Type Elaboration and Subtype Completion for Java Bytecode

TODD B. KNOBLOCK and JAKOB REHOF
Microsoft Research

Java source code is strongly typed, but the translation from Java source to bytecode omits much of the type information originally contained within methods. Type elaboration is a technique for reconstructing strongly typed programs from incompletely typed bytecode by inferring types for local variables. There are situations where, technically, there are not enough types in the original type hierarchy to type a bytecode program. Subtype completion is a technique for adding necessary types to an arbitrary type hierarchy to make type elaboration possible for all verifiable Java bytecode. Type elaboration with subtype completion has been implemented as part of the Marmot Java compiler.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Compilers*; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs—*Type Structure*

General Terms: Languages, Theory

Additional Key Words and Phrases: Java compiler, lattice completion, object-oriented type systems, type-directed compilation, typed intermediate language, type inference, type reconstruction

1. INTRODUCTION

Type elaboration is a technique for type inference on verifiable Java bytecode. Verification provides type consistency rules that are based upon program flow and safety checking. These are weaker than the rules for static typechecking in Java source, and there are verifiable bytecode programs that are not typable under the Java typing rules. The central issue is that the type system of bytecode verification is based upon sets of types, and there may not be names for all of these sets in the Java type hierarchy. Subtype completion is a technique for adding a minimal number of new type names to make the bytecode typable.

The present work was motivated by our goal of using a strongly typed intermediate representation as part of the Marmot bytecode-to-native-code compiler [Fitzgerald et al. 2000]. Strongly typed programming languages have long been recognized as improving program correctness and enhancing efficient implementation. More recently, it has been observed that type-based intermediate representations and type-based compilation can extend these advantages to a compiler itself [Morrisett 1995; Tarditi 1996; Leroy and Ohori 1998]. The use of types provides benefits in

Authors address: Microsoft Research, Redmond, WA, 98052;
email {toddk, rehof}@microsoft.com.

debugging the compiler, as well as in optimization and garbage collection.

Many Java compilers use (or at least will accept) Java bytecode [Lindholm and Yellin 1999] instead of Java source programs as input [IBM 1998; Instantiations, Inc. 1998; NaturalBridge, LLC 1998; SuperCede, Inc. 1998]. However, some of the type information in the original Java source program is lost during the initial translation to bytecode. In order to use a typed intermediate representation, it is first necessary to reconstruct a strongly typed representation from the partially typed bytecode.

The type system that we have chosen as the basis of our typed intermediate representation is a relatively simple one that has several advantages. First, it is very similar to the type system of Java. Second, it does not require *sets of types* representations, as does bytecode verification. Third, it is easy for a human to read, verify, and comprehend the typings. Fourth, it arises in a principled way from the (implicit) type system of bytecode verification. Finally, it is efficient to typecheck a program under this type system.

Type elaboration is performed once, near the beginning of the compilation. After that, the fully typed program can be efficiently typechecked. In the standard mode of operation, the Marmot compiler will typecheck the program 10 times, and even on our largest benchmarks, each typecheck takes less than a second. During debugging, the system can perform dozens of typechecks in order to identify and isolate faults.

In addition to its utility in type-directed compilation, type elaboration for bytecode solves a practical problem for Java decompilers such as Mocha that attempt to reconstruct Java source from Java bytecode. Once again, the simplicity of the underlying type system and its similarity to the original Java type system are advantages for this problem, since the goal is to reconstruct the original program typings.

In order to describe type elaboration, it is necessary that we examine three languages: Java source code, Java bytecode, and a typed intermediate representation that we refer to as the Java Intermediate Representation, *JIR*. These three languages have distinct, but related type systems. The Java source language has a traditionally defined type system presented as part of the standard language definition [Gosling et al. 1996]. Bytecode, per se, is untyped (or partially typed), but the rules for bytecode verification provide further consistency requirements based on dataflow and safety checking. Finally, the type system for JIR is closely related to the original type system for Java, and can be given a conventional set of typing rules. This paper investigates the formal relationship between these three typed languages and their type systems. Our main technical contributions are:

(1) We present a practical algorithm for type elaboration that accepts any verifiable bytecode.

(2) We describe a technique called *subtype completion*, which transforms a subtyping system by conservatively extending its type hierarchy to a lattice. The technique is founded on the Dedekind-MacNeille completion, a mathematical technique for embedding a poset in a lattice.

(3) We formalize type-based safety checking present in Java bytecode verification. We show that subtype completion performed on a Java-like type system inserts exactly the extra types needed for verification, and it gives rise to a strongly

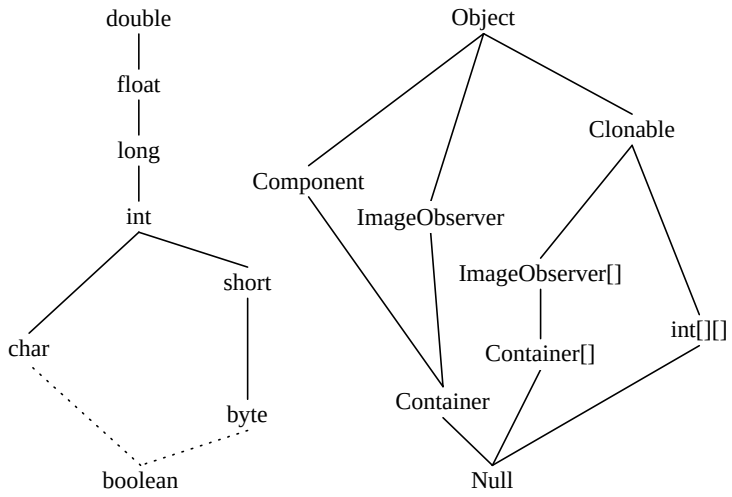

Fig. 1. Fragment of a Java type hierarchy.

typed intermediate representation (JIR) in a principled manner, together with a provably correct type inference algorithm.

The paper is organized as follows. In Sections 2 and 3, the types of bytecode, Java, and JIR are discussed. In Sections 4 through 6, we present an abstraction of the problem and present the technical results on subtype completion. In Section 7, we return from the abstract problem to the concrete problem, and describe a number of other issues that must be addressed in implementing type elaboration, along with some comments on complexity and performance. Finally, we survey related work in Section 8 and offer conclusions in Section 9.

2. TYPES IN JAVA BYTECODE

Java front-end compilers are responsible for translating Java source to bytecode. Java source code programs include complete static type information, but front-end compilers omit some of that information in the conversion to bytecode. The loss of type information in bytecode is apparent in several places:

—Local variables do not have type information.

—Evaluation stack locations are untyped.

—Values represented as *small integers* (defined as booleans, bytes, shorts, chars, and integers) are convolved within bytecode methods.

Separating the various types of small integers is especially useful because they are semantically distinct and valid in differing contexts. For example, while the representation of a boolean may be as another small integer type, boolean values should not be used in arithmetic expressions, and integer values should not be used where a boolean is expected.

Although bytecode has lost some of the original typing of the Java source, nevertheless, much of the original type information from the program source has been preserved:

```
interface SI {
    void siMeth();
}

interface SJ {
    void sjMeth();
}

interface I extends SI, SJ {
    ...
}

interface J extends SI, SJ {
    ...
}
```

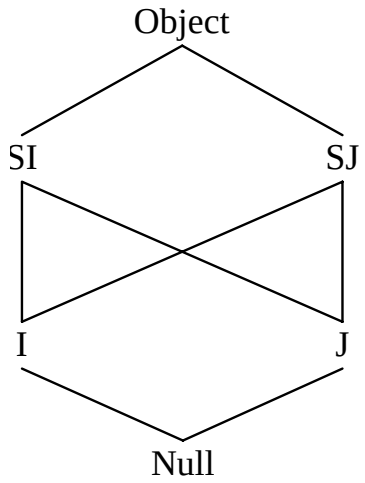

Fig. 2. Type hierarchy representing multiple inheritance using interfaces.

—All class fields maintain representations of the originally declared types.

—All function formals have types.

—The return value of a method, if any, has a type.

—Verified bytecode implies certain internal consistency in the use of types for locals and stack temporaries.

What is missing is most of the type information within a method.

The bytecode specification includes provisions for debug types on locals. This would appear to simplify at least part of the problem of elaboration. Unfortunately, (a) the debug types for locals are optional, (b) they do not distinguish between small integer types, (c) debug information is not available for the VM stack locations, and (d) they are wrong or incomplete in some bytecode files.

Type elaboration assumes verified bytecode as input, but does not depend upon the local variable debug information. If it is available, and deemed reliable, it is easy to employ. Bytecode verification assures various dynamic safety properties of the input program. It does not directly solve the problem of type elaboration because it does not distinguish between small integer types and handles multiple inheritance issues differently than the Java type system does.

3. THE TYPE SYSTEMS OF JAVA AND JIR

The type system of Java is defined, in part, by the *widening conversions* of the language [Gosling et al. 1996]. These encompass both the subtyping rules for reference types and the implicit coercions for numeric types.

All of the widening conversions may be combined to form a partial ordering of the types of Java. We write $A < B$ if there is a widening conversion from type A to type B. Figure 1 shows an example type hierarchy represented as a partial-order diagram.[1]

[1]In Java the type `boolean` may not be widened to any other type. However, to solve for the

```
void foo(boolean flag, I i, J j) {
    if (flag) {
        x = i;
    } else {
        x = j;
    }
    x.siMeth();
    x.sjMeth();
}
```

Fig. 3. Sample method that can not be typed in the given type hierarchy.

The type system of JIR is the same as that of Java except that all primitive numeric types are incomparable (i.e., there are no implicit representation-changing coercions) and the JIR system contains extra types in the subtype hierarchy.

Type hierarchies for Java programs, as in other object-oriented languages, are partial orders, but not necessarily lattices. Figure 2 contains the outline of four interface definitions that give rise to the fragment of a type hierarchy shown. In this hierarchy, there is no *least* upper bound of `I` and `J` and no *greatest* lower bound of `SI` and `SJ`.

Figure 3 contains a sample method (shown in pseudocode rather than bytecode for clarity). The task for type elaboration is to solve for the type of local variable `x` relative to the type hierarchy in Figure 2. We can see from the definitions of `x` that its type must be a supertype of both `I` and `J`, and further from the two uses, it must be a subtype of both `SI` and `SJ`.

This method is an interesting case in bytecode verification. Verification defines the merging of two occurrences of a local variable at join points (such as the control point succeeding the `if`) as containing "an instance of the first common superclass of the two types" [Lindholm and Yellin 1999, p. 146]. As has been noted by Qian [1998], Goldberg [1997], and others, it is not clear how this should be interpreted when the types in question are interfaces, especially when multiple inheritance is involved.[2]

There appears to be general agreement, both in the work on formalizing Java bytecode verification and in the actual implementations of bytecode verifiers, that this step in verification requires that sets of types be employed. Moreover, the merge of the type states at the join point is the union of the possible types. Under this interpretation, the method in Figure 3 is verifiable, but not typable in the given type hierarchy [Goldberg 1997; Coglio et al. 1998; Qian 1998; Pusch 1999; Gagnon and Hendren 1999].

small integer types, it is useful to posit widening conversions for `boolean` for use within (and only during) type elaboration.

[2] One strict reading of this language would have the first "superclass" of any interface be the `Object` type, as it is the only "class" that is a supertype of an interface. However, this reading would reject as unverifiable bytecode corresponding to legal Java programs. For example, consider having a method with a local declared as an `Enumeration`, an interface type, which is initialized in one branch of a conditional as a `VectorEnumeration`, a class implementing an enumeration for a vector, and in the other branch of the conditional as a `HashtableEnumeration`. After the join, under the strict reading, this variable could not be used as an `Enumeration`, but only as an `Object`.

The language of J

Local variables	z	
Parameters	x	
Field names	a	
Method names	f	
Constants	c	
Expressions	e	$::= c \mid x \mid z \mid e.a \mid e.f(e_1, \ldots, e_n)$
LExpressions	le	$::= z \mid e.a$
Statements	s	$::= le = e \mid \texttt{return}_f\ e \mid \texttt{if}(e)\ s_1\ \texttt{else}\ s_2 \mid \texttt{let}\ z = e\ \texttt{in}\ s \mid s_1; s_2$
Declarations	d	$::= \omega :: f(x_1 : \tau_1, \ldots, x_n : \tau_n)\{s\}$

The Types of J

Primitive types	π	
Reference types	ω	$::= null \mid \ldots$
Base types (T_0)	τ	$::= unit \mid \omega \mid \pi$
Types (T)	σ	$::= \tau \mid \omega.\tau \mid \vec{\tau} \rightarrow \tau'$

Fig. 4. Syntax of J and its types.

4. AN ABSTRACT TYPE SYSTEM FOR JAVA

In this section, we begin to study the problem of type elaboration in depth. We consider an abstract and simplified version of the problem which focuses on the core issue of how the three type systems (Java, bytecode, and JIR) are related. The typed language J is an abstraction of Java which captures the salient properties of the type elaboration problem. The syntax of the language J and its types are shown in Figure 4.

4.1 The Language J and Its Types

The language J defined in Figure 4 includes method declarations with typed parameters and assignable local variables introduced in `let`-statements. The types of J (ranged over by σ) are built from *base types* (ranged over by τ) which include the special type *unit* together with *reference types* (ranged over by ω) and *primitive types* (ranged over by π). Reference types include the special type *null* and object types. Primitive types include the boolean type and the numerical types of Java. Let T denote the set of all types, and let T_0 denote the set of base types. Base types are assumed to be organized as a poset $\mathcal{H} = \langle T_0, \leq \rangle$, which defines a *subtype* or *inheritance hierarchy*.

The expression form $e.a$ is field selection. A *field type* is a pair $\omega.\tau$, where ω is an object type in which the field exists, and τ the type of the field itself. The form $e.f(e_1, \ldots, e_n)$ is method invocation, and *method types* have the form $(\tau_1, \ldots, \tau_n) \rightarrow \tau'$. We assume that parameters, ranged over by x, include unique tokens $this_\omega$ for each reference type ω. Local variables are introduced using `let`-statements of the form $\texttt{let}\ z = e\ \texttt{in}\ s$. Method definitions are accommodated by

Expressions.

$$[\text{ Cns }]\frac{}{\Sigma; A \vdash c : \Sigma(c)} \qquad [\text{ Par }]\frac{}{\Sigma; A \vdash x : \Sigma(x)} \qquad [\text{ Var }]\frac{}{\Sigma; A \vdash z : A(z)}$$

$$[\text{ Sel }]\frac{\begin{array}{l}\Sigma(a) = \omega.\tau \\ \Sigma; A \vdash e : \omega' \\ \omega' \leq \omega\end{array}}{\Sigma; A \vdash e.a : \tau} \qquad [\text{ Inv }]\frac{\begin{array}{l}\Sigma(f) = (\tau_1, \ldots, \tau_n) \to \tau \\ \Sigma; A \vdash e_i : \tau_i' \\ \Sigma; A \vdash e : \omega \\ \tau_i' \leq \tau_i, \mathsf{sig}(\omega, f) \neq \emptyset\end{array}}{\Sigma; A \vdash e.f(e_1, \ldots, e_n) : \tau}$$

Statements.

$$[\text{ Asn }]\frac{\begin{array}{l}\Sigma; A \vdash le : \tau \\ \Sigma; A \vdash e : \tau' \\ \tau' \leq \tau\end{array}}{\Sigma; A \vdash le = e : \mathit{unit}} \qquad [\text{ Ret }]\frac{\begin{array}{l}\Sigma(f) = \vec{\tau} \to \tau' \\ \Sigma; A \vdash e : \tau'' \\ \tau'' \leq \tau'\end{array}}{\Sigma; A \vdash \mathtt{return}_f\ e : \mathit{unit}}$$

$$[\text{ Cnd }]\frac{\begin{array}{l}\Sigma; A \vdash e : \tau \\ \Sigma; A \vdash s_1 : \mathit{unit} \\ \Sigma; A \vdash s_2 : \mathit{unit} \\ \tau \leq \mathit{boolean}\end{array}}{\Sigma; A \vdash \mathtt{if}(e)\ s_1\ \mathtt{else}\ s_2 : \mathit{unit}} \qquad [\text{ Let }]\frac{\begin{array}{l}\Sigma; A \vdash e : \tau \\ \Sigma; A, z : \tau' \vdash s : \mathit{unit} \\ \tau \leq \tau'\end{array}}{\Sigma; A \vdash \mathtt{let}\ z = e\ \mathtt{in}\ s : \mathit{unit}}$$

$$[\text{ Cmp }]\frac{\begin{array}{l}\Sigma; A \vdash s_1 : \mathit{unit} \\ \Sigma; A \vdash s_2 : \mathit{unit}\end{array}}{\Sigma; A \vdash s_1; s_2 : \mathit{unit}}$$

Declarations.

$$[\text{ Dcl }]\frac{\begin{array}{l}\Sigma(f) = (\tau_1 \ldots, \tau_n) \to \tau' \\ \Sigma(\mathit{this}_\omega) = \omega \\ \Sigma(x_i) = \tau_i, i = 1 \ldots n \\ \Sigma; A \vdash s : \mathit{unit}\end{array}}{\Sigma; A \vdash \omega :: f(x_1 : \tau_1, \ldots, x_n : \tau_n)\{s\} : \mathit{unit}}$$

Fig. 5. System J.

declarations $\omega :: f(x_1, \ldots, x_n)\{s\}$, which define a method f with body s, within the reference type ω. A `return`-statement is assumed to be tagged with the name of the method in whose declaration it occurs; any `return` statement in the declaration of method f must have the form $\mathtt{return}_f\ e$. A *phrase*, M, is either a statement, a declaration, or an expression.

Type system J. The type system J is defined in Figure 5. These rules define derivable typing judgments of the form $\Sigma; A \vdash M : \tau$. The intended reading of such a judgment is that, under the typing assumptions given by the *signature* Σ and the *type environment* A, the phrase M has type τ. A signature is a function mapping field names, method names, parameters, and constants to types. For M to be typable, all the field names, method names, and constants occurring in M must be given types by Σ. The signature is intended to model declared types and the types of the basic constants of the language, which include predefined functions, such as arithmetic functions. It follows that only local (`let`-bound) variables have no declared types. A type environment, A, is a set of type assumptions of the form $z : \tau$, which assigns type τ to local variable z. Only one assumption may occur for

a given variable.[3] The rules of Figure 5 are parametric in a given subtype hierarchy $\mathcal{H}$ and the signature Σ.

We need to require for a method name f that it is appropriately declared in the hierarchy of reference types. For this purpose, let $\mathsf{Decl}(\omega)$ denote the set of method names f such that f is declared as a method of ω, with ω a reference type in $\mathcal{H}$. Furthermore, define

$$\mathsf{sig}(\omega, f) = \{\omega' \in \mathcal{H} \mid \omega \leq \omega', f \in \mathsf{Decl}(\omega')\}.$$

In other words, $\mathsf{sig}(\omega, f)$ denotes the set of all reference types above ω in $\mathcal{H}$ that declare a method named f. Whenever a method name f is used, we can now require that it must have a declared type by requiring $\mathsf{sig}(\omega, f) \neq \emptyset$. Also, by suitably renaming method names, we can assume without loss of generality that every method name f can be given a single declared type, $\Sigma(f)$. For a set S of reference types we define

$$\mathsf{sig}(S, f) = \bigcap_{\omega \in S} \mathsf{sig}(\omega, f).$$

Notice that under this definition one has $S \subseteq S' \Rightarrow \mathsf{sig}(S', f) \subseteq \mathsf{sig}(S, f)$.

We let $J(\Sigma, \mathcal{H})$ denote the system obtained by using a specific hierarchy $\mathcal{H}$ and signature Σ in the rules for system J.

The *type inference problem* for J is to reconstruct types for the local variables of a program in such a way that the program is well typed according to the rules of Figure 5. More precisely, given a phrase M, a signature Σ, and a hierarchy $\mathcal{H}$, the type inference problem is to decide whether there exists a type τ and an assignment A of types to the locals of M such that $\Sigma; A \vdash M : \tau$. Note that because $\mathcal{H}$ can contain arbitrary finite subposets of interface hierarchies, the type inference problem for system J is NP-complete by reduction from satisfiability of inequalities over posets. The reduction follows from previous results on the complexity of subtype inference [Lincoln and Mitchell 1992; Tiuryn 1992; Pratt and Tiuryn 1996; Benke 1993; Hoang and Mitchell 1995], and a proof of NP-completeness for a framework similar to ours can be found in Gagnon and Hendren [1999].

5. BYTECODE VERIFICATION

This section formalizes the relevant part of the rules of Java bytecode verification as they apply to our intermediate language J. It takes the form of a system of flow constraints generated from a program and a set of safety-checking rules. The intention is that a program passes the verifier if and only if the safety-checking rules are satisfied by the least solution to the flow-system generated from the program.

5.1 Flow System

The flow system shown in Figure 6 conservatively approximates the set of types of the values that a subexpression can evaluate to. The system of constraints generated from phrase M is denoted $\mathcal{F}[\![M]\!]$. For the purpose of defining the flow

[3]In order to apply the present framework to a program, the program's variables, field names, method names, and parameters may have to be renamed appropriately. We tacitly assume that this has been done.

$$
\begin{array}{lcl}
\mathcal{F}[\![c]\!] & = & \{\mathcal{X}_c = \Sigma(c)\} \\
\mathcal{F}[\![x]\!] & = & \{\mathcal{X}_x = \Sigma(x)\} \\
\mathcal{F}[\![e.a]\!] & = & \{\mathcal{X}_{e.a} = \tau\} \cup \mathcal{F}[\![e]\!] \\
 & & \quad \text{where } \Sigma(a) = \omega.\tau \\
\mathcal{F}[\![e.f(e_1,\ldots,e_n)]\!] & = & \{\mathcal{X}_{e.f(e_1,\ldots,e_n)} = \tau'\} \cup (\bigcup_{i=1}^{n} \mathcal{F}[\![e_i]\!]) \\
 & & \quad \text{where } \Sigma(f) = \vec{\tau} \to \tau' \\
\mathcal{F}[\![z = e]\!] & = & \{\mathcal{X}_e \subseteq \mathcal{X}_z\} \cup \mathcal{F}[\![e]\!] \\
\mathcal{F}[\![\texttt{return}_f\ e]\!] & = & \mathcal{F}[\![e]\!] \\
\mathcal{F}[\![\texttt{if}(e)\ s_1\ \texttt{else}\ s_2]\!] & = & \mathcal{F}[\![e]\!] \cup \mathcal{F}[\![s_1]\!] \cup \mathcal{F}[\![s_2]\!] \\
\mathcal{F}[\![\texttt{let}\ z = e\ \texttt{in}\ s]\!] & = & \{\mathcal{X}_e \subseteq \mathcal{X}_z\} \cup \mathcal{F}[\![e]\!] \cup \mathcal{F}[\![s]\!] \\
\mathcal{F}[\![s_1; s_2]\!] & = & \mathcal{F}[\![s_1]\!] \cup \mathcal{F}[\![s_2]\!] \\
\mathcal{F}[\![\omega :: f(x_1 : \tau_1,\ldots,x_n : \tau_n)\{s\}]\!] & = & (\bigcup_{i=1}^{n}\{\mathcal{X}_{x_i} = \tau_i\}) \cup \{\mathcal{X}_{this_\omega} = \omega\} \cup \mathcal{F}[\![s]\!] \\
 & & \quad \text{where } \Sigma(f) = (\tau_1,\ldots,\tau_n) \to \tau' \\
 & & \quad \text{and } \Sigma(x_i) = \tau_i, i = 1\ldots n
\end{array}
$$

Fig. 6. Type-flow constraints for system J.

system, we assume that each variable in the program has been renamed, so that the names are distinct. For each distinct occurrence of a subexpression e in the program, where e is not a local variable and not a parameter, the system uses a distinct flow variable named $\mathcal{X}_e$ to describe the possible types of e. Moreover, for each local variable z and each parameter x there will be a unique flow variable $\mathcal{X}_z$, respectively $\mathcal{X}_x$. Flow variables range over finite subsets of T_0. All constraints take the form $\mathcal{X}_e \subseteq \mathcal{X}_{e'}$ or $\mathcal{X}_e = \tau$. The latter form is a shorthand for $\mathcal{X}_e = \{\tau\}$.

The flow rules shown in Figure 6 should be understood in conjunction with the verification rules shown in Figure 7. Notice that the flow rules of Figure 6 exploit type declarations to localize the flow computation as much as possible. For example, because every field name a is assumed to have a declared type $\Sigma(a)$, there is no flow rule for an assignment $e.a = e'$. Instead, the flow variable $\mathcal{X}_{e.a}$ gets the singleton value of the declared field type, τ, where $\Sigma(a) = \omega.\tau$. The verification rules, however, will contain a check to make sure that, for an expression of the form $e.a = e'$, the flow value associated with e' is consistent with the declared type τ (see rule $(S2)$ of Figure 7).

LEMMA 5.1. *For every phrase M, the constraint system $\mathcal{F}[\![M]\!]$ has a least solution.*

PROOF. See Appendix A. □

5.2 Verification Rules

For a given phrase M, let $\widehat{\mathcal{X}}_e \subseteq T_0$ denote the meaning of flow variable $\mathcal{X}_e$ under the least solution to $\mathcal{F}[\![M]\!]$. Then the verification rules for M are as defined in

(S1) For every occurrence of a subexpression of the form $e.a$:
check that $\widehat{\mathcal{X}}_e \sqsubseteq \omega$, where $\Sigma(a) = \omega.\tau$
(S2) For every occurrence of a statement of the form $e.a = e'$:
check that $\widehat{\mathcal{X}}_{e'} \sqsubseteq \tau$, where $\Sigma(a) = \omega.\tau$
(S3) For every occurrence of a subexpression of the form $e.f(e_1, \ldots, e_n)$:
check that $\mathsf{sig}(\widehat{\mathcal{X}}_e, f) \neq \emptyset$ and $\widehat{\mathcal{X}}_{e_i} \sqsubseteq \tau_i$, where $\Sigma(f) = (\tau_1, \ldots, \tau_n) \to \tau'$
(S4) For every occurrence of a statement of the form $\texttt{return}_f\ e$:
check that $\widehat{\mathcal{X}}_e \sqsubseteq \tau'$, where $\Sigma(f) = \vec{\tau} \to \tau'$
(S5) For every occurrence of a statement of the form $\texttt{if}(e)\ s_1\ \texttt{else}\ s_2$:
check that $\widehat{\mathcal{X}}_e \sqsubseteq \mathit{boolean}$

Fig. 7. Verification rules for system J.

Figure 7. These rules use the notation $S \sqsubseteq \tau$, where S is a subset of T_0, defined by setting $S \sqsubseteq \tau$ iff $\forall \tau' \in S.\ \tau' \leq \tau$. If the least solution to $\mathcal{F}[\![M]\!]$ satisfies the verification rules of Figure 7 using signature Σ and hierarchy $\mathcal{H}$, then we say that M is *safe with respect to* Σ *and* $\mathcal{H}$.

LEMMA 5.2. *If M is typable in system $J(\Sigma, \mathcal{H})$, then M is safe with respect to Σ and $\mathcal{H}$.*

PROOF. By induction on the proof that M types in system $J(\Sigma, \mathcal{H})$. □

Note that verification accepts more programs than system J. Consider the method `foo` shown in Figure 3. It cannot be typed in System J, because the conditional requires `x` to have a type larger than both I and J, and the method invocations require `x` to have a type smaller than both SI and SJ; but no such type exists in the hierarchy shown in Figure 2. However, since $\widehat{\mathcal{X}}_x = \{I, J\}$ holds for the least solution to $\mathcal{F}[\![\texttt{foo}]\!]$, we have $\widehat{\mathcal{X}}_x \sqsubseteq SI$ and $\widehat{\mathcal{X}}_x \sqsubseteq SJ$, and hence it is easy to see that `foo` satisfies the verification rules of Figure 7.

6. SUBTYPE COMPLETION AND THE JIR TYPE SYSTEM

We will now show how the type system of JIR emerges by a *completion* of system J. The resulting JIR type system is a conservative extension of system J, which accepts exactly the verifiable (safe) programs. The construction is an application of the Dedekind-MacNeille completion known from lattice theory [MacNeille 1937; Birkhoff 1995; Davey and Priestley 1990]. Intuitively, this completion technique enriches the hierarchy $\mathcal{H}$ to a *minimal* lattice, by inserting missing least upper bounds and greatest lower bounds into $\mathcal{H}$. Minimality means that the completion inserts new elements only where necessary.

Other completion methods are known, such as *powerset completion* and *ideal completion* [Davey and Priestley 1990]. Moreover, the set-based flow-system of Section 5 is of course sufficient for verification. However, these methods introduce "unnecessary" elements. Consider a hierarchy P with types $\{\bot, A, B, C, D, \top\}$, ordered by $x \leq \top$ and $\bot \leq x$ for all x in $\{A, B, C, D\}$. Ideal completion will include a distinct element for each of the 16 subsets of $\{A, B, C, D\}$. Each such element represents the least upper bound of the types in the subset. Flow-based safety checking may consider any subset of P. In contrast, Dedekind-MacNeille

completion inserts no elements at all, since P is already a lattice and hence the minimal lattice containing itself.

The result that type elaboration captures bytecode verification shows that, even though flow-based safety checking may consider many more sets than are produced by completion, the distinctions that can be made using these "extra" sets are irrelevant for safety.

6.1 The Dedekind-MacNeille Completion

Let $\langle P, \leq \rangle$ be an arbitrary poset. The Dedekind-MacNeille completion of P, denoted $\mathsf{DM}(P)$, is the least complete lattice containing P as an isomorphic subposet. An *order ideal* is a downward closed subset of P, and the *principal ideal* generated from an element, x, is defined as $\downarrow x = \{y \in P \mid y \leq x\}$. The poset P is contained in $\mathsf{DM}(P)$ by the embedding $x \mapsto \downarrow x$. In particular, $\mathsf{DM}(P)$ preserves existing joins and meets in P.[4] We proceed to outline how $\mathsf{DM}(P)$ is constructed from P. If $A \subseteq P$ and $x \in P$ we write $x \leq A$ if and only if $x \leq y$ for all $y \in A$, and we write $A \leq x$ if and only if $y \leq x$ for all $y \in A$. Define the sets A^u and A^ℓ as

$$A^u = \{x \in P \mid A \leq x\} \text{ and } A^\ell = \{x \in P \mid x \leq A\}.$$

Define the operator C by $C(A) = A^{u\ell}$; then one has the basic properties $A \subseteq C(A)$, $A \subseteq B$ implies $C(A) \subseteq C(B)$, and $C(C(A)) = C(A)$. Now define the family of subsets $\mathsf{DM}(P)$ by setting

$$\mathsf{DM}(P) = \{A \subseteq P \mid C(A) = A\},$$

i.e., $\mathsf{DM}(P)$ is the family of all subsets of P that are closed with respect to the operator C, ordered by set inclusion. Meet and join in $\mathsf{DM}(P)$ are given by

$$\bigwedge_i A_i = \bigcap_i A_i \text{ and } \bigvee_i A_i = C(\bigcup_i A_i) \tag{1}$$

where the operator C is as defined above. All elements of $\mathsf{DM}(P)$ are order ideals, but not every ideal is an element of $\mathsf{DM}(P)$.

6.2 The JIR Type System

We now show how the type system of JIR arises from that of $J(\Sigma, \mathcal{H})$ by applying the Dedekind-MacNeille completion to J. Its definition is given in Figure 8. This system is isomorphic to system $J(\Sigma, \mathcal{H})$ (Figure 5), in which the type structure has been reinterpreted over $\mathsf{DM}(\mathcal{H})$. Its base types are just the elements of $\mathsf{DM}(\mathcal{H})$, ranged over by I. We construct method types $\vec{I} \to I'$ by setting

$$\vec{I} \to I' = \{(\tau_1, \ldots, \tau_n) \to \tau' \mid \tau_i \in I_i, \tau' \in I'\},$$

and for a set $\Omega \in \mathsf{DM}(\mathcal{H})$ consisting of object types and a base type element I we define the field type $\Omega.I$ by setting

$$\Omega.I = \{\omega.\tau \mid \omega \in \Omega, \tau \in I\}.$$

[4]Hence, if P is already a lattice, then DM acts as the identity, modulo isomorphism.

Expressions.

$$[\text{ Cns }]\frac{}{\Sigma; A \vdash c : \Sigma(c)} \qquad [\text{ Par }]\frac{}{\Sigma; A \vdash x : \Sigma(x)} \qquad [\text{ Var }]\frac{}{\Sigma; A \vdash z : A(z)}$$

$$[\text{ Sel }]\frac{\begin{array}{l}\Sigma(a) = \Omega.I \\ \Sigma; A \vdash e : \Omega' \\ \Omega' \leq \Omega\end{array}}{\Sigma; A \vdash e.a : I} \qquad [\text{ Inv }]\frac{\begin{array}{l}\Sigma(f) = (I_1, \ldots, I_n) \to I \\ \Sigma; A \vdash e_i : I_i' \\ \Sigma; A \vdash e : \Omega \\ I_i' \leq I_i, \mathsf{sig}(\Omega, f) \neq \emptyset\end{array}}{\Sigma; A \vdash e.f(e_1, \ldots, e_n) : I}$$

Statements.

$$[\text{ Asn }]\frac{\begin{array}{l}\Sigma; A \vdash le : I \\ \Sigma; A \vdash e : I' \\ I' \leq I\end{array}}{\Sigma; A \vdash le = e : \mathit{Unit}} \qquad [\text{ Ret }]\frac{\begin{array}{l}\Sigma(f) = \vec{I} \to I' \\ \Sigma; A \vdash e : I'' \\ I'' \leq I'\end{array}}{\Sigma; A \vdash \mathtt{return}_f\ e : \mathit{Unit}}$$

$$[\text{ Cnd }]\frac{\begin{array}{l}\Sigma; A \vdash e : I \\ \Sigma; A \vdash s_1 : \mathit{Unit} \\ \Sigma; A \vdash s_2 : \mathit{Unit} \\ I \leq \mathit{Boolean}\end{array}}{\Sigma; A \vdash \mathtt{if}(e)\ s_1\ \mathtt{else}\ s_2 : \mathit{Unit}} \qquad [\text{ Let }]\frac{\begin{array}{l}\Sigma; A \vdash e : I \\ \Sigma; A, z : I' \vdash s : \mathit{Unit} \\ I \leq I'\end{array}}{\Sigma; A \vdash \mathtt{let}\ z = e\ \mathtt{in}\ s : \mathit{Unit}}$$

$$[\text{ Cmp }]\frac{\begin{array}{l}\Sigma; A \vdash s_1 : \mathit{Unit} \\ \Sigma; A \vdash s_2 : \mathit{Unit}\end{array}}{\Sigma; A \vdash s_1; s_2 : \mathit{Unit}}$$

Declarations.

$$[\text{ Dcl }]\frac{\begin{array}{l}\Sigma(f) = (I_1, \ldots, I_n) \to I' \\ \Sigma(\mathit{this}_\Omega) = \Omega \\ \Sigma(x_i) = I_i, i = 1 \ldots n \\ \Sigma; A \vdash s : \mathit{Unit}\end{array}}{\Sigma; A \vdash \Omega :: f(x_1 : I_1, \ldots, x_n : I_n)\{s\} : \mathit{Unit}}$$

Fig. 8. The JIR type system.

We translate types σ of J to types $dm(\sigma)$ of JIR by taking principal ideals of base types.[5]

We write $\mathit{Boolean} = \downarrow\mathit{boolean}$, $\mathit{Unit} = \downarrow\mathit{unit}$ and $\mathit{Null} = \downarrow\mathit{null}$. A signature Σ is then translated to the signature $\mathsf{DM}(\Sigma) = dm \circ \Sigma$. The resulting system is denoted $\mathsf{JIR}(\mathsf{DM}(\Sigma), \mathsf{DM}(\mathcal{H}))$.

The order relation in Figure 8 can be thought of in two (isomorphic) ways: as set inclusion, or as an extension of the order on $\mathcal{H}$. It is useful to stress the latter view in the context of *decompilation*, where we desire typings that are as close as possible to the type system J. The completion $\mathsf{DM}(\mathcal{H})$ is instrumental to this by inserting only a minimal number of new points into $\mathcal{H}$. In the example of Figure 2, this amounts to inserting a new type, `IJ`, which is the least upper bound of `I` and `J` and the greatest lower bound of `SI` and `SJ`.

[5]In detail, let $dm(\tau) = \downarrow\tau$, $dm(\omega.\tau) = (\downarrow\omega).(\downarrow\tau)$, and $dm((\tau_1, \ldots, \tau_n) \to \tau) = (\downarrow\tau_1, \ldots, \downarrow\tau_n) \to (\downarrow \tau)$.

$$
\begin{array}{lcl}
\mathcal{I}[\![c]\!] & = & \{\alpha_c = \Sigma(c)\} \\
\mathcal{I}[\![x]\!] & = & \{\alpha_x = \Sigma(x)\} \\
\mathcal{I}[\![e.a]\!] & = & \{\alpha_e \le \Omega, \alpha_{e.a} = I\} \cup \mathcal{I}[\![e]\!] \\
 & & \quad \text{where } \Sigma(a) = \Omega.I \\
\mathcal{I}[\![e.f(e_1,\ldots,e_n)]\!] & = & \{\alpha_{e.f(e_1,\ldots,e_n)} = I, \mathsf{sig}(\alpha_e, f) \neq \emptyset\} \cup \\
 & & (\bigcup_{i=1}^{n} \mathcal{I}[\![e_i]\!]) \cup \\
 & & (\bigcup_{i=1}^{n} \{\alpha_{e_i} \le I_i\}) \\
 & & \quad \text{where } \Sigma(f) = (I_1,\ldots,I_n) \to I \\
\mathcal{I}[\![le = e]\!] & = & \{\alpha_e \le \alpha_{le}\} \cup \mathcal{I}[\![le]\!] \cup \mathcal{I}[\![e]\!] \\
\mathcal{I}[\![\texttt{return}_f\ e]\!] & = & \{\alpha_e \le I'\} \cup \mathcal{I}[\![e]\!] \\
 & & \quad \text{where } \Sigma(f) = \vec{I} \to I' \\
\mathcal{I}[\![\texttt{if}(e)\ s_1\ \texttt{else}\ s_2]\!] & = & \{\alpha_e \le \mathit{Boolean}\} \cup \mathcal{I}[\![e]\!] \cup \mathcal{I}[\![s_1]\!] \cup \mathcal{I}[\![s_2]\!] \\
\mathcal{I}[\![\texttt{let}\ z = e\ \texttt{in}\ s]\!] & = & \{\alpha_e \le \alpha_z\} \cup \mathcal{I}[\![e]\!] \cup \mathcal{I}[\![s]\!] \\
\mathcal{I}[\![s_1; s_2]\!] & = & \mathcal{I}[\![s_1]\!] \cup \mathcal{I}[\![s_2]\!] \\
\mathcal{I}[\![\Omega :: f(x_1 : I_1,\ldots,x_n : I_n)\{s\}]\!] & = & \{\alpha_{\mathit{this}_\Omega} = \Omega\} \cup \\
 & & (\bigcup_{i=1}^{n} \{\alpha_{x_i} = I_i\}) \cup \mathcal{I}[\![s]\!] \\
 & & \quad \text{where } \Sigma(f) = (I_1,\ldots,I_n) \to I' \\
 & & \quad \text{and } \Sigma(x_i) = I_i, i = 1\ldots n
\end{array}
$$

Fig. 9. Type constraints for JIR.

6.3 Safety, Typability, and Type Inference

This section contains our main theoretical results on typability and type inference for JIR. We characterize the type inference problem for JIR, and show that the type system of JIR exactly captures the notion of safety of bytecode verification.

Type elaboration for JIR reconstructs types for local (`let`-bound) variables. Type elaboration can be performed by solving a system of equalities and inequalities between types and type variables generated from the subexpressions of a given program, as specified in Figure 9. The constraint system generated from the phrase M is denoted $\mathcal{I}[\![M]\!]$. Type variables $\alpha \in \mathit{TyVar}$ range over elements of $\mathsf{DM}(\mathcal{H})$. The constraint generation rules are given in terms of a signature Σ. Constraints take the form $\xi \le \xi'$, where ξ and ξ' are either a constant from $\mathsf{DM}(\mathcal{H})$ or a type variable, α.

The following lemma records the fact that typability in the JIR type system is exactly captured by satisfiability of the constraint systems $\mathcal{I}[\![M]\!]$.

LEMMA 6.1. *A program M is typable in $\mathsf{JIR}(\mathsf{DM}(\Sigma), \mathsf{DM}(\mathcal{H}))$ if and only if the system $\mathcal{I}[\![M]\!]$ is satisfiable in $\mathsf{DM}(\mathcal{H})$. Moreover, every solution to $\mathcal{I}[\![M]\!]$ corresponds to a valid typing derivation in $\mathsf{JIR}(\mathsf{DM}(\Sigma), \mathsf{DM}(\mathcal{H}))$.*

PROOF. The constraint system is a close reformulation of the typing rules. A detailed proof can be constructed along the lines of Theorem 2.1 in Wand [1987]

and Kozen et al. [1994], and is omitted. □

Our next theorem establishes that the type system of JIR exactly captures the bytecode verification system defined in Section 5. The proof essentially consists of showing that the least solution to $\mathcal{F}[\![M]\!]$ can be translated to a minimal solution to $\mathcal{I}[\![M]\!]$, provided that M is safe, and, conversely that a minimal solution to $\mathcal{I}[\![M]\!]$ can be translated to the least solution to $\mathcal{F}[\![M]\!]$ such that the solution satisfies the verification rules.

THEOREM (SOUNDNESS AND COMPLETENESS). *A program M is typable in system $\mathsf{JIR}(\mathsf{DM}(\Sigma), \mathsf{DM}(\mathcal{H}))$ if and only if M is safe with respect to Σ and $\mathcal{H}$.*

PROOF. See Appendix A for the proof. □

We now consider type inference. By Lemma 6.1, type inference for JIR reduces to solving type constraints over the lattice $\mathsf{DM}(\mathcal{H})$. Our next theorem characterizes the least solution to a satisfiable type constraint system. The theorem yields a low-order polynomial type inference algorithm, and it is the foundation for the JIR type inference algorithm used in type elaboration.

Note that all type constants in $\mathcal{I}[\![M]\!]$ are principal ideals, of the form $\downarrow\tau$. To solve the constraints, it is not necessary to actually form these ideals, because we can represent a principal ideal, $\downarrow\tau$, by its generator, τ. Accordingly, we define a translation $\lceil\bullet\rceil$ on types, given by $\lceil\downarrow\tau\rceil = \tau$, $\lceil\alpha\rceil = \alpha$, and we lift the translation to constraint sets C of the form $\mathcal{I}[\![M]\!]$ by defining $\lceil C\rceil = \{\lceil\tau\rceil \leq \lceil\tau'\rceil \mid \tau \leq \tau' \in C\}$. If $C = \mathcal{I}[\![M]\!]$, and α is a variable in C, define the set D_α by setting

$$D_\alpha = \{\tau \in \mathcal{H} \mid \tau \leq \alpha \in \lceil C\rceil^*\}$$

where $\lceil C\rceil^*$ is the transitive closure of $\lceil C\rceil$.

THEOREM 6.3. *Let $C = \mathcal{I}[\![M]\!]$ and assume that C is satisfiable. Then there is a unique least solution μ to C in $\mathsf{DM}(\mathcal{H})$, which is given by*

$$\mu(\alpha) = (D_\alpha)^{u\ell} = \left(\bigcap_{\tau\in D_\alpha} \uparrow\tau\right)^{\ell}.$$

PROOF. See Appendix A for the proof. □

Even though $\mathsf{DM}(\mathcal{H})$ may be exponentially large in the size of $\mathcal{H}$, the solution formula in Theorem 6.3 only relies on types which are present in the constraint set. Therefore, solving a system $\mathcal{I}[\![M]\!]$ derived from a program M constructs only the types in $\mathsf{DM}(\mathcal{H})$ which are necessary for typing the particular program M. One can regard this as a kind of *lazy completion*, which avoids the exponential blow up; Theorem 6.3 gives rise to a polynomial time type inference algorithm. More details on how type elaboration is implemented are given in Section 7.

We will show one more property of $\mathsf{DM}(\mathcal{H})$ in the following lemma. It provides an optimization of the formula in Theorem 6.3, and it gives a succinct representation of the sets produced by completion. Whenever a set in $\mathsf{DM}(\mathcal{H})$ is isomorphic to an element in $\mathcal{H}$, the representation gives back that element automatically. It is therefore the basis of "decompiling" the types of $\mathsf{DM}(\mathcal{H})$ back to the types of $\mathcal{H}$.

To state the lemma, we give a few definitions. For subsets A and B of $\mathcal{H}$, we define the relation $\preceq$ by

$$A \preceq B \text{ iff } \forall x \in B.\ \exists y \in A.\ y \leq x$$

where the relation $\leq$ is the order relation of $\mathcal{H}$. If A is a subset of $\mathcal{H}$ and $x \in A$, then x is called a minimal element of A iff $y = x$ for any element $y \in A$ with $y \leq x$. Let $Min\, A$ denote the set of minimal elements of A.

LEMMA 6.4. *Let A and B be subsets of the poset $\mathcal{H}$. If $\mathcal{H}$ has no infinite descending chains, then one has*

$$A^{u\ell} \subseteq B^{u\ell} \Leftrightarrow \mathrm{Min}\,(A^u) \preceq \mathrm{Min}\,(B^u).$$

PROOF. See Appendix A for the proof. □

The previous lemma is applicable to solution sets of the form $(D_\alpha)^{u\ell}$ from Theorem 6.3. The lemma shows that the function that maps a set of the form $A^{u\ell}$ to the set $Min\,(A^u)$ is an order isomorphism.[6] We can therefore represent (more efficiently) the operation $\bullet^\ell$ by the operation Min. Moreover, since $\phi(\tau) = \downarrow\tau$ embeds $\mathcal{H}$ into $\mathsf{DM}(\mathcal{H})$, we know that

$$\phi^{-1}(A^{u\ell}) = \tau \Leftrightarrow Min\,(A^u) = \{\tau\} \tag{2}$$

(if $A^{u\ell} = \downarrow\tau$, then $A^u = A^{u\ell u} = \uparrow\tau$; hence $Min\,(A^u) = \{\tau\}$. Conversely, if $Min\,(A^u) = \{\tau\}$, then $A^{u\ell} = \{\tau\}^\ell = \downarrow\tau$). This shows that the representation given by Lemma 6.4 automatically maps elements of $\mathsf{DM}(\mathcal{H})$ back to $\mathcal{H}$, whenever this is possible. The application of Lemma 6.4 in type elaboration is discussed in more detail in Section 7.

7. IMPLEMENTING TYPE ELABORATION

Type elaboration has been implemented as part of the Marmot optimizing compiler [Fitzgerald et al. 2000]. This section describes how type elaboration was implemented for the full Java bytecode. In addition to the issues formalized in the abstract system of the last few sections, a number of other, more pragmatic issues must be addressed, including that of typing small integer variables and the Java definition of covariant array subtyping.

7.1 Preliminary Processing

Because the stack-based bytecode is not a convenient compiler intermediate form for reasons beyond type elaboration, bytecode is first converted to a conventional temporary-variable based intermediate form, JIR. In JIR, references to the interpreter stack have been replaced by explicit temporaries which may be treated as normal local variables.[7]

It is legal in bytecode for a single local variable to hold values of distinct types at different places in the method. For example, `local_3` may be used as both an

[6] This map is indeed a function: let $f(X) = Min\,(X^u)$, for any $X \subseteq \mathcal{H}$. Then f is a well-defined function, and one has $f(A^{u\ell}) = Min\,(A^{u\ell u}) = Min\,(A^u)$, by the identity $A^{u\ell u} = A^u$.

[7] While it is convenient to have unique names for variables during type elaboration, it would be possible to modify the type elaboration algorithms to work directly on the Java bytecode. Stack locations would be identified by both stack depth and program point.

`int` and as an `Object` within a single method. In verifiable bytecode, this is valid only if the lifetimes of the two uses of the local do not overlap (ignoring subroutines for the moment). Because type elaboration is required to assign a single type to each variable, it is necessary to separate any ambiguous uses of locals. This can be accomplished by renaming all uses of variables with distinct lifetimes. In Marmot, this is accomplished as a by-product of converting to SSA form [Cytron et al. 1989]. SSA form has the property that all static assignments to a variable have unique names. In the current example, the first name might become `local_3'1` with type `int` and the second `local_3'2` with type `Object`.

Java bytecode includes instructions that support a form of lightweight *subroutines* which preserve the local variable context. Such subroutines may be used to represent the `finally` part of `try/finally` handlers. The verification rules for locals and their interactions with subroutines are complex. They allow, for example, multiple types for the same live local so long as that local is not referenced in the finally block. As Freund [1998] and O'Callahan [1999] have noted, the space savings from using bytecode subroutines does not appear to justify the substantial additional complexity.

In Marmot, we chose to eliminate these subroutines by inline expansion. Type elaboration could be made to support subroutines directly, and in fact, we prototyped the code to support them, but eventually decided that the simpler expedient of inlining them was the more elegant solution.

For the purposes of the following description of type elaboration, it is assumed that the bytecode input has been preprocessed such that all local variables and stack temporaries have been assigned designators, and all ambiguous uses with distinct lifetimes have been separated.

7.2 The Type Elaboration Algorithm

After preprocessing to JIR, all of the locals that did not have manifest types (either declared or credible debug information) in the bytecode are assigned unique type variables of the form α_n.

Step 1*: Constraint Collection.* Constraint collection proceeds on a per-method basis using constraint formation rules analogous to those given in Figure 9. Equality constraints, $x = y$, are represented as two inequality constraints, $x \leq y$ and $y \leq x$.

For the purpose of constraint collection, small integer constants are given the type of the smallest containing small integer type. The signatures for phi applications, inserted as part of the translation to SSA form, are types of the from $\alpha^n \rightarrow \alpha$ where n is the arity for the phi function, and α is a fresh type variable for each distinct phi.

Step 2*: Constraint Closure.* Java defines covariant subtyping for array types: by definition `A[]` $\leq$ `B[]` iff `A` $\leq$ `B` for all reference types `A` and `B`. This rule requires that additional constraints be added to the constraint set in a process called *constraint closure*. Whenever a constraint is established between two potential array types, another constraint is induced between their element types. For example, if `A[]` $\leq$ `B[]` is established, then `A` $\leq$ `B` is also added to the constraint set. Further, recursively, if either `A` or `B` is a potential array type, then a constraint relating their element types is added. A *potential array type* is defined to be an explicit array

type `A[]` or a type variable α that is related to (greater or less than or equal) to a potential array type.

To relate a type variable's element type to a potential array type, say `A[]`, a fresh type variable is created, and then $\alpha = \alpha_{\text{elt}}\texttt{[]}$ is added to the constraints. Then $\texttt{A} \leq \alpha_{\text{elt}}$ or $\texttt{A} \geq \alpha_{\text{elt}}$, as appropriate, is added to the constraint set. This in turn may require further closure on the constraint set. If a potential array eventually turns out not to be an array, then the type variable introduced as its "element" type is disregarded.

The result of constraint collection and constraint closure is a finite set of constraints of the form $A \leq B$, which relate types and type variables as employed in the program. The next task is to *solve* the constraints, i.e., to find an assignment of the types for all type variables such that the constraints are satisfied.

Step 3*: Cycle Elimination.* Type elaboration first eliminates cycles in the constraint set by computing the strongly connected component of the constraints under the order relation [Tarjan 1972], and examines the acyclic directed hypergraph induced from the constraint graph by collapsing the nodes in a strongly connected component. Since our type structure is a partial order, all types within a strongly connected component, SCC, are equal, and all type variables in it will receive the same assignment in the solution. The resulting graph is called the *SCC graph* and represents a partial order, the *SCC order*.

The SCC graph is then traversed in depth-first order, and the types in each strongly connected component are computed using Theorem 6.3.

Step 4*: Constructing Filters.* In this step, order filters[8] of the original type hierarchy are used in the construction of solution types. This is done by computing the values $\bigcap_{\tau \in D_\alpha} \uparrow\tau$ for all variables α in the constraint graph, according to Theorem 6.3. Each such value is a subset of $\mathcal{H}$. Note that the intersection of filters is a filter. The filters are computed incrementally by a single depth-first traversal of the SCC graph, where the filter for each node is computed as follows:

(1) If the node contains a base type, T, the solution type is the principal filter generated from T. The principal filters may be precomputed and cached for each source type.

(2) If the node only contains type variables, the solution type is the intersection of the solution types of all immediate predecessors (lower elements in the SCC order).

(3) The solution computed by the previous two steps is cached at each node, so that it can be used to incrementally compute the solutions for its immediate successors in the SCC order in 2.

Step 5*: Constructing Types.* In this step, we compute minimal elements of the filter intersections computed in Step 4. This turns the filters computed in Step 4 into types (isomorphic to those) of $\mathsf{DM}(\mathcal{H})$ and at the same time maps them back to $\mathcal{H}$ whenever possible. This step is founded on Lemma 6.4, which allows us to

[8] An *order filter* is an upward closed subset of a poset. The *principal filter* generated from an element x is denoted $\uparrow x$.

represent the value A^{ℓ} by $MinA$, where A is a filter intersection. The type hierarchy $\mathcal{H}$ may have infinite descending chains due to the array type constructor:

$$\texttt{Object} \geq \texttt{Object[]} \geq \texttt{Object[][]} \geq \ldots$$

However, for any given program, the depth of array types will be bounded, and all descending chains will be finite for the section of $\mathcal{H}$ that matters for typing the given program. Lemma 6.4 is therefore applicable.

Step 6*: Applying the Solution.* One pragmatic issue is that it may not be possible to separate the uses of the small integers. A node in the SCC graph will normally contain at most one primitive type. For example, `Object` and `Clonable` would not be in the same SCC node in a verifiable program, since they are distinct in the type hierarchy partial order. However, it is possible for small integer types, e.g., `boolean` and `short`, to occur in the same node. This may happen because bytecode verification does not distinguish between these types, and so bytecode may legally use a value of one small integer type where the another is expected (e.g., a short `1` as a boolean condition).

This situation arises rarely in practice. When it does, all type calculations are based upon the join of the small integer types contained in the node. This is equivalent to equating, for example `boolean` and `short`, in the type hierarchy for the method being elaborated.

To complete type elaboration, the solution is recorded for each type variable. Further, any implied widening conversions that involve representational changes (e.g., `short` to `integer`) are made manifest by inserting explicit coercions in the JIR.

If the bytecode for a method convolves integer types in a way that causes a larger integer value to be used in a context expecting a smaller integer, then applying the solution will also introduce narrowing conversions for small integers. This is the only place where narrowing conversions (ones that lose information) are introduced by type elaboration.

A final pragmatic issue is that the precise definitions of classes and interfaces in the Java type system can, in rare instances, preclude inserting the desired completion point into the type hierarchy. Figure 10 shows a problematic case. This figure shows a crown-like class hierarchy with a combination of classes and interfaces. The point `BD` is a completion type for classes `B` and `D`. If `BD` is a class then `D` would require multiple inheritance of superclasses. If `BD` were an interface, then the instance variables of `B` and `D` are not representable (Java interfaces may include static fields, but not instance fields).

There are several possible work-arounds for this problem. The JIR type system could be modified to allow either a class or an interface to be inserted. Alternately, the point may be typed as `Object` and down-casts at uses inserted. Note, that since these casts can never fail, they may be implemented without any run-time cost. We chose the last option because it keeps the type system of JIR close to the original Java type system and because it can be implemented with almost the same mechanism used to narrow small integer types.

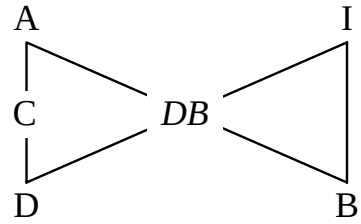

Fig. 10. Multiple inheritance type hierarchy with infeasible completion point. Types `A-D` are classes, `I` an interface, and `DB` is the desired completion point.

7.3 Complexity

As implemented in Marmot, in the worst case, the preprocessing of the bytecode (conversion to temporary form, lifetime splitting via SSA, and subroutine inlining) is exponential in the size of the original bytecode. It would suffice to implement preprocessing using a quadratic algorithm, resulting in a linear output.[9] Let m represent the size of the program after preprocessing.

Constraint collection (Step 1) takes time $O(m)$ in the input size of the program (i.e., after preprocessing). Constraint closure (Step 2) takes time and results in a constraint graph that is $O(d * m) = O(m^2)$ where d is the maximal depth of array type constructors. SCC formation (Step 3) is linear in the size of its input graph: $O(d * m)$ in this case. Calculating filters (Step 4) of lower bounds involves intersecting subsets of $\mathcal{H}$. The sets are of size bounded by h, where h is the size of the hierarchy $\mathcal{H}$. An intersection is performed at most once for each edge in the closed constraint graph. The intersections take time $O(h)= O(m)$ per edge, hence $O(d * m * h)$ in total. Precomputing the filters (represented as boolean vectors of length h) for all type constants requires $O(h)$ calls to reachability in $\mathcal{H}$ (viewed as a graph) and hence is $O(h^2)$. The total cost of Step 4 is therefore $O(h^2 + d * m * h) = O(m^3)$. Constructing types (Step 5) by computing minimal elements of a set $A \subseteq \mathcal{H}$ can be done by processing the elements of A in reverse postorder: for each $x_i \in A$ taken in that order, we mark all unmarked nodes in $\mathcal{H}$ reachable from x in $\mathcal{H}$. If x_i is unmarked, we add it to the set $Min\, A$. This step visits an edge in $\mathcal{H}$ at most once (edges from marked nodes are not visited again); hence Step 5 can be done in time $O(h)$. Finally, applying the solution (Step 6) takes $O(m)$ time. We have shown:

THEOREM 7.1. *Let m be the size of the preprocessed program; let d be the maximal depth of array constructors in the program; and let h be the size of the hierarchy $\mathcal{H}$. Then type elaboration can be computed in time $O(h^2 + d * m * h) = O(m^3)$.*

Thus, the complexity of type elaboration excluding preprocessing is $O(m^3)$. If preprocessing were implemented using linear encoding of subroutines and variable lifetime splitting, then $O(m)$=$O(n)$ and type elaboration takes $O(n^3)$ time, where

[9]Subroutines may be supported using the `jsr` and `ret` instructions or encoded as in Freund [1998], and lifetime splitting without SSA conversion in $O(n^2)$ time and $O(n)$ space where n is the size of the bytecode program.

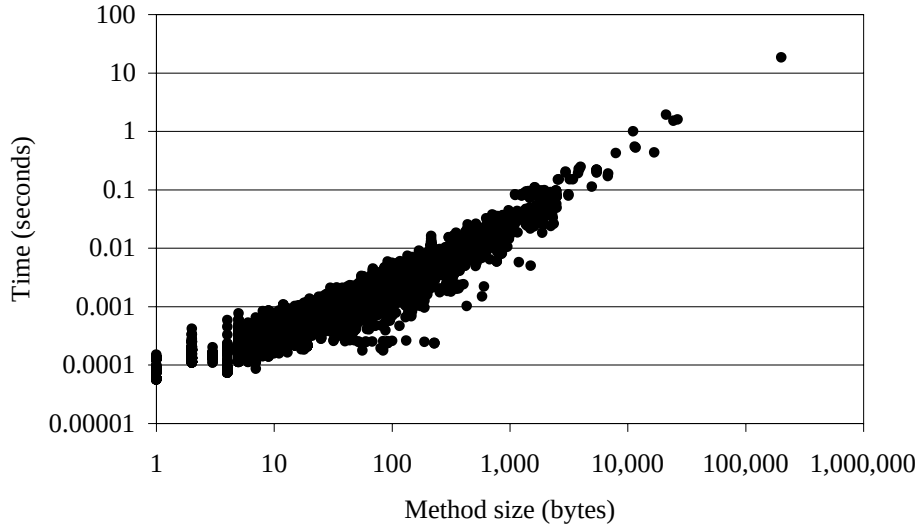

Fig. 11. Per method type elaboration.

n is the size of the original program. In practice, there is some evidence that the conversion to SSA and subroutine inlining are linear in the average case [Cytron et al. 1989; Freund 1998]. Also, d and h will be small relative to n in most cases.

7.4 Empirical Results

While the worst-case complexity of type elaboration is in terms of the overall program size, it is interesting to examine the cost of type elaboration relative to the size of the individual methods. Figure 11 shows the time it takes to type elaborate a method relative to the size of the unpreprocessed bytecode method. Note that the data are presented on a log-log scale, which makes the data for the very small method appear distinctly on the left of the graph. The data for this figure represents type elaboration run over approximately 22,300 methods from 65 programs.[10] The methods range in size from 1 to 200,634 bytes. The largest method, a class initialization method written by an LALR parser-generator, took 18.5 seconds to type elaborate. Data were collected on an otherwise idle dual-processor Pentium II/300MHz processor computer running Windows NT/4.

Type elaboration is run early in the optimization pipeline, and hence must solve types for many more methods and many more locals than will persist at later stages. Type elaboration times for entire benchmark programs range from 0.3 seconds to 45.5 seconds, representing approximately 4% of the compilation time. For comparison, SSA conversion represents approximately 5% of compilation time.

Only 2 of the 65 programs had any methods that required that the small integer types be combined (one had 3 methods that convolved small integers; the other

[10]There is some duplication of methods between the different programs caused, principally, by the runtime system.

had 27 methods).

8. RELATED WORK

Gagnon and Hendren [1999] also present an algorithm for type inference for Java bytecodes based upon solving constraints. Although superficially similar, the approaches differ in important ways. Gagnon and Hendren do not introduce new types. Type elaboration successfully types verifiable bytecode that Gagnon and Hendren's technique can not. Type elaboration is polynomial whereas their technique is exponential and type elaboration supports typing of the small integer types.

Palsberg and O'Keefe [1995] have shown that their notion of safety analysis is equivalent to typability in a subtyping system. A major difference to our results is that their type system is already lattice-based, and the problem of completion does not arise in their work.

Previous work, including Lincoln and Mitchell [1992], Tiuryn [1992], Pratt and Tiuryn [1996], and Benke [1993], has established that subtype inference over arbitrary posets is intractable. However, if the subtype order on base types is a lattice, the problem becomes PTIME solvable [Tiuryn 1992]. Subtype completion exploits this fact by transforming a poset to a lattice. The transformation simplifies the problem and makes it easier to solve. As we have shown, this simplified formulation is adequate for our notion of type safety.

The use of upward and downward closed sets, corresponding to order-filters and order-ideals of types, is standard in many works on object-oriented analysis (e.g., Bacon [1997]). Aït-Kaci et al. [1989] present data structures for efficient lattice operations on partially ordered sets representing inheritance hierarchies. See also Caseau [1993], which describes an algorithm for lattice completion.

There has been a substantial amount of work on formalizing various aspects of bytecode verification [Qian 1998; Stata and Abadi 1998; Coglio et al. 1998; Yelland 1999; Pusch 1999; O'Callahan 1999; Goldberg 1997]. Only a subset of these directly address the issue of verification with class hierarchies that have multiple inheritance. Others are focused on particular aspects of verification such as the polymorphism in subroutines [Stata and Abadi 1998; Freund and Mitchell 1999; O'Callahan 1999] or initialization of locals [Freund and Mitchell 1998]. Goldberg [1997] and Qian [1999] formalize bytecode using dataflow and safety. Coglio et al. [1998] formalize verification using constraints. A number of papers, including Oheimb and Nipkow [1999] and Syme [1997] have formalized the Java type system. League et al. [1999] describe a typed intermediate language for Java based upon system F^{ω}.

Shivers [1991] presents a technique recovering what we called *small integer types* for Scheme. That domain is different, and more difficult, because the programs are not statically typed.

9. CONCLUSION

This paper has presented type elaboration, a practical algorithm for recovering a strongly typed intermediate representation from partially typed verifiable bytecode. We showed that the type system of JIR arises in a principled way from the type system of Java and the flow-based verification of Java bytecode. For an abstract formalization of the problem, we showed that the technique of subtype completion adds just enough types to the subtype hierarchy to precisely correspond to the rules

for bytecode verification, allowing us to derive a provably correct type inference algorithm.

In addition to issues caused by multiple inheritance, a practical implementation of type elaboration must work for verifiable bytecode that includes distinct variables assigned the same name, convolved small integer types, and the Java rule for covariant array subtyping. We have sketched how these pragmatic issues are addressed in the implementation of type elaboration in Marmot. Although the asymptotic complexity of type elaboration is cubic, it appears to be acceptable in practice.

Type elaboration and the use of a strongly typed intermediate representation has proved to be a useful technique in the development of the Marmot compiler. Although having precise and accurate types is useful in optimization, the main benefit has been as a machine-verifiable weak semantic correctness check used in debugging the optimizations.

Of course, typechecking the intermediate representation can find only some errors in the compiler. The type system that we employ is relatively weak. By employing stronger type systems, it would be possible to catch a larger class of incorrect transformations [League et al. 1999; Morrisett et al. 1997; Necula 1998].

In addition to being too weak, the type system can also be too strong. A minor annoyance in employing a strongly typed intermediate representation is that occasionally one must augment optimization transformations so as to preserve typability of the intermediate representation. Even in the simple case of copy propagation, transformations must be inhibited in certain cases. For example, it is not legal to propagate a `null` constant into an array indexing context (because the element type is not explicitly stated). Nor is it legal to propagate a smaller array type into the left-hand side of an array assignment replacing a larger array type (because of issues precipitated by the covariant array subtyping rules of Java).

The technique of subtype completion can transform an intractable type inference system into a tractable one while preserving essential safety properties. This may be of independent interest beyond its application to type elaboration.

APPENDIX

A. PROOFS

A.1 Proof of Lemma 5.1

Before proving the lemma, we give a technical definition. For a system $\mathcal{F}[\![M]\!]$ of flow constraints (Figure 6), we divide its flow variables into two categories, typed and untyped. A flow variable $\mathcal{X}_e$ is called *typed*, if e is a formal parameter, e is of the form $this_\omega$, e is a method invocation, e is a constant, or e is a field selection. A flow variable is *untyped* if it is not typed.

LEMMA 5.1 *For every phrase M, the constraint system $\mathcal{F}[\![M]\!]$ has a least solution.*

PROOF. To see that any system of the form $\mathcal{F}[\![M]\!]$ has a least solution if it has any solutions, notice that the constraints in $\mathcal{F}[\![M]\!]$ have one of the forms $\mathcal{X} = \tau$ or $\mathcal{X} \subseteq \mathcal{X}'$. Moreover, constraint variables range over sets of types, which form a lattice. Constraint resolution for systems of the form $\mathcal{F}[\![M]\!]$ therefore falls within

the framework of monotone inequalities over a lattice, which implies the existence of least solutions to solvable constraint systems [Rehof and Mogensen 1999].

To see that every system $\mathcal{F}[\![M]\!]$ indeed has a solution, note, from the definition of $\mathcal{F}$, that every constraint has the form $\mathcal{X}^t = \tau$ or $\mathcal{X}_e \subseteq \mathcal{X}^u$, where $\mathcal{X}_e$ is either typed or untyped, $\mathcal{X}^t$ is typed, and $\mathcal{X}^u$ is untyped; moreover, whenever we have $\mathcal{X}^t = \tau_1$ and $\mathcal{X}^t = \tau_2$ then $\tau_1 = \tau_2$. It then easily follows that every system can be solved by choosing sufficiently large sets of types for the untyped variables. □

A.2 Proof of Theorem 6.2

THEOREM 6.2 (SOUNDNESS AND COMPLETENESS) *A program M is typable in system $\mathsf{JIR}(\mathsf{DM}(\Sigma), \mathsf{DM}(\mathcal{H}))$ if and only if M is safe with respect to Σ and $\mathcal{H}$.*

PROOF. We begin by proving soundness, i.e., whenever M has a type in system $\mathsf{JIR}(\mathsf{DM}(\Sigma), \mathsf{DM}(\mathcal{H}))$, then M is safe with respect to Σ and $\mathcal{H}$.

To prove soundness, assume that M is typable, so that (by Lemma 6.1) we have a least solution $\mu : \mathit{TyVar} \to \mathsf{DM}(\mathcal{H})$ to the constraint system $\mathcal{I}[\![M]\!]$ (the fact that a *least* solution exists follows by an argument similar to the one given in the proof of Lemma 5.1).

We now proceed to define a function ϕ (depending on μ), which maps flow variables to subsets of $\mathcal{H}$. It will be shown that ϕ satisfies the constraints in $\mathcal{F}[\![M]\!]$ as well as all the safety checks of Figure 7. Soundness follows from the existence of ϕ with these properties, because if *any one* solution to $\mathcal{F}[\![M]\!]$ satisfies the safety checking rules, then so does the least solution to $\mathcal{F}[\![M]\!]$ (observe that all checks have the form $S \sqsubseteq \tau$, and if S satisfies this condition, then so does any subset of S).

In order to define the map ϕ, let $\mathit{Max}\, S$ denote the set of all maximal elements of the subset $S \subseteq \mathcal{H}$ (an element $x \in S$ is maximal, if and only if $x \leq y$ implies $x = y$ for all $y \in S$). The map $\phi : \mathit{FlowVar} \to \wp(\mathcal{H})$ is then defined by

$$\begin{aligned} \phi(\mathcal{X}_e^t) &= \mathit{Max}\, \mu(\alpha_e) \\ \phi(\mathcal{X}_e^u) &= \mu(\alpha_e) \end{aligned}$$

where $\mathcal{X}_e^t$ is a *typed* variable, and $\mathcal{X}_e^u$ is an *untyped* variable (definitions of typed and untyped flow variables are given in Section A.1). We claim that ϕ solves the system $\mathcal{F}[\![M]\!]$ and satisfies all safety checks. Notice that all flow constraints in $\mathcal{F}[\![M]\!]$ have one of the forms $\mathcal{X}_e^t = \tau$ or $\mathcal{X}_e \subseteq \mathcal{X}_{e'}^u$, where $\mathcal{X}_e$ is either typed or untyped. Since $\mathit{Max}\, S \subseteq S$, it follows that one has

$$\mu(\alpha_e) \subseteq \mu(\alpha_{e'}) \Rightarrow \phi(\mathcal{X}_e) \subseteq \phi(\mathcal{X}_{e'}^u) \tag{3}$$

regardless of whether $\mathcal{X}_e$ is typed or untyped. We will use this observation freely in the following. We now show that ϕ solves $\mathcal{F}[\![M]\!]$ and satisfies all safety checks. We do so by considering all constraints generated from an arbitrary occurrence of a phrase in M. We consider each occurrence by cases over the form of the phrase, and for each such occurrence we consider the associated constraints and safety checks according to the definitions in Figure 6 and Figure 7:

—*Case* $e.f(e_1, \ldots, e_n)$ with constraint $\mathcal{X}_{e.f(e_1,\ldots,e_n)} = \tau'$, where $\Sigma(f) = \vec{\tau} \to \tau'$.

The map ϕ solves the flow constraint: One has $\alpha_{e.f(e_1,\ldots,e_n)} = \downarrow\tau'$ in $\mathcal{I}[\![M]\!]$, and $\mathcal{X}_{e.f(e_1,\ldots,e_n)}$ is typed. Hence,

$$\phi(\mathcal{X}_{e.f(e_1,\ldots,e_n)}) = \mathit{Max}\,\mu(\alpha_{e.f(e_1,\ldots,e_n)}) = \mathit{Max}\,(\downarrow\tau') = \{\tau'\}$$

as desired.

The map ϕ satisfies the safety checks: The relevant safety check is $(S3)$, requiring $\phi(\mathcal{X}_{e_i}) \sqsubseteq \tau_i$. One has $\alpha_{e_i} \leq \downarrow\tau_i$ in $\mathcal{I}[\![M]\!]$, hence $\mu(\alpha_{e_i}) \subseteq \downarrow\tau_i$, from which we get that $\mu(\alpha_{e_i}) \sqsubseteq \tau_i$, hence also $\mathit{Max}\,\mu(\alpha_{e_i}) \sqsubseteq \tau_i$, and therefore $\phi(\mathcal{X}_{e_i}) \sqsubseteq \tau_i$, as desired. Moreover, the safety condition $\mathsf{sig}(\widehat{\mathcal{X}}_e) \neq \emptyset$ is satisfied, because the constraint $\mathsf{sig}(\alpha_e) \neq \emptyset$ is in $\mathcal{I}[\![M]\!]$.

—*Case* $e.a$, with constraint $\mathcal{X}_{e.a} = \tau$, where $\Sigma(a) = \omega.\tau$.
One has $\mathsf{DM}(\Sigma)(a) = (\downarrow\omega).(\downarrow\tau)$ and $\{\alpha_e \leq \downarrow\omega, \alpha_{e.a} = \downarrow\tau\} \subseteq \mathcal{I}[\![M]\!]$. Therefore, $\mu(\alpha_{e.a}) = \downarrow\tau$. Since $\mathcal{X}_{e.a}$ is typed, one has $\phi(\mathcal{X}_{e.a}) = \mathit{Max}\,\mu(\alpha_{e.a}) = \mathit{Max}\,(\downarrow\tau) = \{\tau\}$, so ϕ is seen to satisfy the flow constraint.
The safety check requires $\phi(\mathcal{X}_e) \sqsubseteq \omega$. Because $\alpha_e \leq \downarrow\omega$ is in $\mathcal{I}[\![M]\!]$, one has $\mu(\alpha_e) \subseteq \downarrow\omega$, and therefore $\mu(\alpha_e) \sqsubseteq \omega$. Because we have $\mathit{Max}\,\mu(\alpha_e) \subseteq \mu(\alpha_e)$, it follows that $\phi(\mathcal{X}_e) \sqsubseteq \omega$, as desired.

—*Case* $z = e$ with constraint $\mathcal{X}_e \subseteq \mathcal{X}_z$.
We have $\alpha_e \leq \alpha_z$ in $\mathcal{I}[\![M]\!]$, so $\mu(\alpha_e) \subseteq \mu(\alpha_z)$, hence $\phi(\mathcal{X}_e) \subseteq \phi(\mathcal{X}_z)$. Here, we used that z is untyped, which guarantees that $\phi(\mathcal{X}_z) = \mu(\alpha_z)$, whereas $\phi(\mathcal{X}_e)$ is either $\mu(\alpha_e)$ or $\mathit{Min}\,\mu(\alpha_e)$, and the desired inclusion holds in both cases.
There is no safety check associated with an assignment statement.

—*Case* x with constraint $\mathcal{X}_x = \tau$, where $\Sigma(x) = \tau$. We have $\alpha_x = \downarrow\tau$ in $\mathcal{I}[\![M]\!]$, so that (using the fact that $\mathcal{X}_x$ is typed) one gets $\phi(\mathcal{X}_x) = \mathit{Max}\mu(\alpha_x) = \mathit{Max}\,(\downarrow\tau) = \{\tau\}$, as desired.
There is no safety check associated with a parameter occurrence.

—*Case* $\omega :: f(x_1 : \tau_1, \ldots, x_n : \tau_n)\{s\}$ with constraints $\mathcal{X}_{\mathit{this}_\omega} = \omega$, $\mathcal{X}_{x_i} = \tau_i$, where $\Sigma(f) = (\tau_1, \ldots, \tau_n) \to \tau'$.
One has $\{\alpha_{\mathit{this}_\omega} = \downarrow\omega, \alpha_{x_i} = \downarrow\tau_i\} \subseteq \mathcal{I}[\![M]\!]$, and $\mathcal{X}_{\mathit{this}_\omega}$ and $\mathcal{X}_{x_i}$ are all typed. It follows that one has $\phi(\mathcal{X}_{\mathit{this}_\omega}) = \{\omega\}$ and $\phi(\mathcal{X}_{x_i}) = \mathit{Max}\ \ \mu(\alpha_{x_i}) = \mathit{Max}\,(\downarrow\tau_i) = \{\tau_i\}$, and the flow constraints are seen to be satisfied under ϕ.
There is no safety check associated with a declaration statement.

The remaining cases are similar and left out. This concludes the soundness proof.

To prove completeness, we need to show that, whenever the least solution to $\mathcal{F}[\![M]\!]$ satisfies the safety checking rules of Figure 7, then the constraint system $\mathcal{I}[\![M]\!]$ has a solution in $\mathsf{DM}(\mathcal{H})$, so that (by Lemma 6.1) M types. We will do this by defining a mapping ψ from type variables to $\mathsf{DM}(\mathcal{H})$ depending on the least solution to $\mathcal{F}[\![M]\!]$. The function $\psi : \mathit{TyVar} \to \mathsf{DM}(\mathcal{H})$ is defined by

$$\psi(\alpha_e) = (\widehat{\mathcal{X}}_e)^{u\ell}$$

We show that all constraints generated from an arbitrary occurrence of a phrase in M are satisfied under the mapping ψ, provided that the least solution to $\mathcal{F}[\![M]\!]$ satisfies all safety checks. We proceed by cases over the form of phrases:

—*Case* $e.f(e_1, \ldots, e_n)$ with constraints $\alpha_{e.f(e_1,\ldots,e_n)} = \downarrow\tau', \mathsf{sig}(\alpha_e, f) \neq \emptyset, \alpha_{e_i} \leq \downarrow\tau_i$, where $\Sigma(f) = (\tau_1, \ldots, \tau_n) \to \tau'$.

One has $\mathcal{X}_{e.f(e_1,\ldots,e_n)} = \tau'$ in $\mathcal{F}[\![M]\!]$, so that $\widehat{\mathcal{X}}_{e.f(e_1,\ldots,e_n)} = \{\tau'\}$ holds, and therefore we have $\psi(\alpha_{e.f(e_1,\ldots,e_n)}) = \{\tau'\}^{u\ell} = (\uparrow\tau')^{\ell} = \downarrow\tau'$, which shows that $\psi(\alpha_{e.f(e_1,\ldots,e_n)}) \subseteq \downarrow\tau'$, and so ψ solves the first type inequality mentioned above. To see that the second condition, $\mathsf{sig}(\alpha_e, f) \neq \emptyset$, is satisfied, consider that by $(S3)$ one has $\cap_{\omega\in\widehat{\mathcal{X}}_e} \mathsf{sig}(\omega, f) \neq \emptyset$, hence for some $\omega' \in (\widehat{\mathcal{X}}_e)^u$ one has $\mathsf{sig}(\omega', f) \neq \emptyset$. It follows that for each $\omega'' \in (\widehat{\mathcal{X}}_e)^{ul}$ one has $\mathsf{sig}(\omega'', f) \neq \emptyset$, which shows that the constraint is satisfied as desired. To see that the third inequality is also satisfied under ψ, consider that the safety check $(S3)$ requires that $\widehat{\mathcal{X}}_{e_i} \sqsubseteq \tau_i$. It follows that $\widehat{\mathcal{X}}_{e_i} \subseteq \downarrow\tau_i$, and therefore one has $\psi(\alpha_{e_i}) = (\widehat{\mathcal{X}}_{e_i})^{u\ell} \subseteq (\downarrow\tau_i)^{u\ell} = \downarrow\tau_i$, thereby showing $\psi(\alpha_{e_i}) \subseteq \downarrow\tau_i$, as desired.

—*Case* $e.a$ with constraints $\alpha_e \leq \downarrow\omega, \alpha_{e.a} = \downarrow\tau$, where $\Sigma(a) = \omega.\tau$.
One has $\mathcal{X}_{e.a} = \tau$ in $\mathcal{F}[\![M]\!]$, hence $\widehat{\mathcal{X}}_{e.a} = \{\tau\}$, and hence $\psi(\alpha_{e.a}) = \{\tau\}^{u\ell} = \downarrow\tau$, proving that ψ satisfies the constraint $\alpha_{e.a} = \downarrow\tau$. To see that ψ satisfies the constraint $\alpha_e \leq \downarrow\omega$, consider that the safety checking rules require that $\widehat{\mathcal{X}}_e \sqsubseteq \omega$, which implies $\widehat{\mathcal{X}}_e \subseteq \downarrow\omega$, and hence $\psi(\alpha_e) = (\widehat{\mathcal{X}}_e)^{u\ell} \subseteq (\downarrow\omega)^{u\ell} = \downarrow\omega$, as desired.

—*Case* $z = e$ with constraint $\alpha_e \leq \alpha_z$.
One has $\mathcal{X}_e \subseteq \mathcal{X}_z$ in $\mathcal{F}[\![M]\!]$, hence $\widehat{\mathcal{X}}_e \subseteq \widehat{\mathcal{X}}_z$, hence $(\widehat{\mathcal{X}}_e)^{u\ell} \subseteq (\widehat{\mathcal{X}}_z)^{u\ell}$, which shows that $\psi(\alpha_e) \subseteq \psi(\alpha_z)$, as desired.

—*Case* $e.a = e'$ with constraint $\alpha_{e'} \leq \alpha_{e.a}$, where $\Sigma(a) = \omega.\tau$.
One has $\widehat{\mathcal{X}}_{e.a} = \tau$ by the constraints in $\mathcal{F}[\![M]\!]$, and moreover $\widehat{\mathcal{X}}_{e'} \sqsubseteq \tau$ by the safety checking rule $(S2)$. It follows from the former relation that $\psi(\alpha_{e.a}) = \{\tau\}^{u\ell} = \downarrow\tau$, and since $\widehat{\mathcal{X}}_{e'} \sqsubseteq \tau$, we get $\widehat{\mathcal{X}}_{e'} \subseteq \downarrow\tau$, which implies $\psi(\alpha_{e'}) \subseteq \downarrow\tau = \psi(\alpha_{e.a})$. This shows that ψ satisfies the inequality generated in this case.

—*Case* x with constraint $\alpha_x = \downarrow\tau$, where $\Sigma(x) = \tau$.
By the definition of $\mathcal{F}[\![M]\!]$, one has $\widehat{\mathcal{X}}_x = \tau$, and therefore $\psi(\alpha_x) = \{\tau\}^{u\ell} = \downarrow\tau$, showing that ψ satisfies the inequality in this case.

—*Case* $\omega :: f(x_1 : \tau_1, \ldots, x_n : \tau_n)\{s\}$ with constraints $\alpha_{this_\omega} = \downarrow\omega, \alpha_{x_i} = \downarrow\tau_i$, where $\Sigma(f) = (\tau_1, \ldots, \tau_n) \to \tau'$.
One has $\widehat{\mathcal{X}}_{this_\omega} = \omega$ and $\widehat{\mathcal{X}}_{x_i} = \tau_i$ by the definition of $\mathcal{F}[\![M]\!]$. These relations imply that $\psi(\alpha_{this_\omega}) = \downarrow\omega$ and $\psi(\alpha_{x_i}) = \downarrow\tau_i$.

The remaining cases are similar. This concludes the proof of completeness. □

A.3 Proof of Theorem 6.3

In order to prove Theorem 6.3 we need to understand the satisfiability problem for inequalities over the lattice $\mathsf{DM}(\mathcal{H})$. Here it is useful to look at an abstract version of the problem first, by recalling a general lemma concerning inequalities over an arbitrary complete lattice. In order to state this lemma, we need a few definitions. Let P be any poset. A *system of inequalities over* P is a finite set of formal inequalities of the form $\alpha \leq \alpha'$, $k \leq \alpha$ or $\alpha \leq k$, where α and α' range over variables and k ranges over constants drawn from P. A system C is said to be *satisfiable* in P if and only if there exists a substitution v from variables to elements of P such that $v(\xi) \leq_P v(\xi')$ is true in P, for all $\xi \leq \xi' \in C$ (here ξ and ξ' range over variables or constants from C). The *closure* of a system C, denoted C^*, is the

least system of inequalities containing C and satisfying the condition

$$(\xi \leq \xi') \in C^*, (\xi' \leq \xi'') \in C^* \Rightarrow (\xi \leq \xi'') \in C^*.$$

A system C is said to be *consistent* if and only if we have

$$(k_1 \leq k_2) \in C^* \Rightarrow k_1 \leq_P k_2$$

for all $k_1, k_2 \in P$. Finally, for a system of inequalities, C, define for each variable α in C the subset $\downarrow_C(\alpha) \subseteq P$, given by

$$\downarrow_C(\alpha) = \{k \in P \mid (k \leq \alpha) \in C^*\}.$$

Then we have

LEMMA A.1. *Let $\mathcal{L}$ be a complete lattice, and let C be a system of inequalities over $\mathcal{L}$. Then C is satisfiable if and only if C is consistent. Moreover, if C is satisfiable, then the substitution μ given by*

$$\mu(\alpha) = \bigvee \downarrow_C(\alpha)$$

is the least solution to C.

PROOF. It is standard that the substitution μ solves C, whenever C is consistent, and proofs of this fact in a subtyping setting can be found in, for example, Lincoln and Mitchell [1992] and Tiuryn [1992]. It is easy to verify that μ must be the least solution to C, if indeed μ is a solution. □

To prove Theorem 6.3, recall the definition of D_α,

$$D_\alpha = \{\tau \in \mathcal{H} \mid \tau \leq \alpha \in \lceil C \rceil^*\}$$

THEOREM 6.3 *Let $C = \mathcal{I}[\![M]\!]$, and assume that C is satisfiable. Then there is a unique least solution μ to C in $\mathsf{DM}(\mathcal{H})$, which is given by*

$$\mu(\alpha) = (D_\alpha)^{u\ell} = \left(\bigcap_{\tau \in D_\alpha} \uparrow\tau\right)^{\ell}.$$

PROOF. We apply Lemma A.1 to the case $\mathcal{L} = \mathsf{DM}(\mathcal{H})$, which allows us to characterize the least solution μ to $\mathcal{I}[\![M]\!]$ by

$$\mu(\alpha) = \bigvee \downarrow_C(\alpha)$$

with $C = \mathcal{I}[\![M]\!]$. Now recall that we have $\bigvee_{i\in I} A_i = (\bigcup_{i\in I} A_i)^{u\ell}$, by equation (1). Moreover, it is easy to verify, using definitions, that one has $(\bigcup_{i\in I} A_i)^u = \bigcap_{i\in I}(A_i)^u$ and $(\downarrow x)^u = \uparrow x$. Using these identities, we can compute as follows:

$$\begin{aligned}
\bigvee \downarrow_C(\alpha) &= \bigvee\{\downarrow\tau \mid \downarrow\tau \in \downarrow_C(\alpha)\} \\
&= \bigvee\{\downarrow\tau \mid \tau \in \downarrow_{\lceil C \rceil}(\alpha)\} \\
&= \bigvee_{\tau\in D_\alpha} \downarrow\tau \\
&= \left(\bigcup_{\tau\in D_\alpha} \downarrow\tau\right)^{u\ell} \\
&= \left(\bigcap_{\tau\in D_\alpha} (\downarrow\tau)^u\right)^{\ell} \\
&= \left(\bigcap_{\tau\in D_\alpha} \uparrow\tau\right)^{\ell}
\end{aligned}$$

which proves that $\mu(\alpha) = \left(\bigcap_{\tau \in D_\alpha} \uparrow\tau\right)^\ell$ solves C. At the first equation in the calculation above we used the fact that the only constants occurring in constraint systems of the form $\mathcal{I}[\![M]\!]$ are principal ideals, of the form $\downarrow\tau$.

To see that $(D_\alpha)^{u\ell} = \left(\bigcap_{\tau \in D_\alpha} \uparrow\tau\right)^\ell$, it is sufficient (by previous equations) to show that $(D_\alpha)^u = \left(\bigcup_{\tau \in D_\alpha} \downarrow\tau\right)^u$. To see the inclusion from right to left, suppose that $x \in \left(\bigcup_{\tau \in D_\alpha} \downarrow\tau\right)^u$. Then $x \geq \bigcup_{\tau \in D_\alpha} \downarrow\tau$. Since $D_\alpha \subseteq \bigcup_{\tau \in D_\alpha} \downarrow\tau$, it follows that $x \geq D_\alpha$, hence $x \in (D_\alpha)^u$. To see the other inclusion, assume $x \in (D_\alpha)^u$. Then for any $y \in D_\alpha$ one has $x \geq y$. Hence $x \geq \downarrow y$ for all $y \in D_\alpha$, and therefore also $x \geq \bigcup_{\tau \in D_\alpha} \downarrow\tau$, hence $x \in \left(\bigcup_{\tau \in D_\alpha} \downarrow\tau\right)^u$. □

A.4 Proof of Lemma 6.4

In order to prove Lemma 6.4 we first prove a couple of auxiliary lemmas. Recall that we define $A \preceq B$ by

$$A \preceq B \text{ iff } \forall x \in B.\ \exists y \in A.\ y \leq x.$$

LEMMA A.2. *Let A and B be upward closed subsets of a poset P. Then*

$$A \subseteq B \Leftrightarrow B \preceq A.$$

PROOF. Easy. □

LEMMA A.3. *Let A, B be subsets of a poset P. Then*

$$A^{u\ell} \subseteq B^{u\ell} \Leftrightarrow A^u \preceq B^u.$$

PROOF. ($\Rightarrow$). One has

$$A^{u\ell} \subseteq B^{u\ell} \Rightarrow (B^{u\ell})^u \subseteq (A^{u\ell})^u \Leftrightarrow B^u \subseteq A^u$$

by the identity $A^{u\ell u} = A^u$ and antimonotonicity (with respect to inclusion) of the operations $\bullet^u$ and $\bullet^\ell$. Now, $B^u \subseteq A^u$ implies $A^u \preceq B^u$, by Lemma A.2, which is applicable since A^u and B^u are upward closed sets.

($\Leftarrow$). Assume $A^u \preceq B^u$. Since $\bullet^\ell$ is antimonotonic with respect to inclusion, it is sufficient to show that $B^u \subseteq A^u$. So let $x \in B^u$. Then, by the assumption, $A^u \preceq B^u$, there exists $y \in A^u$ with $y \leq x$. Since A^u is upward closed and $y \in A^u$ and $y \leq x$, it follows that $x \in A^u$. We have shown $B^u \subseteq A^u$ as desired. □

LEMMA A.4. *Let A and B be subsets of the poset $\mathcal{H}$. If $\mathcal{H}$ has no infinite descending chains, then one has*

$$A \preceq B \Leftrightarrow \text{Min } A \preceq \text{Min } B.$$

PROOF. ($\Rightarrow$). Assume $A \preceq B$. Let $x \in \mathit{Min}\, B$. Then, by the assumption, there is $y \in A$ such that $y \leq x$. Since $\mathcal{H}$ has no infinite descending chains, we can find $y_m \in \mathit{Min}\, A$ such that $y_m \leq y$. So we have $y_m \leq x$, hence we have shown $\mathit{Min}\, A \preceq \mathit{Min}\, B$.

($\Leftarrow$) Assume $\mathit{Min}A \preceq \mathit{Min}B$. Let $x \in B$. Then, since $\mathcal{H}$ has no infinite descending chains, we can find $x_m \in \mathit{Min}\, B$ such that $x_m \leq x$. Then, by the assumption, there is $y \in \mathit{Min}\, A$ such that $y \leq x_m$, hence also $y \leq x$. This shows that $A \preceq B$. □

LEMMA 6.4 *Let A and B be subsets of the poset $\mathcal{H}$. If $\mathcal{H}$ has no infinite descending chains, then one has*

$$A^{u\ell} \subseteq B^{u\ell} \Leftrightarrow \mathrm{Min}\,(A^u) \preceq \mathrm{Min}\,(B^u)$$

PROOF. Follows from Lemma A.3 together with Lemma A.4. □

ACKNOWLEDGMENTS

We would like to thank Erik Ruf, Bjarne Steensgaard, Bob Fitzgerald, and David Tarditi for ideas, implementation, and suggestions on improving the presentation.

REFERENCES

AÏT-KACI, H., BOYER, R., LINCOLN, P., AND NASR, R. 1989. Efficient implemenation of lattice operations. *ACM Transactions on Programming Languages and Systems 11,* 1 (January), 115–146.

BACON, D. F. 1997. Fast and effective optimization of statically typed object-oriented languages. Ph.D. thesis, U.C. Berkeley.

BENKE, M. 1993. Efficient type reconstruction in the presence of inheritance. In *Mathematical Foundations of Computer Science (MFCS).* Springer Verlag, LNCS 711, 272–280.

BIRKHOFF, G. 1995. *Lattice Theory*, third ed. Colloquium Publications, vol. 25. American Mathematical Society, Providence, RI.

CASEAU, Y. 1993. Efficient handling of multiple inheritance hierarchies. In *Proceedings OOPSLA '93.* Washington,DC,USA, 271–287.

COGLIO, A., GOLDBERG, A., AND QIAN, Z. 1998. Towards a provably-correct implementation of the JVM bytecode verifier. Tech. Rep. KES.U.98.5, Kestrel Institue. August 1998. Also availabe in Proceedings of the OOPSLA '98 Workshop on the Formal Underpinnings of Java, Vancouver, B.C., October 1998.

CYTRON, R., FERRANTE, J., ROSEN, B. K., WEGMAN, M. N., AND ZADECK, F. K. 1989. An efficient method of computing static single assignment form. In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Programming Languages.*

DAVEY, B. A. AND PRIESTLEY, H. A. 1990. *Introduction to Lattices and Order.* Cambridge Mathematical Textbooks, Cambridge University Press.

FITZGERALD, R., KNOBLOCK, T. B., RUF, E., STEENSGAARD, B., AND TARDITI, D. 2000. Marmot: An optimizing compiler for Java. *Software: Practice and Experience 30,* 3 (Mar.), 199–232.

FREUND, S. N. 1998. The costs and benefits of Java bytecode subroutines. In *Formal Underpinnings of Java Workshop at OOPSLA.* http://ww-dse.doc.ic.ac.edu/∼sue/ oopsla/cfp.html.

FREUND, S. N. AND MITCHELL, J. C. 1998. A type system for object initialization in the Java bytecode langague. In *Proceedings OOPSLA '98, ACM SIGPLAN Notices.* 310–328.

FREUND, S. N. AND MITCHELL, J. C. 1999. A type system for Java bytecode subroutines and exceptions. Tech. rep., Stanford Univeristy, Computer Science Department. April.

GAGNON, E. AND HENDREN, L. 1999. Intra-procedural inference of static types for Java bytecode. Tech. Rep. 1, McGill University.

GOLDBERG, A. 1997. A specification of Java loading and bytecode verification. Tech. Rep. KES.U.92.1, Kestrel Institute. December.

GOSLING, J., JOY, B., AND STEELE, G. 1996. *The Java Language Specification.* The Java Series. Addison-Wesley, Reading, MA, USA.

HOANG, M. AND MITCHELL, J. C. 1995. Lower bounds on type inference with subtypes. In *Proc. 22nd Annual ACM Symposium on Principles of Programming Languages (POPL).* ACM Press, 176–185.

IBM. 1998. IBM high performance compiler for Java: An optimizing native code compiler for Java applications. http://www.alphaworks.ibm.com/ graphics.nsf/system/graphics/HPCJ/ $file/highpcj.html.

Instantiations, Inc. 1998. Jove: Super optimizing deployment environment for Java. http://www.instantiations.com/javaspeed/ jovereport.htm.

Kozen, D., Palsberg, J., and Schwartzbach, M. I. 1994. Efficient inference of partial types. *Journal of Computer and System Sciences 49,* 2, 306–324.

League, C., Shao, Z., and Trifonov, V. 1999. Representing java classes in a typed intermediate language. In *Proceedings of the 1999 ACM SIGPLAn Internationl Conference on Functional Programming*.

Leroy, X. and Ohori, A., Eds. 1998. *Types in Compilation*. Number 1473 in LNCS. Springer-Verlag.

Lincoln, P. and Mitchell, J. C. 1992. Algorithmic aspects of type inference with subtypes. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*. 293–304.

Lindholm, T. and Yellin, F. 1999. *The Java Virtual Machine Specification*, Second Edition ed. Addison-Wesley.

MacNeille, H. M. 1937. Partially ordered sets. *Transactions of the American Mathematical Society 42*, 90–96.

Morrisett, G., Walker, D., Crary, K., and Glew, N. 1997. From system F to typed assembly language. Tech. Rep. TR97-1651, Cornell University.

Morrisett, J. G. 1995. Compiling with types. Ph.D. thesis, Carnegie Mellon University. Published as CMU Technical Report CMU-CS-95-226.

NaturalBridge, LLC. 1998. Bullettrain Java compiler technology. http://www.naturalbridge.com/.

Necula, G. C. 1998. Compiling with proofs. Ph.D. thesis, Carnegie Mellon University.

O'Callahan, R. 1999. A simple, comprehensive type system for Java bytecode subroutines. In *Proceedings 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 70–78.

Oheimb, D. v. and Nipkow, T. 1999. Machine-checking the Java specification: Proving type-safety. In *Formal Syntax and Semantics of Java*, J. Alves-Foss, Ed. LNCS, vol. 1523. Springer-Verlag, 119–156.

Palsberg, J. and O'Keefe, P. 1995. A type system equivalent to flow analysis. *ACM Transactions on Programming Languages and Systems 17,* 4 (July), 576–599.

Pratt, V. and Tiuryn, J. 1996. Satisfiability of inequalities in a poset. *Fundamenta Informaticae 28,* 1–2, 165–182.

Pusch, C. 1999. Proving the soundness of a Java bytecode verifier specification in Isabelle/HOL. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99)*, W. R. Cleaveland, Ed. LNCS, vol. 1579. Springer-Verlag, 89–103.

Qian, Z. 1998. A formal specification of a large subset of Java virtual machine instructions for objects, methods and subroutines. In *Formal Syntax and Semantics of Java*, J. Alves-Foss, Ed. Number 1523 in LNCS. Springer-Verlag.

Qian, Z. 1999. Least types for memory locations in (Java) bytecode. Tech. rep., Kestrel Institute. http://www.kestrel.edu/HTML/people/ qian/pub-list.html.

Rehof, J. and Mogensen, T. Æ.. 1999. Tractable constraints in finite semilattices. *Science of Computer Programming 35,* 2, 191–221.

Shivers, O. 1991. Data-flow analysis and type recovery in scheme. In *Topics in Advanced Language Implementation*, P. Lee, Ed. The MIT Press, Chapter 3, 47–87.

Stata, R. and Abadi, M. 1998. A type system for Java bytecode subroutines. In *Proceedings 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 149–160.

SuperCede, Inc. 1998. SuperCede for Java, Version 2.03, Upgrade Edition. http://www.supercede.com/.

Syme, D. 1997. Proving JavaS type soundness. Tech. Rep. 427, University of Cambridge Computer Laboratory.

Tarditi, D. 1996. Design and implementation of code optimizations for a type-directed compiler for standard ml. Ph.D. thesis, Carnegie Mellon University.

Tarjan, R. E. 1972. Depth first search and linear graph algorithms. *SIAM Journal on Computing 1,* 2, 146–160.

Tiuryn, J. 1992. Subtype inequalities. In *Proc. 7th Annual IEEE Symp. on Logic in Computer Science (LICS), Santa Cruz, California*. IEEE Computer Society Press, 308–315.

Wand, M. 1987. A simple algorithm and proof for type inference. *Fundamenta Informaticae X*, 115–122.

Yelland, P. M. 1999. A compositional account of the Java virtual machine. In *Proceedings 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 57–59.

Received February 2000; accepted November 2000