamond_{class-name}$ and $\sqsubseteq_{class}$ relations are defined by the text

```
least_fixed_point wf_class
(Object) [rw,prolog]
            ------------------------------------
            TE |- "Object" wf_class

(Decl) [rw,prolog]
            Cdec(TE,C) = SOME(classdec)
            ------------------------------------
            TE |- C wf_class

least_fixed_point subclass_of
(Refl) [rw,prolog]

            ------------------------------------
            TE |- C subclass_of C
```

[5]Not all the features listed here are fully implemented in the current version of DECLARE, for example monotonicity conditions are not currently checked.

```
(Step) [prolog]
           Cdec(TE,C) = SOME(CLASS(C',_,_,_)) &
           TE |- C' subclass_of C''
           ------------------------------------
           TE |- C subclass_of C''
```

Here *TE* is the type environment, and contains a partial function (i.e. a table) from class-names to class declarations. Note the necessity of extra syntactic detail that would be omitted in a written presentation, such as `SOME` to indicate that the class is actually in the domain of *CE*.

Pragmas such as `rw` and `prolog` provide interpretative information to proof tools when the specification is interpreted as a set of logical axioms: in particular `rw` indicates that the rule can be safely used as a (conditional) rewrite, and `prolog` that the rule can be safely used as a backchaining Prolog-style rule.

Formalizing the static semantics, compilation, the runtime rewrite system and runtime typing for runtime terms is relatively straight forward given DECLARE's collection of background theories. Iterated constructs are replaced by (bounded) universal quantification, thus a side condition like:

$$\text{each } arg_i \text{ is some ground value } val_i \ (1 \leq i \leq n)$$

becomes

```
∀i. i < n → ground(EL(i)(args)) &
LEN(vals) = LEN(args) &
∀i. i < n → EL(i)(args) = RValue(EL(i)(vals))
```

Note the index change to take advantage of the inbuilt theory of natural numbers and zero-indexed lists, the inclusion of the syntax constructor RValue that injects values into the domain of expressions, and the use of the inbuilt list operators `EL` and `LEN`.

The machine-acceptable specification runs to around 2500 lines in total. The specification was easily read and understood by the authors of the original journal paper when shown to them.

5 Validating the Model

We claim the specification developed in the previous section is a correct formulation of the language semantics presented in Section 2. But how do we know that this specification represents a model of the Java subset we are considering, and in what sense does it do so? Are our definitions even logically consistent?

Because of the style of definition we have used, relying on least fixed point and simple recursive definitions, consistency of our specification is essentially trivial. Validity is a harder question: we have to measure this against the Java language standard [GJS96], in addition to our own understanding of the meaning of constructs in the subset.

We use three techniques to validate the specification:

1. Eye-balling;

2. Type checking of higher order logic;

3. Compiling to ML and running test cases.

Here we concentrate on the third of these. In the appendix we describe how, by compiling 'manifestly executable' specifications down to executable ML code, we can generate an interpreter for the language based directly on our definitions. The interpreter is able to typecheck and execute concrete Java$_S$ programs if given a concrete environment. The interpreter is not efficient, but is sufficient to test small programs.

An example is required. The $\sqsubseteq_{class}$ relation shown in Sections 2.2.2 and 4.2 compiles to a ML function that is semantically equivalent to the following (we use CaML syntax as that is our target variant of ML):

```
let rec subclass_of CE C =
  (fun () -> if wf_class(C) then seq_cons(C,seq_nil) else fail()) seq_then
  (fun () -> match (PLOOKUP CE C) with
               NONE -> fail()
             | SOME(C',_,_,_) -> subclass_of CE C');
```

where `seq_nil`, `seq_cons` and the infix operator `seq_then` are the obvious operations on lazy lists, which we use to implement backtracking. Thus `subclass_of` will return a lazy list of identifiers and acts as a non-standard model of the relation defined by the inductive rules. Likewise we translate recursive functions to ML code, though no backtracking is needed here.

Of course, not all inductive relations or higher order logic terms are executable under this scheme. The exact executable subset is large, but importantly only inductive relations that satisfy strict *mode constraints* are allowed. That is, arguments must be divisible into inputs and outputs, and inputs must always be defined by previous inputs or generated outputs. This concept is familiar from Prolog: the mode constraints for the $\sqsubseteq_{class}$ relation are `(+,+,-)`. We choose not to translate directly to Prolog rules as unification is almost never required when expressing 'manifestly executable' rules. The elimination of all implicit unification steps is one way in which the existence of an algorithm is demonstrated.

We can now validate the semantics we have written for Java$_S$. DECLARE produces a CaML module for each abstract we have written. The modules are compiled together and linked against a module which implements core functionality. Test cases are expressed as higher order logic expressions (though better would be the ability to parse, compile and run Java programs directly from the source code).

Approximately 40 errors were discovered by using these techniques. The breakdown of these was as follows:

- Around 30 typing mistakes which led to mode violations.
- Around 5 logical mistakes in the typing rules.
- Around 5 logical mistakes in the runtime rules.

6 Formulating Key Properties

Our next step is to formulate, in the terms of the validated machine acceptable model, the properties that we expect to be true. In particular we will outline the formulation of the type soundness property.

6.1 Type Soundness

Loosely speaking, this theorem says that as evaluation progresses the configuration of our rewrite system always conforms to the types we expect, and that terms only ever narrow in type. The formulation differs from Drossopoulou and Eisenbach's because we assume a reduction is made, rather than deriving one for every non-ground term. This formulation will 'scale up' to the non-deterministic, threaded language constructs.

A *frame typing* F is a list of tables of types for local variables. A frame typing indicates what types we expect local variables to conform to. We will define the auxiliary concepts of typing for rterms, conformance ($\rightleftharpoons$) between frame typings and states, self-consistency of a heap ($\diamond_{heap}$) and widening between two heaps ($\leq_{heap}$) and two frame typings ($\leq_{ftyp}$) in the sections that follow.

Theorem 1 Type Soundness *Given a state s_0 that conforms to some frame typing F_0, if a well-typed term t_0 rewrites to some t_1 and a new state s_1, then either t_1 represents a raised exception, or there exists a new, larger frame typing F_1 such that t_1 has some narrower type than t_0 in the new state and environment, and s_1 conforms to F_1. That is, if*

- $s_0 = (f_0, h_0)$
- $\Gamma \vdash h_0 \diamond$
- $\Gamma, h_0 \vdash f_0 \rightleftharpoons F_0$
- $\Gamma, h_0, F_0 \vdash t_0 : ty_0$ *and*
- $t_0, s_0 \rightsquigarrow_{(\Gamma,p)} t_1, s_1$ *with* $s_1 = (f_1, h_1)$

then t_1 represents an exception or there exists F_1 and ty_1 such that

- $\Gamma \vdash h_1 \diamond$
- $\Gamma, h_1 \vdash f_1 \rightleftharpoons F_1$
- $\Gamma, h_1, F_1 \vdash t_1 : ty_1$,
- $h_1 \leq_{heap} h_0$,
- $F_1 \leq_{ftyp} F_0$ *and*
- $\Gamma \vdash ty_1 \leq_{wdn} ty_0$.

The proof is by induction on the derivation of the typing judgement for t_0. The outline sketched by Drossopoulou and Eisenbach is a good guide, but is 'rough around the edges.'

Drossopoulou and Eisenbach limit their invariant to state conformance. In fact, a much stronger invariant, 'widening between heaps', is needed to ensure type soundness. We will discuss this further in Section 8.

6.1.1 Typing for rterms

We have yet to define what we mean by typing for rterms. This is a central consideration somewhat overlooked by Drossopoulou and Eisenbach, and led in part to an error in their proof (described in Section 8). The typing rules for rterms generally follow those for annotated Java$_S$ expressions, with the addition of rules for addresses (these make the typing relation dependent on the current heap):

$$\frac{h(addr) = \ll \ldots \gg^{C}}{\Gamma, h \vdash addr : C} \qquad \frac{s(addr) = \texttt{[[...]]}^{ty}}{\Gamma, h \vdash addr : ty\,\texttt{[]}}$$

$$\frac{}{\Gamma, h \vdash \texttt{null} : C} \qquad \frac{}{\Gamma, h \vdash \texttt{null} : I} \qquad \frac{n > 0}{\Gamma, h \vdash \texttt{null} : ty\,\texttt{[]}^n}$$

Note that we no longer demand unique typing: null values can be considered to have any reference type. The rule for assignments must also be different: the new rule drops the requirement that the source type be narrower than the target type in the case of array assignments, since this will be checked by runtime type checking. We will return to this issue in Section 8. The rule for local variables is:

$$\frac{\begin{array}{l}\Gamma, h \vdash e : ty_e \\ \Gamma, h \vdash (\mathit{fidx}, \mathit{id}) : ty_v \\ \Gamma, h \vdash ty_e \leq_{wdn} ty_v\end{array}}{\Gamma, h \vdash (\mathit{fidx}, \mathit{id})\ \texttt{:=}\ e : \texttt{void}}$$

While the rule for arrays is:

$$\frac{\begin{array}{l}\Gamma, h \vdash e : ty_e \\ \Gamma, h \vdash arr\,\texttt{[}idx\texttt{]} : ty_v\end{array}}{\Gamma, h \vdash arr\,\texttt{[}idx\texttt{]}\ \texttt{:=}\ e : \texttt{void}}$$

6.1.2 Conformance

The notion of a state conforming to a frame typing was used in the statement of the type soundness theorem, and is defined as follows. The definitions are a corrected version of those found in [DE97b]. A value v *weakly conforms* to a type ty with a heap h and type environment Γ if

- ty is a primitive type and v is an element of that primitive type; or
- ty is a reference type and v is a null pointer; or
- v is an address, $h(v)$ is an instance of a class type C and $\Gamma \vdash C \leq_{wdn} ty$; or
- v is an address, $h(v)$ is an array with element type $ty'\,\texttt{[]}^n$ and $\Gamma \vdash ty'\,\texttt{[]}^{n+1} \leq_{wdn} ty$.

Value conformance states that the components of an object or array weakly conform. A value v *conforms* to a type ty with heap h and type environment Γ if v weakly conforms to ty and

- if v is an address then $h(v) = \ll \mathit{fldvals} \gg^C$ and for each $(\mathit{field}, \mathit{idx}, ty') \in \mathsf{FDecs}(C)$ $\mathit{fldvals}(\mathit{field})$ is defined and weakly conforms to ty'; and
- if v is an address then $h(v) = \texttt{[[}vec\texttt{]]}^{ty'}$ and each $val \in vec$ weakly conforms to ty'.

A heap h *conforms* (i.e. is self-consistent) in Γ, written $\Gamma \vdash h \diamond$ if these hold:

- if $addr$ is an address and $h(addr) = \ll \mathit{fldvals} \gg^C$ then $addr$ conforms to C.
- if $addr$ is an address and $h(addr) = \texttt{[[}vec\texttt{]]}^{ty'}$ then $addr$ conforms to $ty'\,\texttt{[]}$.

A set of frames f *conforms* to a frame typing F (with a heap h and in Γ), written $\Gamma, h \vdash f \rightleftharpoons F$ if

- every local variable in every frame of f conforms to the corresponding type given in F;

6.2 Heap Widening

We now define the notion of widening between heaps: this is a notion new to this report and is required for the induction invariant of the type soundness proof. Heap widening is a strong property, and is essential for proving the maintenance of types under state changes. We expect heaps to maintain value conformance in the following way: for in environment Γ a heap h_1 is *narrower* than a heap h_0 (i.e. h_0 is wider than h_1) at a set of addresses A, written $\Gamma, A \vdash h_1 \leq_{heap} h_0$ if

- for every *addr* in both A and h_0, if *addr* conformed to some type *ty* in the context of h_0, then *addr* also conforms to *ty* in the context of h_1.

We restrict the definition to a set of addresses A to allow for the possibility of garbage collection: we would then demand continued conformance only at a set of 'active addresses'. Our current working definition makes A universal.

6.3 Formalized Type Soundness

Type soundness as expressed in Section 6.1 is in fact three properties, one for each syntax category within rterms. For variables the property is stated in DECLARE as follows:

```
  TE wf_tyenv ∧
  TE |- heap0 frames_conform_to F0 ∧
  TE |- heap0 heap_conforms ∧
  TE |- p prog_hastype ∧
  s0 = (frames0,heap0) ∧
  s1 = (frames1,heap1) ∧
  (var0,s0) var_reduce(TE,p) (var1,s1) ∧
  (TE,F0,heap0) |- var0 rvar_hastype ty0
→
   exceptional_var(var1)
∨ ∃F1 ty1.
     F1 ftyenv_leq F0 ∧
     heap1 heap_leq heap0 ∧
     (TE) |- frames1 frames_conform_to F1 ∧
     (TE) |- heap1 heap_conforms ∧
     (TE,F1,heap1) |- var1 rvar_hastype ty1 ∧
     TE |- ty1 widens_to ty0)
```

The formulation is a straight-forward transcription of the property expressed in Section 6.1. The property is expressed similarly for expressions and statements. The latter two syntax categories do not have types, thus the statements are simpler.

When we actually prove type soundness, we strengthen the induction invariant in the following ways:

- If t_0 is a field variable, then t_1 is also and $ty_0 = ty_1$. This is needed because field types on the left of assignments cannot narrow, otherwise runtime typechecking would be needed.

- If t_0 is an array variable, then t_1 is also, and similarly for stack variables.

6.4 Key Lemmas

The following is a selective list of the lemmas that have been proved that form the basis for the type soundness proof.

`Object` is the least class

- If $\vdash TE \diamond_{tyenv}$ and $TE \vdash \texttt{Object} \sqsubseteq_{class} C$ then $C = \texttt{Object}$.

Widening is transitive and reflexive

- This result holds for the $\sqsubseteq_{class}$, $\sqsubseteq_{int}$, $\leq_{ref}$, $\leq_{comp}$, $\leq_{arr}$ and $\leq_{wdn}$ relations.
- The transitivity results only holds for well-formed environments.

Visibility is maintained at narrower types

- If $\vdash TE \diamond_{tyenv}$, $TE \vdash C_1 \sqsubseteq_{class} C_0$ and $((C_f, fld), vt) \in \mathsf{FDecs}(\Gamma, C_0)$ then $((C_f, fld), vt) \in \mathsf{FDecs}(\Gamma, C_1)$.
- Similarly if $\vdash TE \diamond_{tyenv}$, $TE \vdash t_1 \leq_{ref} t_0$ and $(m, at \to rt_0) \in \mathsf{MSigs}(TE, t_0)$ then there exists some rt_1 with $TE \vdash rt_1 \leq_{wdn} rt_0$ and $(m, at \to rt_1) \in \mathsf{MSigs}(TE, t_1)$ (i.e. methods are still visible though with possible narrower return types).

Method fetching behaves correctly

- If $\vdash TE \diamond_{tyenv}$, $TE \vdash prog \diamond$, $(m, at \to rt) \in \mathsf{MSigs}(TE, t)$ and $MethBody(m, at, t, p) = method$ then $TE \vdash method : rt$.
- That is, fetching the body of a method at runtime results in a method of the type we expect.

Relations are monotonic under $\leq_{heap}$ and $\leq_{ftyp}$

- This holds for the typing, value conformance and frame conformance relations.

State manipulations preserve $\leq_{heap}$ and heap/frame conformance

- Holds for object and array allocation.
- Also holds for field and array and local variable assignment, and method call.

6.5 Type Soundness of Compilation

To complement the type soundness proof, we must prove that the process of compile-time disambiguation preserves types. This is easy to state:

```
   (CE,IE) wf_tyenv ∧
   (CE,IE) |- p prog_hastype ∧
   (CE,IE) |- p prog_compiles_to p'
→
   (CE,IE) |- p' prog_hastype
```

This property is proved by demonstrating that a similar property holds for all syntax classes from expressions through to class bodies.

7 Sketching Outlines of the Proofs

We are now ready to begin the proof of the type soundness theorem formulated in the previous section. The reader should keep in mind that when this proof was begun, the only guide available was the rough proof outline in [DE97b], and this was based on a formulation of the problem that was subsequently found to be flawed. Thus the process is very much one of *proof discovery* rather than proof transcription. In this section we describe the process of proof discovery, and in Section 8 a major flaw in the proof discovered by this technique is described.

The proof of type soundness proceeds by the derivation of the typing judgment for the term t - this is clear. We have to consider one case for each rule in that inductive relation. Our aim is to write out some of the cases of the proof in a language that is close to DECLARE's proof language, and at a level that is close to that which will be machine checkable.

7.1 Sketching v. Machine Checking

We will give an example that demonstrates the difference between a scratch proof outline, and a machine checkable version of the same case of the proof. Below is an the scratch outline for a trivial case:

```
case StackVar
  "var0 = RStackVar(fidx,id)"

   // var0 has a reduction, but stack vars do not (until they are resolved)
   contradiction by <var0_reduces>;
```

The machine-acceptable form of the typing rule that this case of the induction is based on is:

```
    fidx < LEN(FT) &
    PLOOKUP(EL(fidx)(FT))(id) = SOME(varty)
    -----------------------------------------------------
    (CE,IE,FT,s) |- RStackVar(fidx,id) rvar_hastype SOME(varty)
```

The final, machine-acceptable version of the proof is

```
case StackVar
  "var0 = RStackVar(fidx,id)" [auto]
  "fidx < LEN(FT0)"
  "PLOOKUP(EL(fidx)(FT0))(id) = SOME(var0_ty)"

  contradiction by rule cases on <var0_reduces>;
```

Several things have been added to get the machine to accept the proof. For various reasons, DECLARE demands that all inductive hypotheses be listed for each induction case, It gives assistance in constructing these, but they must be listed in the document. Secondly, an extra fact (that rule case analysis must be applied) has been added to the justification of the claim that a contradiction has arisen; and the tag `[auto]` has been added to implicitly include the given fact in all future justifications. These are typical examples of how machine checking requires that the user be more precise and/or helpful in small ways.

7.2 A More Complicated Case

Below is the scratch proof outline for the case of field access variables. The proof decomposes into two cases, one for when each sub-expression reduces. The proof outline is deliberately left scratchy, with documentation notes included.

```
case Field
  "var0 = RField(obj0,C',fld)"
  "ihyp for (TE,FT,s) |- obj0 rexp_hastype SOME(VT(Class(C0),0N))" <ihyp>
  "(TE,FT0,s0) |- obj0 rexp_hastype SOME(VT(Class(C0),0N))"     <obj0_types>
  "TE |- C0 subclass_of C'"                                      <C0_subclass>
  "Cdec(TE,C') = SOME(CLASS(C'',Is,fields,methods))"            <C_lookup>
  "PLOOKUP(fields)(fld) = SOME(var0_ty)"                            <fld_lookup>

  let "obj0_ty = SOME(VT(Class(C0),0N))";

   // do case analysis on whether obj0 has a reduction:
  suppose "(obj0,s0) exp_reduces (obj1,s1)"                         <obj0_reduces>

      // did obj0 give an exception?
     per cases by <ihyp>,<obj0_reduces>
       case "exceptional_exp(obj1)"
          // easy - var1 will also be exceptional

        // in this case, obj0 reduces conformantly, to something narrower.
       case "FT0 ftyenv_leq FT1"
            "TE |- s1 state_conforms_to FT1"
            "(TE,FT1,s1) |- obj1 rexp_hastype obj1_ty"
            "TE |- obj1_ty widens_to obj0_ty"

          // obj1 will be a class, and indeed some subclass of C0
          // Note: we have to make sure we didn't start with an Object and finish
          // with an array.  But C can't be "Object", 'cos its got a field.
          // We'll need a lemma for this.
         have `C0 <> "Object"` [auto] by <Object-field-lemma>,...;

         consider C1 st
            "obj1_ty = SOME(VT(Class(C1),0N))"
            "TE |- C1 subclass_of C0" <C1_subclass> by <class-widens-lemma>,...;

          // and this makes C1 a subclass of C0
         have "TE |- C1 subclass_of C'" <C_subclass1>
                   by <subclass-trans>,<C1_subclass>,<C0_subclass>;;

          // Now we can proce the typing judgment for var1.
          // It will have the same type as before, as field types never change.
         have "(TE,FT1,s1) |- var1 rexp_hastype var0_ty" by ...;

          // and that's all folks!
         qed by <bits-n-pieces>;
     end;

  otherwise
     // var0 is ground, and the access is left unresolved until var0 is used.
    contradiction by not <obj0_reduces>, ...;
```

The proof outline follows the same discipline that any mathematician or computer scientist is taught: decompose the problem; assess the information that is available; assess what is to be proved; determine the correct plan of action based on this; determine the lemmas that are required; and construct the solution accordingly. We are simply using a different medium to pen-and-paper, and progressively heading toward a machine checked proof.

7.3 Detailed Machine Checking of the Proofs

In the previous section we outlined two cases of the type soundness proof, and also indicated what is involved in getting DECLARE's proof checker to accept the proof. This process is repeated for all 36 major cases of the type soundness proof.

Typically, a first pass at a case of the proof will result in:

- 50% of the steps (i.e. logical leaps) in the proof are accepted immediately;
- 25% of the steps require the addition of one or two supporting facts, and occasionally some explicit instantiations;
- 25% of the cases require more thought than anticipated.

The completed `Field` case is shown below. `exp_types` is a macro for the induction invariant.

```
case Field
   "var0 = RField(obj0,C',fld)" [auto]
   "~exp_ground obj0 ==>
       exp_types TE FT0 s0 obj0 (SOME(VT(Class(C0),0N)))" <ihyp0>
   "(TE,FT0,s0) |- obj0 rexp_hastype SOME(VT(Class(C0),0N))" <obj0_types>
   "TE |- C0 subclass_of C'"                                 <C0_subclass>
   "Cdec(TE,C') = SOME(CLASS(C'',Is,fields,methods))"       <C_lookup>
   "PLOOKUP(fields)(fld) = SOME(var0_ty)"                    <fld_lookup>

  let "obj0_ty = SOME(VT(Class(C0),0N))";

   // do case analysis on whether obj0 has a reduction:
  suppose "(obj0,s0) exp_reduces (obj1,s1)"                  <obj0_reduces>

     // did obj0 give an exception?
    per cases by <ihyp0>,<obj0_reduces>
      case "exceptional_exp(obj1)" [auto]
        qed by <exceptional>;

       // obj0 reduces conformantly, to something narrower.
      case "FT0 ftyenv_leq FT1" <FT1_larger>
           "TE |- s1 state_conforms_to FT1" <s1_conforms>
           "(TE,FT1,s1) |- obj1 rexp_hastype obj1_ty" <obj1_ty>
           "TE |- obj1_ty expty_widens_to obj0_ty" <obj1_narrower>

         // C can't be "Object", 'cos its got a field.
        suppose `C0 = "Object"` [auto]
           contradiction by <TE_wf>,<fld_lookup>,<Object-field-lemma>;
        otherwise

        consider C1 st
           "TE |- C1 subclass_of C0" <C1_subclass>
           "obj1_ty = SOME(VT(Class(C1),0N))" [auto]
                 by <class-widens-lemma>,<TE_wf>,<IE_wf>,<obj1_narrower>;;

         // prove var1 is welltyped, with the same type

        have "TE |- C1 subclass_of C'" <C_subclass1>
               by <subclass-trans>,<C1_subclass>,<C0_subclass>;;
        then have "(TE,FT1,s1) |- var1 rvar_hastype var0_ty" <var1_types>
                by <rstatics__Field> ["fields","methods","Is"],<obj1_ty>,
                   <C_lookup>,<fld_lookup>,<C_subclass1>;
        qed by <FT1_larger>,<s1_conforms>,<var1_types>,<var1_narrower>;
    end;

  otherwise
     // var0 is ground, and the access is left unresolved until var0 is used.
    contradiction by not <obj0_reduces>, rule cases on <var0_reduces>;
```

Machine checking the proofs up to various lemmas that were initially assumed took around 30 minutes to one hour per case.

The reader should note that although a powerful automated routine may be able to find the above proof after the fact, the very process of writing the proof corrected significant errors that would have confounded even the best automated routines, and thus increased automation is of less use before a correct formulation is found.

8 Unmasking Mistakes

In this section we describe one major error and one major omission in Drossopoulou and Eisenbach's presentation of the type soundness proof. The error was discovered while outlining the proof, and the omission while performing detailed machine checking. We also describe an error in the language spec that we independently rediscovered.

8.1 Runtime Typechecking, Array Assignments, and Exceptions

In Drossopoulou and Eisenbach's original formulation the type soundness property was stated along the following lines (emphasis added):

Theorem 2 *If a well-typed term t is not ground, then it rewrites to some t' (and a new state s and environment Γ). Furthermore, either t'* **eventually rewrites** *to an exception, or t' has some narrower type than t, in the new state and environment.*

The iterated rewriting was an attempted fix for a problem demonstrated by the following program:

```
void silly(C arr[], C s) {
   arr[1] = s;
}
```

At runtime, `arr` may actually be an array of some narrower type, say `C'` where `C'` is a subclass of `C`. Then the array assignment appears to become badly typed *before* the exception is detected, because during the rewriting the left side becomes a narrower type than the right. Thus they allow the exception to appear after a number of additional steps.

However, `arr` can become narrower, and then subsequently fail to terminate! Then an exception is never raised, e.g.

```
      arr[loop()] = s;
```

The problem occurs in even simpler cases, e.g. when both `arr` and `s` have some narrower types `C'[]` and `C'`. Then, after the left side is evaluated, the array assignment appears badly typed, but will again be well typed after the right side is evaluated.

To fix this problem requires a different understanding of the role of the types we assign to rterms. Types for intermediary rterms exist just to help express the type soundness invariant of the abstract machine, i.e. the allowable states that a well-typed instance of the machine can reach. In particular, the array assignment rule must be relaxed to allow what appear to be badly typed assignments, but which later get caught by the runtime typechecking mechanism.

This problem is an interesting case where the attempted re-use of typing rules in a different setting (i.e. the runtime setting rather than the typechecking setting) led to a subtle error, and one which we believe would only have been detected with the kind of detailed analysis that machine formalization demands. The mistake is unmissable in that setting. It is a surprisingly difficult exercise to discover the exact logical step where Drossopoulou and Eisenbach's original proof is flawed.

8.2 Side Effects on Types

A significant omission in Drossopoulou and Eisenbach's proof is as follows: when a term has two or more subterms, e.g. *arr*[*idx*] := *e*, and *arr* makes a reduction to *arr*', then the types of *idx* and *e* may change (become narrower) due to side effects on the state. This possibility had not been considered by Drossopoulou and Eisenbach, and requires a strengthening of the induction invariant incorporating heap widening ($\leq_{heap}$), and a number of significant new lemmas stating that typing is monotonic with respect to the $\leq_{frame}$ and $\leq_{heap}$ relationship, up to the $\leq_{wdn}$ relationship. The foremost of these lemmas has been mentioned in Section 6.4. This problem was only discovered while doing detailed machine checking of the rough proof outline, and the proofs of these lemmas are surprisingly the most complex in the entire proof.

8.3 What Methods are visible from Interfaces?

In the process of finishing the proofs of the lemmas described in Section 6.4 we independently rediscovered a significant flaw in the Java language specification that had recently been found by developers of a Java implementation [PB97]. In theory the flaw will not effect type soundness, but the authors of the language specification have confirmed that the specification needs alteration.

The problem is this: In Java, all interfaces and arrays are considered subtypes of the type `Object`, in the sense that a cast from an interface or array type to `Object` is permitted. The type `Object` supports several "primitive" methods, such as `<object>.hashValue()` and `<object>.getClass()` (there are 11 in total). The question is whether expressions whose static type is some interface support these methods.

Morally speaking, interfaces should indeed support the `Object` methods - any class that actually implements the interface will support these methods by virtue of being a subclass of `Object` or an array. Indeed, the Sun JDK toolkit does allow these methods to be called from static interface types, as indicated by the successful compilation (but not execution) of the code:

```
public interface I

public class Itest
  public static void main(String args[])
    I a[] =  null, null ;
    a[0].hashCode();
    a[0].getClass();
    a[0].equals(a[1]);
```

However, the language specification clearly says that interfaces only support those methods listed in the interface or its superinterfaces, and that there is no 'implicit' superinterface (i.e. there is no corollary to the 'mother-of-all-classes' `Object` for interfaces:

> The members of an interface type are fields and methods. The members of an interface are all of the following:
>
> - Members declared from any direct superinterfaces
> - Members declared in the body of the interface.

...
There is no analogue of the class `Object` for interfaces; that is, while every class is an extension of class `Object`, there is no single interface of which all interfaces are extensions.

[GJS96], pages 87 and 185

9 Summary

This report has presented corrections to the semantics of Java$_S$, a machine formalization of this semantics, a technique to partially validate the semantics, and an example of the use of new mechanized proof techniques to prove the type soundness property for that language.

9.1 Using Formal Techniques to Specify Languages

The work developed here demonstrates in part how formal techniques can be used convincingly to help specify a major language. Java itself is far more complicated than Java$_S$, but we have covered some of the initial groundwork.

Drossopoulou and Eisenbach's formalization was the inspiration for this work and is an excellent example of the use of operational semantics. We suggest that the form of specification presented here may ultimately provide a better repository of the information presented in the formal paper, especially when flexible tools are provided to read and interpret it. The specification can be used as both an interpreter and as the basis for formal proofs.

9.2 Using Formal Proofs to Find Mistakes

The disciplined approach enforced when writing a proof to be accepted by a mechanized tool ensures errors like those described in Section 8 are detected. It also encourages a high degree of clarity in the formalization of a problem in the first place.

The declarative proof language played a crucial role: it allowed the author to think clearly about the language while preparing the proof outline for the computer. The first error was found when simply preparing the proof outline, rather than when checking it in detail. However, during this process of preparation the question 'will a machine accept this proof?' was consistently asked, and this ensured that unwarranted logical leaps were not made.

The independent rediscovery of the mistake in the language specification described in Section 8.3 indicates that such errors can indeed be discovered by the process of formal proof. The mistake had already been discovered, presumably by implementors attempting to follow the language specification precisely.

Formal specification in a logic is well known to be of value in discovering bugs in specifications. This work has demonstrated that proof sketching and proof checking within a formal framework can also detect serious mistakes while the theoretical framework for the language is still under development. It is interesting to note that of the three major errors, two were only discovered at a late stage in the formal proof.

9.3 Related Work

Tobias Nipkow and David von Oheimb have been working on developing a proof of the type soundness property for a similar subset of Java in the Isabelle theorem prover. I am extremely grateful for the chance to talk with them and have adopted suggestions they have made. These two works will provide a valuable case study of the utility of various theorem proving methods to this kind of problem. Isabelle is a mature system and has complementary strengths to DECLARE, notably strong generic automation and manifest soundness. A tool which unites these based on such concrete experience is an exciting prospect.

Computational Logic, Inc. have released a formal model of a subset of the Java Virtual Machine (JVM) [Coh97]. The model is called the 'defensive' JVM (or dJVM) because it includes sufficient run-time checks to assure type-safe execution (or at least to detect and prevent any unsafe execution). In the standard JVM these checks are not present. Any future work on extending our work to cover type safety for the JVM could be based partly on this model.

9.4 Future Work

The model presented in this article has scope to be extended in many directions. The treatment could be expanded to encompass features such as exceptions, constructors, access modifiers, static fields and static methods without major problems, although this would involve a significant expansion in the size of the proofs. Features such as threads and Java's semantically visible garbage collection pose greater problems, but should also be possible.

The work began as a case study for the application of a declarative proof language to operational reasoning, and there are ways in which DECLARE (or similar systems) could be improved based on this experience. The most necessary feature is some degree of 'Computer Aided Proof Writing', as described briefly in [Sym97].

Acknowledgments

I would like to thank Sophia Drossopoulou and Sarfraz Khurshid for an excellent day at Imperial spent discussing this work and its possible relevance to their project. I also thank David von Oheimb, Tobias Nipkow, Mark Staples, Michael Norrish, Mike Gordon and Frank Pfennig for the useful discussions I have had with them concerning this work.

References

[CM92] J. Camilleri and T.F Melham. Reasoning with inductively defined relations in the HOL theorem prover. Technical Report 265, University of Cambridge Computer Laboratory, August 1992.

[Coh97] Richard Cohen. Defensive Java Virtual Machine, version 0.5 alpha release, June 1997. Published on the WWW at `http://www.cli.com/software/djvm`.

[DE97a] Sophia Drossopolou and Susan Eisenbach. Is the Java type system sound? June 1997. Submitted to special edition of Theory and Practive of Object Systems.

[DE97b] Sophia Drossopolou and Susan Eisenbach. Java is type safe - probably. In *11th European Conference on Object Oriented Programming.* June 1997. To be published.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Addison-Wesley, 1996.

[GM93] M.J.C Gordon and T.F Melham. *Introduction to HOL: A Theorem Proving Assistant for Higher Order Logic.* Cambridge University Press, 1993.

[Har97] John R. Harrison. Proof style. Technical Report 410, University of Cambridge Computer Laboratory, January 1997.

[Muz93] M. Muzalewski. *An Outline of PC Mizar.* Foundation Philippe le Hodey, 1993.

[NN96] Dieter Nazareth and Tobias Nipkow. Formal verification of algorithm W: The monomorphic case. In J. Grundy J. von Wright and J. Harrison, editors, *Theorem Proving in Higher Order Logics: 9th International Conference*, Lecture Notes in Computer Science, pages 331–346. Springer-Verlag, August 1996.

[Nor97] Michael Norrish. An abstract dynamic semantics for C. Technical Report 421, University of Cambridge Computer Laboratory, May 1997.

[ORR+96] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M.K. Srivas. PVS: Combining specification, proof checking, and model checking. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, pages 411–414, New Brunswick, NJ, July/August 1996. Springer-Verlag.

[Pau90] L. C. Paulson. Isabelle: The next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–385. Academic Press, 1990.

[Pau94] L. C. Paulson. A fixed point approach to implementing (co)inductive definitions. In A. Bundy, editor, *12th International Conference on Automated Deduction*, pages 148–161. Springer, 1994.

[PB97] Roly Perera and Peter Bertelsen. The unofficial Java bug report, June 1997. Published on the WWW at `http://www2.vo.lu/homepages/gmid/java.htm`.

[Sym93] Don Syme. Reasoning with the formal definition of Standard ML in HOL. In *Higher order logic theorem proving and its applications: 6th International Workshop, HUG '93*, number 780 in LNCS, Vancouver, B.C., August 1993. Springer-Verlag.

[Sym97] Don Syme. DECLARE: A prototype declarative proof system for higher order logic. Technical Report 416, University of Cambridge Computer Laboratory, March 1997.

[Van93] Myra VanInwegen. HOL-ML. In *Higher order logic theorem proving and its applications: 6th International Workshop, HUG '93*, number 780 in LNCS, pages 59–72, Vancouver, B.C., August 1993. Springer-Verlag.

[Van97] Myra VanInwegen. Towards type preservation for core SML. *Journal of Automated Reasoning*, January 1997. Submitted for Special Issue on Formal Proof.

A An Introduction to DECLARE

DECLARE is a declarative proof system for polymorphic, simple higher order logic. It is designed to aid in the production of clear, readable, maintainable specification and proof documents. DECLARE is not a complete or polished system, and has been developed with the aim of testing various features that could be incorporated in existing, supported theorem provers such as HOL, Isabelle and PVS. It has been influenced heavily by Mizar [Muz93], HOL [GM93], Isabelle [Pau90] and PVS [ORR+96]. It is not an LCF-style s ystem as not all deductions are reduced to a primitive deductive framework. The features of interest here are:

- The declarative language used to express proof outlines.

- The support for modularization, separate processing and top-down formalization, which leads to a well-structured, efficient working environment.
- The automated proof support.

A.1 The Proof Language

We try to achieve, by the simplest means possible, results that are both *machine checkable* and *human readable.* [6]

DECLARE's proof language is inspired by Mizar and work by Harrison [Muz93, Har97], and is a way of specifying the outline of a proof using induction, case-decomposition, and arbitrary (but substantiated) claims. The language will be demonstrated by example in Section 7.3. Such proof languages are called *declarative*, to place them in contrast to 'procedural' tactic based mechanisms for specifying proofs. The use of a declarative proof language has several advantages:

- Declarative proofs are more readable than tactic proofs.
- Proof interpretation always terminates, unlike tactic proofs which are expressed in a Turing-complete language.
- Declarative proofs are potentially more maintainable under changes to the specification and the prover.
- Declarative proofs are potentially more portable. Specification and proof documents developed with DECLARE are, in principle, portable to other proof systems.
- A declarative style may appeal to a wider class of users, helping to deliver automated reasoning and formal methods to mathematicians and others.

A.2 The Working Environment

When using DECLARE, large bodies of work are broken into a series of *articles*, each of which may have an interface called an *abstract.* Articles are checked relative to the abstracts they import, and must 'implement' the abstract they export. Abstracts may be pre-compiled, which, in combination with the `make` system, gives us a simple yet light-weight and effective means for maintaining the coherence of large collections of specifications and proofs. This approach also means DECLARE typically uses only 5-6 MB of memory when executing.

A.3 Automation

DECLARE proofs are only proof outlines, and require automation to fill in the gaps in the argument. In this way the proof language acts as a bridge between the human and the automated capabilities of the proof checker.

The automation we use in this report is fairly straightforward:

- We use Boyer-Moore/Isabelle style simplification with conditional, higher-order rewriting to normalize expressions. Simplification is performed under the auspices of a two-sided sequent calculus like that used by PVS. During simplification we:

[6]Some researchers take the view that human readable proofs should be generated as output from mechanized proofs: this may be possible, but it is a highly complex process and the results are not yet convincing. Our approach is to make the input readable in the first place.

- Apply safe introduction and elimination rules, e.g. choosing witnesses and splitting conjuncts in goal formulae;
- Apply 'unwinding' rules to eliminate local constants from existential and universal formulae, and the sequent itself;
- 'Untuple' all pair, tuple and record values;
- Case-split on constructs such as conditionals and quantified structural variables (booleans, options etc.);
- Apply a large background set of rewrites collected from imported abstracts;
- Normalize arithmetic expressions;
- Use exploratory unwinding of some definitions, in the style of PVS.
- Support the controlled left-right simplification of certain guarded expressions, which helps avoid common causes of non-terminating rewriting strategies.

• After simplification we use a simple tableau prover (with iterative deepening and some minor equality rules) to search for values for unknowns.

This level of automation has been sufficient during exploratory proof development, since in this most important stage we are content with guiding the prover through the proof without expecting complex steps, such as inductions, to be automated. The only significant problems arise when we venture into the problem space which requires both significant equality and proof-search reasoning (this is still a major research area), or equality reasoning not amenable to rewriting (completion of DECLARE's decision procedures should help with this).

Automation in DECLARE is guided by purely declarative tags: lemmas are given once-only 'how to use me' declarations, and no weightings or other hints are specified when a lemma is used. This ensures that proof documents are not overly reliant on quirks of the underlying prover.

B The Full Widening Rules

These rules determine the widening (subtype) relation.

$$\frac{\Gamma \vdash C \sqsubseteq_{class} C'}{\Gamma \vdash C \leq_{sref} C'} \qquad \frac{\Gamma \vdash I \sqsubseteq_{int} I'}{\Gamma \vdash I \leq_{sref} I'} \qquad \frac{I \in \Gamma}{\Gamma \vdash I \leq_{sref} \texttt{Object}} \qquad \frac{\begin{array}{l}\Gamma \vdash C \sqsubseteq_{class} C' \\ \Gamma \vdash C' :_{imp} I \\ \Gamma \vdash I \sqsubseteq_{int} I'\end{array}}{\Gamma \vdash C \leq_{sref} I'}$$

$$\frac{ty \in \textit{prim-types}}{\Gamma \vdash ty \leq_{comp} ty} \qquad \frac{\begin{array}{l} ty, ty' \in \textit{simple-ref-types} \\ \Gamma \vdash ty \leq_{sref} ty'\end{array}}{\Gamma \vdash ty \leq_{comp} ty'}$$

$$\frac{\begin{array}{l} ty \in \textit{component-types} \\ n > 0\end{array}}{\Gamma \vdash ty\,\texttt{[]}^n \leq_{arr} \texttt{Object}} \qquad \frac{\begin{array}{l} n > 0 \\ \Gamma \vdash ty \leq_{comp} ty'\end{array}}{\Gamma \vdash ty\,\texttt{[]}^n \leq_{arr} ty'\,\texttt{[]}^n}$$

$$\frac{\begin{array}{l} ty, ty' \in \textit{array-types} \\ \Gamma \vdash ty \leq_{arr} ty'\end{array}}{\Gamma \vdash ty \leq_{ref} ty'} \qquad \frac{\begin{array}{l} ty, ty' \in \textit{simple-ref-types} \\ \Gamma \vdash ty \leq_{sref} ty'\end{array}}{\Gamma \vdash ty \leq_{ref} ty'} \qquad \frac{ty \in \textit{ref-types}}{\Gamma \vdash \texttt{nullT} \leq_{ref} ty}$$

$$\frac{ty \in \textit{prim-types}}{\Gamma \vdash ty \leq_{wdn} ty} \qquad \frac{\begin{array}{l} ty, ty' \in \textit{ref-types} \\ \Gamma \vdash ty \leq_{ref} ty'\end{array}}{\Gamma \vdash ty \leq_{wdn} ty'}$$

C The Full Traversal Rules

These rules determine what methods and fields are visible from a given class. The relations evaluate graphs, which in well-formed environments determine partial functions.

$$\frac{\Gamma(C).\mathrm{flds}(\mathit{field}) = \mathit{ty}}{\Gamma \vdash (C, \mathit{ty}) \in \mathsf{FDec}(C, \mathit{field})}$$

$$\frac{\Gamma(C) = \langle C_{sup}, \mathit{flds}, \ldots\rangle \quad \mathit{flds}(\mathit{field}) = \texttt{undef} \quad \Gamma \vdash (C', \mathit{ty}) \in \mathsf{FDec}(C_{sup}, \mathit{field})}{\Gamma \vdash (C', \mathit{ty}) \in \mathsf{FDec}(C, \mathit{field})}$$

$$\frac{\Gamma(C).\mathrm{flds}(\mathit{field}) = \mathit{ty}}{\Gamma \vdash ((C, \mathit{field}), \mathit{ty}) \in \mathsf{FDecs}(C)} \qquad \frac{\Gamma(C) = \langle C_{sup}, \ldots\rangle \quad \Gamma \vdash ((C', \mathit{field}), \mathit{ty}) \in \mathsf{FDecs}(C_{sup})}{\Gamma \vdash ((C', \mathit{field}), \mathit{ty}) \in \mathsf{FDecs}(C)}$$

$$\frac{\Gamma(C).\mathrm{meths}(\mathit{meth}, \mathit{at}) = \mathit{rt}}{\Gamma \vdash ((\mathit{meth}, \mathit{at}), \mathit{rt}) \in \mathsf{MSigs}_C(C)}$$

$$\frac{\Gamma(C) = \langle C_{sup}, \ \ldots \ , \mathit{meths}\rangle \quad \mathit{meths}(\mathit{meth}, \mathit{at}) = \texttt{undef} \quad \Gamma \vdash ((\mathit{meth}, \mathit{at}), \mathit{rt}) \in \mathsf{MSigs}_C(C_{sup})}{\Gamma \vdash ((\mathit{meth}, \mathit{at}), \mathit{rt}) \in \mathsf{MSigs}_C(C)}$$