Java$_{\ell ight}$ is Type-Safe — Definitely

Tobias Nipkow and David von Oheimb*
Fakultät für Informatik, Technische Universität München
`http://www4.informatik.tu-muenchen.de/~{nipkow|oheimb}`

Abstract

Java$_{\ell ight}$ is a large sequential sublanguage of Java. We formalize its abstract syntax, type system, well-formedness conditions, and an operational evaluation semantics. Based on this formalization, we can express and prove type soundness. All definitions and proofs have been done formally in the theorem prover Isabelle/HOL. Thus this paper demonstrates that machine-checking the design of non-trivial programming languages has become a reality.

1 Introduction

Java$_{\ell ight}$ is a large subset of the sequential part of Java [12]. This paper presents its formalization and a proof of type soundness — specified and verified in the theorem prover Isabelle/HOL [18]. In the sequel, 'Java$_{\ell ight}$' is abbreviated to 'Bali'.

On the face of it, this paper is mostly about Bali, its abstract syntax, type system, well-formedness conditions, and operational semantics, formalized as a hierarchy of Isabelle theories, and the structure of the machine-checked proof of type soundness. Although these technicalities do indeed take up much of the space, there is a meta-theme running through the paper, which we consider at least as important: the technology for producing machine-checked programming language designs has arrived. We emphasize that by 'machine-checked' we do not just mean that it has passed some type checker, but that some non-trivial properties of the language have been established with the help of a (semi-automatic) theorem prover. The latter process is still not a piece of cake, but it has become more than just feasible. Therefore any programming language intended for serious applications should strive for such a machine-checked design. The benefits are not just greater reliability, but also greater maintainability because the theorem prover keeps track of the impact that changes have on already established properties.

*Research supported by DFG SPP *Deduktion*.

To appear in the 25th ACM Symposium on Principles of Programming Languages, January 19–21, 1998, San Diego, California, USA.

1.1 Related work

The history of type soundness proofs goes back to the subject reduction theorem for typed λ-calculus but starts in earnest with Milner's slogan "Well-typed expressions do not go wrong" [14] in the context of ML. Milner uses a denotational semantics, in contrast to most of the later work, including ours. The question of type soundness came to prominence with the discovery of its failure in Eiffel [7]. Ever since, many designers of programming languages (especially OO ones) have been at pains to prove type soundness of their languages (see, for example, the series of papers by Bruce et al. [3, 4, 5]).

Directly related to our work is that by Drossopoulou and Eisenbach [10] who prove (on paper) type soundness of a subset of Java very similar to Bali. Although we were familiar with an earlier version [9] of their work and have certainly profited from it, our work is not a formalization of theirs in Isabelle/HOL but differs in many respects from it, for example in the representation of programs and the use of an evaluation (aka "big-step") semantics instead of a transition (aka "small-step") semantics. Simultaneously with our work, Syme [21] formalized the paper [9] as far as possible, uncovering two significant mistakes, both due to the use of transition semantics. Syme uses his own theorem prover DECLARE, also based on higher-order logic.

There are two other efforts to formalize aspects of Java in a theorem prover. Dean [8] studies the interaction of static typing with dynamic linking. His simple PVS specification addresses only the linking aspect and requires a formalization of Java (such as our work provides) to turn his lemmas about linking into theorems about the type soundness of dynamically linked programs. Cohen [6] has formalized the semantics of large parts of the Java Virtual Machine, essentially by writing an interpreter in Common Lisp. He used ACL2, the latest version of the Boyer-Moore theorem prover [2]. No proofs have been reported yet.

2 Overview of Bali

Bali includes the features of Java that we believe to be important for an investigation of the semantics of a practical imperative object-oriented language:

- interface and class declarations with instance fields and methods,
- subinterface, subclass, and implementation relations with inheritance, overriding, and hiding,
- some primitive types, objects (including arrays),

- method calls with static overloading and dynamic binding,
- type casts,
- a minimal treatment of exceptions.

The portion of Java we consider is roughly the same as covered by [10] and [21].

We do not consider Java packages and concurrency. For simplicity, we also leave out several features of Java like class variables and static methods, constructors and finalizers, final classes, and others. Several constructs are simplified without limiting the expressiveness of the language (see §4.1). We have not yet considered full exception handling and the visibility of names, but we aim to include them in later stages of our project.

3 The basics of Isabelle/HOL

Before we present the formalization of BALI, we briefly introduce the underlying theorem proving system.

Isabelle/HOL is the instantiation of the generic interactive theorem prover Isabelle [18] with Church's formulation of Higher-Order Logic and is very close to Gordon's HOL system [11]. In this paper HOL is short for Isabelle/HOL.

The appearance of formulas is standard, e.g. '$\longrightarrow$' is the implication symbol.

Logical constants are declared by giving their name and type, separated by '::'. Primitive recursive function definitions are written as usual. Non-recursive definitions are written with '$\stackrel{\mathsf{def}}{=}$'.

Types follow the syntax of ML, except that the function arrow is '$\Rightarrow$'. There are the basic types *bool* and *int*, and the polymorphic types $\alpha \times \beta$, α *set* and α *list*, and a conversion function set :: α *list* $\Rightarrow \alpha$ *set*. The "cons" operator on lists is the infix '#', concatenation the infix '@'. Tuples are pairs (with projections fst and snd) nested to the right, e.g. $(a,b,c) = (a,(b,c))$. Type abbreviations are simply given as equations, free datatypes are introduced with the **datatype** keyword. We frequently use the following type:

datatype α *option* = None | Some α

It has an unpacking function the :: α *option* $\Rightarrow \alpha$ such that the (Some x) = x and the None = arbitrary, where arbitrary is an unknown value.

Most of the HOL text shown in this paper is directly taken from the input files. However, it has been massaged by hand to hide Isabelle idiosyncrasies, increase readability, and adapt the layout. Minor typos may have been introduced in the process.

4 The formalization of Bali

This section presents all the important aspects of our formalization of BALI[1].

As far as BALI is a subset of Java, it strictly adheres to the official Java language specification [12], with three generalizations:

- we allow the result type of a method overriding another method to widen to the result type of the other method instead of requiring it to be identical.
- no check of result types in dynamic method lookup.
- the type of an assignment is determined by the right-hand (not left-hand) side.

[1] The Isabelle sources are available from the BALI project page http://www4.informatik.tu-muenchen.de/~isabelle/bali/

4.1 Abstract syntax

First, we describe how we represent the syntax of BALI as Isabelle datatypes, and which abstractions we have introduced thereby.

4.1.1 Programs

A BALI program is a pair of lists of interface and class declarations:

$$prog = idecl\ list \times cdecl\ list$$

Throughout the paper, the symbol 'Γ' denotes a BALI program, as we use programs as part of the static type context commonly written 'Γ'.

Each declaration is a pair of a name and the defined entity. We do not further specify the structure of names, but use the opaque HOL types *tname*, *mname*, and *ename* for BALI's type names, method names, and "expression names" (e.g. field identifiers, see [12, 6.5]).

$$\begin{aligned} iface &= tname\ list \times imdecl\ list \\ idecl &= tname \times iface \\ class &= tname\ option \times tname\ option \times \\ &\quad\ fdecl\ list \times cmdecl\ list \\ cdecl &= tname \times class \end{aligned}$$

An interface (*iface*) contains lists of superinterface names and method declarations. A *class* specifies the names of an optional superclass and implemented interface, and lists of field and method declarations. (A class that implements more than one interface can be modeled as implementing an intermediate interface that extends all these interfaces.)

$$\begin{aligned} field &= ty \\ fdecl &= ename \times field \\ sig &= mname \times ty \\ mhead &= ty \\ mbody &= stmt \times expr \\ methd &= mhead \times mbody \\ cmdecl &= sig \times methd \\ imdecl &= sig \times mhead \end{aligned}$$

A field declaration (*fdecl*) simply gives the field type (*ty*, see §4.2). A method declaration (*cmdecl* or *imdecl*) consists of a "signature" [12, 8.4.2] (i.e. the method name and parameter type(s), excluding the result type) followed by the result type (*mhead*) and, if it appears within a class, the method body (*mbody*). The latter consists of a statement and a return expression (*stmt* and *expr*, see below). Local variables of a method may be simulated with additional parameters. The separate return expression saves us from dealing with return statements occurring in arbitrary positions within the method body. Such statements may be replaced by assignments to a suitable result variable followed by a control transfer to the end of the method body, using the result variable as return expression. We provide a dummy result type and value for "void" methods. For simplicity, each method has exactly one parameter; multiple parameters can be simulated by a single parameter object with multiple fields.

The list representation of declarations gives an implicit finiteness constraint, which turns out to be necessary for the well-foundedness of the subclass and subinterface relation.

4.1.2 Representation of lookup tables

For the lookup of declared entities, we transform declaration lists into abstract tables. They are realized in HOL as "partial" functions mapping names to values:

$$(\alpha,\beta)\mathit{table} \;=\; \alpha \Rightarrow \beta\ \mathit{option}$$

The empty table, pointwise update, extension of one table by another, the function converting a declaration list into a table, and an auxiliary predicate relating entries of two tables, are easily defined:

$$\begin{array}{lll}
\mathsf{etable} & :: & (\alpha,\beta)\mathit{table} \\
_[_{:=}_] & :: & (\alpha,\beta)\mathit{table} \Rightarrow \alpha \Rightarrow \beta \quad \Rightarrow (\alpha,\beta)\mathit{table} \\
\oplus & :: & (\alpha,\beta)\mathit{table} \Rightarrow (\alpha,\beta)\mathit{table} \Rightarrow (\alpha,\beta)\mathit{table} \\
\mathsf{table} & :: & (\alpha{\times}\beta)\mathit{list} \;\Rightarrow (\alpha,\beta)\mathit{table} \\
_\ \mathsf{hiding}\ _\ \mathsf{entails}\ _ & :: & (\alpha,\beta)\mathit{table} \Rightarrow (\alpha,\gamma)\mathit{table} \Rightarrow \\
 & & (\beta \Rightarrow \gamma \Rightarrow \mathit{bool}) \Rightarrow \mathit{bool}
\end{array}$$

$$\begin{array}{ll}
\mathsf{etable} & \stackrel{\mathsf{def}}{=} \lambda k.\ \mathsf{None} \\
t[x{:=}y] & \stackrel{\mathsf{def}}{=} \lambda k.\ \mathsf{if}\ k = x\ \mathsf{then\ Some}\ y\ \mathsf{else}\ t\ k \\
s \oplus t & \stackrel{\mathsf{def}}{=} \lambda k.\ \mathsf{case}\ t\ k\ \mathsf{of\ None} \;\Rightarrow s\ k \\
 & \qquad\qquad\qquad\quad \mid \mathsf{Some}\ x \Rightarrow \mathsf{Some}\ x
\end{array}$$

$$\begin{array}{ll}
\mathsf{table}\ [] & = \mathsf{etable} \\
\mathsf{table}\ ((k,x)\#t) & = (\mathsf{table}\ t)[k{:=}x]
\end{array}$$

$$\begin{array}{l}
t\ \mathsf{hiding}\ s\ \mathsf{entails}\ R \stackrel{\mathsf{def}}{=} \forall k\ x\ y. \\
\qquad t\ k = \mathsf{Some}\ x \longrightarrow s\ k = \mathsf{Some}\ y \longrightarrow R\ x\ y
\end{array}$$

A simple application is the translation of programs to tables indexed by interface and class names:

$$\begin{array}{l}
\mathsf{iface}\ \Gamma \stackrel{\mathsf{def}}{=} \mathsf{table}\ (\mathsf{fst}\ \ \Gamma) \\
\mathsf{class}\ \Gamma \stackrel{\mathsf{def}}{=} \mathsf{table}\ (\mathsf{snd}\ \Gamma)
\end{array}$$

More interesting are the following functions that traverse the type hierarchy of a program, collecting the methods and fields into a table:

$$\begin{array}{lll}
\mathsf{imethd} & :: & \mathit{prog} \times \mathit{tname} \Rightarrow (\mathit{sig},\ \mathit{ref_ty} \times \mathit{mhead})\mathit{table} \\
\mathsf{cmethd} & :: & \mathit{prog} \times \mathit{tname} \Rightarrow (\mathit{sig},\ \mathit{ref_ty} \times \mathit{methd})\mathit{table} \\
\mathsf{fields} & :: & \mathit{prog} \times \mathit{tname} \Rightarrow ((\mathit{ename} \times \mathit{ref_ty}) \times \mathit{field})\mathit{list}
\end{array}$$

As Syme [21] points out, a naive recursive definition is not possible in HOL because the class hierarchy might be cyclic, which is ruled out for well-formed programs only. This leads to partial functions, which HOL does not support directly. Syme defines these functions as relations instead. In contrast, we have chosen to define them as proper functions, based on Slind's work on well-founded recursion [19]. We do not give the definitions, but only the recursion equations which we derive as easy consequences:

$$\begin{array}{l}
\mathsf{wf_prog}\ \Gamma \wedge \mathsf{iface}\ \Gamma\ I = \mathsf{Some}\ (\mathit{is},\mathit{ms}) \longrightarrow \\
\mathsf{imethd}\ (\Gamma,I) = \mathsf{let}\ \mathit{imethds} = (\lambda J.\ \mathsf{imethd}\ (\Gamma,J))\,``\,\mathsf{set}\ \mathit{is} \\
\qquad\quad \mathsf{in}\ (\mathsf{s2o} \circ (\mathsf{Un_tables}\ \mathit{imethds})) \oplus \\
\quad \mathsf{table}\ (\mathsf{map}\ (\lambda(s,\mathit{mh}).\ (s,\mathsf{IfaceT}\ I,\mathit{mh}))\ \mathit{ms})
\end{array}$$

$$\begin{array}{l}
\mathsf{wf_prog}\ \Gamma \wedge \mathsf{class}\ \Gamma\ C = \mathsf{Some}\ (\mathit{sc},\mathit{si},\mathit{fs},\mathit{ms}) \longrightarrow \\
\mathsf{cmethd}\ (\Gamma,C) = (\mathsf{case}\ \mathit{sc}\ \mathsf{of\ None} \quad \Rightarrow \mathsf{etable} \\
\qquad\qquad\qquad\qquad\qquad \mid \mathsf{Some}\ D \Rightarrow \mathsf{cmethd}\ (\Gamma,D)) \oplus \\
\quad \mathsf{table}\ (\mathsf{map}\ (\lambda(s,m).\ (s,(\mathsf{ClassT}\ C,m)))\ \mathit{ms})
\end{array}$$

$$\begin{array}{l}
\mathsf{wf_prog}\ \Gamma \wedge \mathsf{class}\ \Gamma\ C = \mathsf{Some}\ (\mathit{sc},\mathit{si},\mathit{fs},\mathit{ms}) \longrightarrow \\
\mathsf{fields}\ (\Gamma,C) = \mathsf{map}\ (\lambda(\mathit{fn},\mathit{ft}).\ ((\mathit{fn},\mathsf{ClassT}\ C),\mathit{ft}))\ \mathit{fs}\ @ \\
\quad (\mathsf{case}\ \mathit{sc}\ \mathsf{of\ None} \Rightarrow [] \mid \mathsf{Some}\ D \Rightarrow \mathsf{fields}\ (\Gamma,D))
\end{array}$$

where

$$\begin{array}{ll}
\mathsf{s2o}\ A & \stackrel{\mathsf{def}}{=} \mathsf{if}\ \exists_1 x.\ x{\in}A\ \mathsf{then\ Some}\ (\varepsilon x.\ x{\in}A)\ \mathsf{else\ None} \\
f``A & \stackrel{\mathsf{def}}{=} \{y.\ \exists x{\in}A.\ y = f\ x\} \\
\mathsf{Un_tables}\ \mathit{ts} & \stackrel{\mathsf{def}}{=} \lambda k.\ \bigcup t{\in}\mathit{ts}.\ \mathsf{case}\ t\ k\ \mathsf{of\ None} \;\Rightarrow \{\ \} \\
 & \qquad\qquad\qquad\qquad\qquad\quad \mid \mathsf{Some}\ x \Rightarrow \{x\}
\end{array}$$

4.1.3 Statements and expressions

We define statements (appearing in method bodies), expressions (appearing in statements), and literal values (appearing in expressions) as recursive datatypes.

Statements are reduced to their bare essentials. We do not formalize syntactic variants of conditionals and loops. Neither do we consider jumps like the *break* statement. The only non-standard statement is the "expression statement" Expr, which is evaluated for its side effects only. Assignments and method calls, both of which are expressions because they yield a value, can be turned into statements via Expr.

$$\begin{array}{lll}
\textbf{datatype}\ \mathit{stmt} & = & \mathsf{Skip} \\
 & \mid & \mathsf{Expr}\ \mathit{expr} \\
 & \mid & \mathit{stmt};\ \mathit{stmt} \\
 & \mid & \mathsf{If}\quad (\mathit{expr})\ \mathit{stmt}\ \mathsf{Else}\ \mathit{stmt} \\
 & \mid & \mathsf{While}(\mathit{expr})\ \mathit{stmt}
\end{array}$$

Concerning expressions, our formalization leaves out the standard unary and binary operators as their typing and semantics is straightforward. Creation of multi-dimensional arrays can be simulated with nested array creation. Because methods have just one local variable, namely the (single) parameter, we have given it the special name LVar. We have chosen not to introduce the general syntactic category of variables because the semantic treatment of local variables (including parameters), class instance variables, and array components differs considerably.

$$\begin{array}{ll}
\textbf{datatype}\ \mathit{expr} & \\
\quad = \mathsf{This} & \text{this} \\
\quad\ \mid \mathsf{New}\ \mathit{tname} & \text{class instance creation} \\
\quad\ \mid \mathsf{New}\ \mathit{ty}[\mathit{expr}] & \text{array creation} \\
\quad\ \mid (\mathit{ty})\mathit{expr} & \text{type cast} \\
\quad\ \mid \mathsf{Lit}\ \mathit{litval} & \text{literal} \\
\quad\ \mid \mathsf{LVar} & \text{local/param. access} \\
\quad\ \mid \mathsf{LVar}{:=}\mathit{expr} & \text{local/param. assign.} \\
\quad\ \mid \mathit{expr}\{\mathit{ref_ty}\}.\mathit{ename} & \text{field access} \\
\quad\ \mid \mathit{expr}\{\mathit{ref_ty}\}.\mathit{ename}{:=}\mathit{expr} & \text{field assignment} \\
\quad\ \mid \mathit{expr}[\mathit{expr}] & \text{array access} \\
\quad\ \mid \mathit{expr}[\mathit{expr}]{:=}\mathit{expr} & \text{array assignment} \\
\quad\ \mid \mathit{expr}.\mathit{mname}\{\mathit{ty}\}(\mathit{expr}) & \text{method call}
\end{array}$$

The terms in braces $\{\ldots\}$ above, called *type annotations*, are normally added by the compiler in order to implement the static, respectively dynamic, binding of fields and methods. We avoid distinguishing between the actual input language and the augmented language, because this would lead to a considerable amount of redundancy. Instead, we can safely assume that the annotations are added beforehand, as they are checked by the typing rules (in §4.2.3) anyway.

The definition of literal values is straightforward:

$$\begin{array}{ll}
\textbf{datatype}\ \mathit{litval} & \\
\qquad = \mathsf{Unit} & \text{dummy result of void methods} \\
\qquad\ \mid \mathsf{Null} & \text{null reference} \\
\qquad\ \mid \mathsf{Bool}\ \mathit{bool} & \text{Boolean value} \\
\qquad\ \mid \mathsf{Intg}\ \ \mathit{int} & \text{integer}
\end{array}$$

This definition is based on the HOL types of Boolean values (*bool*) and integers (*int*).

4.2 Type system

This section defines types, various ordering relations betwen types, and the typing rules for statements and expressions.

4.2.1 Types

We formalize BALI types as values of datatype *ty*, dividing them into primitive and reference types:

datatype *prim_ty*	primitive type
= Void	dummy type for void methods
| Boolean	Boolean type
| Integer	integer type

datatype *ref_ty*	reference type
= NullT	null type
| IfaceT *tname*	interface type
| ClassT *tname*	class type
| ArrayT *ty*	array type

datatype *ty*	type
= PrimT *prim_ty*	primitive type
| RefT *ref_ty*	reference type

In the sequel Iface I stands for RefT(IfaceT I), Class C for RefT(ClassT C) and T[] for RefT(ArrayT T).

4.2.2 Type relations

The relations between types depend on the interface and class hierarchy of a given program Γ, and are therefore expressed with reference to Γ. The direct subinterface ($\vdash \prec_{\mathsf{i}}^1$), subclass ($\vdash \prec_{\mathsf{c}}^1$) and implementation ($\vdash \leadsto^1$) relations are of type *prog* $\times$ *tname* $\times$ *tname* $\Rightarrow$ *bool* and are defined as follows:

$$\Gamma \vdash I \prec_{\mathsf{i}}^1 J \overset{\text{def}}{=} \mathsf{is\_iface}\ \Gamma\ I \ \wedge\ \mathsf{is\_iface}\ \Gamma\ J\ \wedge\ J \in \mathsf{set}\ (\mathsf{fst}\ (\mathsf{the}\ (\mathsf{iface}\ \Gamma\ I)))$$

$$\Gamma \vdash C \prec_{\mathsf{c}}^1 D \overset{\text{def}}{=} \mathsf{is\_class}\ \Gamma\ C \ \wedge\ \mathsf{is\_class}\ \Gamma\ D\ \wedge\ \mathsf{Some}\ D = \mathsf{fst}\ (\mathsf{the}\ (\mathsf{class}\ \Gamma\ C))$$

$$\Gamma \vdash C \leadsto^1 I \overset{\text{def}}{=} \mathsf{is\_class}\ \Gamma\ C \ \wedge\ \mathsf{is\_iface}\ \Gamma\ I\ \wedge\ \mathsf{Some}\ I = \mathsf{fst}\ (\mathsf{snd}\ (\mathsf{the}\ (\mathsf{class}\ \Gamma\ C)))$$

They are based on the auxiliary functions

$$\mathsf{is\_iface}\ \Gamma\ I \overset{\text{def}}{=} \mathsf{iface}\ \Gamma\ I \neq \mathsf{None}$$

$$\mathsf{is\_class}\ \Gamma\ C \overset{\text{def}}{=} \mathsf{class}\ \Gamma\ C \neq \mathsf{None}$$

The transitive (but not reflexive) closures $\vdash \prec_{\mathsf{i}}$ and $\vdash \prec_{\mathsf{c}}$ are defined as usual. There is also a kind of transitive closure of $\vdash \leadsto^1$ defined inductively:

$$\frac{\Gamma \vdash C \leadsto^1 J}{\Gamma \vdash C \leadsto J}$$

$$\frac{\Gamma \vdash C \leadsto^1 I;\ \Gamma \vdash I \prec_{\mathsf{i}} J}{\Gamma \vdash C \leadsto J} \qquad \frac{\Gamma \vdash C \prec_{\mathsf{c}}^1 D;\ \Gamma \vdash D \leadsto J}{\Gamma \vdash C \leadsto J}$$

The key relation is widening: $\Gamma \vdash S \preceq T$, where S and T are of type *ty*, means that S is a syntactic subtype of T, i.e. in any expression context (especially assignments and method invocations) expecting a value of type T, a value of type S may occur. Note that this does not necessarily mean that type S behaves like type T, but only that it has a syntactically compatible set of fields and methods. The widening relation is defined inductively as

$$\frac{\mathsf{is\_type}\ \Gamma\ T}{\Gamma \vdash T \preceq T} \qquad \frac{\Gamma \vdash C \leadsto I}{\Gamma \vdash \mathsf{Class}\ C \preceq \mathsf{Iface}\ I}$$

$$\frac{\mathsf{is\_iface}\ \Gamma\ I;\ \mathsf{is\_class}\ \Gamma\ \mathsf{Object}}{\Gamma \vdash \mathsf{Iface}\ I \preceq \mathsf{Class\ Object}}$$

$$\frac{\Gamma \vdash C \prec_{\mathsf{c}} D}{\Gamma \vdash \mathsf{Class}\ C \preceq \mathsf{Class}\ D} \qquad \frac{\Gamma \vdash I \prec_{\mathsf{i}} J}{\Gamma \vdash \mathsf{Iface}\ I \preceq \mathsf{Iface}\ J}$$

$$\frac{\mathsf{is\_type}\ \Gamma\ (\mathsf{RefT}\ R)}{\Gamma \vdash \mathsf{RefT\ NullT} \preceq \mathsf{RefT}\ R}$$

$$\frac{\mathsf{is\_type}\ \Gamma\ T;\ \mathsf{is\_class}\ \Gamma\ \mathsf{Object}}{\Gamma \vdash T[] \preceq \mathsf{Class\ Object}}$$

$$\frac{\Gamma \vdash \mathsf{RefT}\ S \preceq \mathsf{RefT}\ T}{\Gamma \vdash (\mathsf{RefT}\ S)[] \preceq (\mathsf{RefT}\ T)[]}$$

where

$$\begin{array}{ll} \mathsf{is\_type}\ \Gamma\ (\mathsf{PrimT}\ \_) & = \mathsf{True} \\ \mathsf{is\_type}\ \Gamma\ (\mathsf{RefT\ NullT}) & = \mathsf{True} \\ \mathsf{is\_type}\ \Gamma\ (\mathsf{Iface}\ I) & = \mathsf{is\_iface}\ \Gamma\ I \\ \mathsf{is\_type}\ \Gamma\ (\mathsf{Class}\ C) & = \mathsf{is\_class}\ \Gamma\ C \\ \mathsf{is\_type}\ \Gamma\ (T[]) & = \mathsf{is\_type}\ \Gamma\ T \end{array}$$

Object is the name of the top of the class hierarchy.

To allow for type casting we also have the relation $\vdash \preceq_?$, where $\Gamma \vdash S \preceq_? T$ means that a value of type S may be cast to type T:

$$\frac{\Gamma \vdash S \preceq T}{\Gamma \vdash S \preceq_? T} \qquad \frac{\Gamma \vdash C \prec_{\mathsf{c}} D}{\Gamma \vdash \mathsf{Class}\ D \preceq_? \mathsf{Class}\ C}$$

$$\frac{\mathsf{is\_class}\ \Gamma\ C;\ \mathsf{is\_iface}\ \Gamma\ I}{\Gamma \vdash \mathsf{Class}\ C \preceq_? \mathsf{Iface}\ I} \qquad \frac{\mathsf{is\_iface}\ \Gamma\ I;\ \mathsf{is\_class}\ \Gamma\ C}{\Gamma \vdash \mathsf{Iface}\ I \preceq_? \mathsf{Class}\ C}$$

$$\frac{\mathsf{is\_class}\ \Gamma\ \mathsf{Object};\ \mathsf{is\_type}\ \Gamma\ T}{\Gamma \vdash \mathsf{Class\ Object} \preceq_? T[]}$$

$$\frac{\begin{array}{c}\mathsf{is\_iface}\ \Gamma\ J;\ \neg\Gamma \vdash I \prec_{\mathsf{i}} J; \\ \mathsf{imethd}\ (\Gamma,I)\ \mathsf{hiding}\ \mathsf{imethd}\ (\Gamma,J)\ \mathsf{entails} \\ (\lambda(m_1,rT_1)\ (m_2,rT_2).\ \Gamma \vdash rT_1 \preceq rT_2)\end{array}}{\Gamma \vdash \mathsf{Iface}\ I \preceq_? \mathsf{Iface}\ J}$$

$$\frac{\Gamma \vdash \mathsf{RefT}\ S \preceq_? \mathsf{RefT}\ T}{\Gamma \vdash (\mathsf{RefT}\ S)[] \preceq_? (\mathsf{RefT}\ T)[]}$$

4.2.3 Typing rules

Now we come to the actual type checking rules. An *environment* consists of a global part, namely a program, and a local part, namely the type of the (single) current method parameter and the current class, i.e. the type of This:

env = *prog* $\times$ *ty* $\times$ *tname*

The well-typedness of statements ($\vdash :: \Diamond$) and the typing of expressions ($\vdash ::$) are defined inductively relative to an environment.

$$\begin{array}{lll} \_ \vdash \_ :: \Diamond & :: \mathit{env} \Rightarrow \mathit{stmt} & \Rightarrow \mathit{bool} \\ \_ \vdash \_ :: \_ & :: \mathit{env} \Rightarrow \mathit{expr} \Rightarrow \mathit{ty} & \Rightarrow \mathit{bool} \end{array}$$

The rules for statements are obvious:

$$\frac{}{E \vdash \mathsf{Skip} :: \Diamond} \qquad \frac{E \vdash e :: T}{E \vdash \mathsf{Expr}\ e :: \Diamond} \qquad \frac{E \vdash s_1 :: \Diamond;\ E \vdash s_2 :: \Diamond}{s_1;\ s_2 :: \Diamond}$$

$$\frac{E \vdash e :: \mathsf{PrimT\ Boolean};\ E \vdash s_1 :: \Diamond;\ E \vdash s_2 :: \Diamond}{\mathsf{If}(e)\ s_1\ \mathsf{Else}\ s_2 :: \Diamond}$$

$$\frac{E \vdash e :: \mathsf{PrimT\ Boolean};\ E \vdash s :: \Diamond}{E \vdash \mathsf{While}(e)\ s :: \Diamond}$$

More interesting are the rules for expressions:

$$\frac{\text{is\_class (prg } E\text{) (thisT } E\text{)}}{E\vdash \text{This::Class (thisT } E\text{)}} \qquad \frac{\text{is\_class (prg } E\text{) } C}{E\vdash \text{New } C\text{::Class } C}$$

$$\frac{\text{is\_type (prg } E\text{) } T;\; E\vdash i\text{::PrimT Integer}}{E\vdash \text{New } T[i]\text{::}T[]}$$

$$\frac{E\vdash e\text{::}T;\;\text{prg } E\vdash T\preceq_? T'}{E\vdash (T')e\text{::}T'} \qquad \frac{}{E\vdash \text{Lit } x\text{::typeof } (\lambda a.\text{ None})\; x}$$

$$\frac{\text{is\_type (prg } E\text{) (localT } E\text{)}}{E\vdash \text{LVar::localT } E}$$

$$\frac{E\vdash v\text{::}T;\;\text{prg } E\vdash T\preceq \text{localT } E}{E\vdash \text{LVar:=}v\text{::}T}$$

$$\frac{E\vdash e\text{::Class } C;\;\text{cfield (prg } E,C\text{) } fn = \text{Some } (fd,fT)}{E\vdash e\{fd\}.fn\text{::}fT}$$

$$\frac{E\vdash e\{fd\}.fn\text{::}T;\; E\vdash v\text{::}T';\;\text{prg } E\vdash T'\preceq T}{E\vdash e\{fd\}.fn\text{:=}v\text{::}T'}$$

$$\frac{E\vdash a\text{::}T[];\; E\vdash i\text{::PrimT Integer}}{E\vdash a[i]\text{::}T}$$

$$\frac{E\vdash a[i]\text{::}T;\; E\vdash v\text{::}T';\;\text{prg } E\vdash T'\preceq T}{E\vdash a[i]\text{:=}v\text{::}T'}$$

$$\frac{\begin{array}{c}E\vdash e\text{::RefT } T;\; E\vdash p\text{::}pT;\\ \text{max\_spec (prg } E\text{) } T\;(mn,pT) = \{((md,rT),pT')\}\end{array}}{E\vdash e.mn\{pT'\}(p)\text{::}rT}$$

The rules are based on the auxiliary functions given below. The function cfield is a variant of fields implementing a field lookup that is based on the field name alone in contrast to a combination of field name and defining class. So in the above typing rule for field access, equal field names hide each other, while at run-time all fields are accessible, using the defining class as an additional search key.

$$\begin{array}{ll}\text{prg} & (\Gamma,lT,tT) = \Gamma\\ \text{localT} & (\Gamma,lT,tT) = lT\\ \text{thisT} & (\Gamma,lT,tT) = tT\end{array}$$

$$\begin{array}{lll}\text{typeof } dt \text{ Unit} & = & \text{PrimT Void}\\ \text{typeof } dt \text{ Null} & = & \text{RefT NullT}\\ \text{typeof } dt \text{ (Bool } b\text{)} & = & \text{PrimT Boolean}\\ \text{typeof } dt \text{ (Intg } i\text{)} & = & \text{PrimT Integer}\\ \text{typeof } dt \text{ (Addr } a\text{)} & = & dt\; a\end{array}$$

$$\text{cfield} \stackrel{\text{def}}{=} \text{table} \circ (\text{map } (\lambda((fn,fd),ft).\;(fn,(fd,ft)))) \circ \text{fields}$$

The typing rules are rather straightforward, except for the type annotations $\{\ldots\}$, which are used to implement static binding for fields and to resolve overloaded method names statically. The rules for field access and method call determine how to compute these annotations. They read as follows.

A field access $e\{fd\}.fn$ is annotated correctly if fd is the first defining class for a field with name fn when searching the class hierarchy (using cfield) starting from the static type Class C of e. The annotation $\{fd\}$ will be used subsequently to access the field (the one just found) via the pair (fn,fd).

A method call $e.mn\{pT'\}(p)$ is type-correct only if the function max_spec determining the set of "maximally specific" [12, 15.11.2] methods for reference type T (as defined below) yields exactly one method entry. In this case, the call is annotated by pT', which is the argument type of the most specific method mn applicable according to the static types T of e and pT of p. So the dynamic method lookup at run-time can be based on the signature (mn,pT').

$$\begin{array}{l}\text{max\_spec } \Gamma\; rT\; sig \stackrel{\text{def}}{=} \{m \mid m \in \text{appl\_methds } \Gamma\; rT\; sig \wedge\\ \qquad (\forall m' \in \text{appl\_methds } \Gamma\; rT\; sig.\\ \qquad\quad \text{more\_spec } \Gamma\; m'\; m \longrightarrow m' = m)\}\end{array}$$

$$\begin{array}{l}\text{appl\_methds } \Gamma\; rT\; (mn,\; pT) \stackrel{\text{def}}{=} \{(m,pT') \mid\\ \quad \text{mhead } \Gamma\; rT\; (mn,\; pT') = \text{Some } m \wedge \Gamma\vdash pT\preceq pT'\}\end{array}$$

$$\begin{array}{l}\text{more\_spec } \Gamma\; ((d,r),p)\; ((d',r'),p') \stackrel{\text{def}}{=}\\ \qquad \Gamma\vdash \text{RefT } d\preceq \text{RefT } d' \wedge \Gamma\vdash p\preceq p'\end{array}$$

$$\begin{array}{l}\text{mhead } \Gamma\; t\; sig \stackrel{\text{def}}{=} \text{case } t \text{ of NullT} \Rightarrow \text{None}\\ \qquad \mid \text{IfaceT } I \Rightarrow \text{imethd } (\Gamma,I)\; sig\\ \qquad \mid \text{ClassT } C \Rightarrow \text{option\_map } (\lambda(md,(mh,mb)).\;(md,mh))\\ \qquad\qquad (\text{cmethd } (\Gamma,C)\; sig)\\ \qquad \mid \text{ArrayT } T \Rightarrow \text{None}\end{array}$$

4.3 Well-formedness

A program must satisfy a number of well-formedness conditions concerning global properties of all declarations. The conditions are expressed as predicates on field, method, interface, and class declarations, as follows.

A field declaration is well-formed iff its type exists:

$$\text{wf\_fdecl } \Gamma\; (fn,ft) \stackrel{\text{def}}{=} \text{is\_type } \Gamma\; ft$$

A method declaration is well-formed if its argument and result types are defined. If the declaration appears in a class, additionally its body has to be well-typed (in the static context of its parameter type and the current class) and its result expression have a type that widens to the result type:

$$\text{wf\_mhead } \Gamma\; (mn,pT)\; rT \stackrel{\text{def}}{=} \text{is\_type } \Gamma\; pT \wedge \text{is\_type } \Gamma\; rT$$

$$\begin{array}{l}\text{wf\_cmdecl } \Gamma\; C\; (sig,rT,blk,res) \stackrel{\text{def}}{=} \text{wf\_mhead } \Gamma\; sig\; rT \wedge\\ \quad \text{let } E{=}(\Gamma,\text{snd } sig,C) \text{ in } E\vdash blk\text{::}\diamond \wedge \exists T.\\ \qquad\qquad E\vdash res\text{::}T \wedge \Gamma\vdash T\preceq rT\end{array}$$

More complex conditions are required for well-formed interface and class declarations. The name of a well-formed interface declaration is not a class name. All superinterfaces exist and are not subinterfaces at the same time. All methods newly declared in the interface are named uniquely and are well-formed. Furthermore, there are no ambiguously inherited methods, and any method hiding a method of a superinterface has a compatible result type:

$$\begin{array}{l}\text{wf\_idecl } \Gamma\; (I,(is,ms)) \stackrel{\text{def}}{=} \neg\; \text{is\_class } \Gamma\; I \wedge\\ (\forall J{\in}\text{set } is.\; \text{is\_iface } \Gamma\; J \wedge \neg\; \Gamma\vdash J\prec_{\text{i}} I) \wedge\\ \text{unique } ms \wedge (\forall (sig,mh){\in}\text{set } ms.\; \text{wf\_mhead } \Gamma\; sig\; mh) \wedge\\ \text{let } mtab = \text{Un\_tables } ((\lambda J.\; \text{imethd } (\Gamma,J))\text{ `` set } is) \text{ in}\\ \qquad (\forall sig.\; \text{atmost1 } (mtab\; sig)) \wedge\\ \qquad (\text{table } ms) \text{ hiding } (\text{s2o} \circ mtab) \text{ entails}\\ \qquad (\lambda rT_1\; (md,rT_2).\; \Gamma\vdash rT_1\preceq rT_2)\end{array}$$

where

$$\begin{array}{l}\text{unique } t \stackrel{\text{def}}{=} \forall (x_1,y_1){\in}\text{set } t.\; \forall (x_2,y_2){\in}\text{set } t.\\ \qquad\qquad x_1 = x_2 \longrightarrow y_1 = y_2\end{array}$$

$$\text{atmost1 } S \stackrel{\text{def}}{=} \forall x{\in}S.\; \forall y{\in}S.\; x = y$$

Similarly, the name of a well-formed class declaration is not an interface name. If the class implements an interface, this interface exists, and for any method of the interface, the class provides an implementing method with a possibly narrower return type. All fields and methods newly declared in the class are named uniquely and are well-formed. If the class is not Object, it refers to an existing superclass, which is not subclass of the current class. Furthermore, any method overriding a method of the superclass has a compatible result type:

$\mathsf{wf\_cdecl}\ \Gamma\ (C,(sc,si,fs,ms)) \stackrel{\mathsf{def}}{=} \neg\ \mathsf{is\_iface}\ \Gamma\ C \wedge$
$(\forall I.\ si = \mathsf{Some}\ I \longrightarrow \mathsf{is\_iface}\ \Gamma\ I \wedge$
$\quad \forall s\ m\ \ rT_1.\ \mathsf{imethd}\ (\Gamma,I\ )\ s = \mathsf{Some}\ (m\ ,rT_1) \longrightarrow$
$\quad \exists b\ m'\ rT_2.\ \mathsf{cmethd}\ (\Gamma,C)\ s = \mathsf{Some}\ (m',rT_2,b) \wedge$
$\qquad\qquad \Gamma \vdash rT_2 \preceq rT_1\ ) \wedge$
$\mathsf{unique}\ fs\ \wedge (\forall f\ \in\mathsf{set}\ fs.\ \ \mathsf{wf\_fdecl}\quad \Gamma\quad f\ ) \wedge$
$\mathsf{unique}\ ms \wedge (\forall m\in\mathsf{set}\ ms.\ \mathsf{wf\_cmdecl}\ \Gamma\ C\ m) \wedge$
$\ (\mathsf{case}\ sc\ \mathsf{of}\ \mathsf{None}\quad \Rightarrow C = \mathsf{Object}$
$\qquad\quad |\ \mathsf{Some}\ D \Rightarrow \mathsf{is\_class}\ \Gamma\ D \wedge \neg\ \Gamma \vdash D \prec_{\mathsf{C}} C \wedge$
$\qquad\qquad (\mathsf{table}\ ms)\ \mathsf{hiding}\ (\mathsf{cmethd}\ (\Gamma,D))\ \mathsf{entails}$
$\qquad\qquad (\lambda(rT_1,b)\ (m,(rT_2,b')).\ \Gamma \vdash rT_1 \preceq rT_2)$

Finally, a well-formed program contains the standard declaration of Object, namely the empty class declaration ObjectC $\stackrel{\mathsf{def}}{=}$ (Object,(None,None,[],[])). All its declared interfaces and classes are named uniquely and are well-formed:

$\mathsf{wf\_prog}\ \Gamma \stackrel{\mathsf{def}}{=} \mathsf{ObjectC} \in \mathsf{set}\ (\mathsf{snd}\ \Gamma) \wedge$
$\ \mathsf{unique}\ (\mathsf{fst}\ \ \Gamma) \wedge \forall i\in\mathsf{set}\ (\mathsf{fst}\ \ \Gamma).\ \mathsf{wf\_idecl}\ \ \Gamma\ i) \wedge$
$\ \mathsf{unique}\ (\mathsf{snd}\ \Gamma) \wedge \forall c\in\mathsf{set}\ (\mathsf{snd}\ \Gamma).\ \mathsf{wf\_cdecl}\ \Gamma\ c)$

4.4 Operational semantics

In this section, we describe the notion of a state and give the evaluation rules for expressions and statements.

4.4.1 State

A *state* consists of an optional exception (of type *xcpt*, which currently consists of system exceptions like NullPointerXcpt only), a heap, and the current invocation frame, which is the value of the (single) parameter and the This pointer:

$state = xcpt\ option \times heap \times val \times loc$

A value is either a literal value or a location, i.e. an abstract non-null pointer to an object; a heap is a mapping from locations to objects:

$$heap = (loc,\ obj)table$$
datatype $val = \mathsf{Val}\ litval\ |\ \mathsf{Addr}\ loc$

The type *loc* of locations is not further specified.

An object is either a class instance, modeled as a pair of its class name and a table mapping pairs of a field name and the defining class to values, or an array, modeled as a pair of its component type and a table mapping integers to values.

$$fields = (ename \times ref\_ty,\ val)table$$
$$components = (int\qquad\qquad ,\ val)table$$
datatype $obj = \mathsf{Obj}\ tname\ fields\ |\ \mathsf{Arr}\ ty\ components$

There is a number of auxiliary functions handling the state, namely:

- the_Addr :: $val \Rightarrow loc$ is defined such that the_Addr (Addr a) = a;
- the_Obj :: $obj\ option \Rightarrow tname \times fields$ with the_Obj (Obj $C\ fs$) = (C,fs);
- the_Arr :: $obj\ option \Rightarrow ty \times components$ with the_Arr (Arr $T\ cs$) = (T,cs);
- heap $(h,l,t) \stackrel{\mathsf{def}}{=} h$, local $(h,l,t) \stackrel{\mathsf{def}}{=} l$, this $(h,l,t) \stackrel{\mathsf{def}}{=} t$
- obj_ty $obj \stackrel{\mathsf{def}}{=}$ case obj of Obj $C\ fs \Rightarrow$ Class C
  | Arr $T\ cs \Rightarrow T$[]
- raise_if $c\ x\ xo \stackrel{\mathsf{def}}{=}$
  if $c \wedge (xo = \mathsf{None})$ then Some x else xo
- np $v \stackrel{\mathsf{def}}{=}$ raise_if (v = Null) NullPointerXcpt
- c_hupd $h'\ (xo,(h,l,t)) \stackrel{\mathsf{def}}{=}$
  if xo = None then (None,(h',l,t)) else $(xo,(h,l,t))$
- cast_ok $\Gamma\ T\ h\ v \stackrel{\mathsf{def}}{=} (\exists pt.\ T = \mathsf{PrimT}\ pt) \vee$
  $\Gamma \vdash$ obj_ty (the (h (the_Addr v))) $\preceq T$
- default_val (PrimT Void) = Unit
  default_val (PrimT Boolean) = Bool *False*
  default_val (PrimT Integer) = Intg *0*
  default_val (RefT r) = Null

4.4.2 Evaluation rule format

We define the operational semantics of statements and expressions via mutually inductive rules. To obtain a concise description, we use an evaluation semantics rather than a transition semantics.

- $\Gamma \vdash (x,\sigma)\ -s\rightarrow\ (x',\sigma')$ means that execution of statement s transforms state (x,σ) into (x',σ').
- $\Gamma \vdash (x,\sigma)\ -e\rhd v\rightarrow\ (x',\sigma')$ means that expression e evaluates to value v, transforming (x,σ) into (x',σ').

Strictly speaking it is neither necessary to include an exception in the start state of a computation nor the This pointer in the final state (because This does not change). Similarly, an expression needs only return either a value or an exception, but not both. However, the symmetry achieved by our slightly redundant model simplifies the rules considerably. In particular, in many rules we can avoid case distinctions on whether exceptions occur in intermediate states, which would cause the rules to be split. As a result, there is exactly one rule for each syntactic construct.

For both statements and expressions there is a general rule defining that exceptions simply propagate:

$$\frac{}{\Gamma \vdash (\mathsf{Some}\ xc,\sigma)\ -s\rightarrow\ (\mathsf{Some}\ xc,\sigma)}$$

$$\frac{}{\Gamma \vdash (\mathsf{Some}\ xc,\sigma)\ -e\rhd\mathsf{arbitrary}\rightarrow\ (\mathsf{Some}\ xc,\sigma)}$$

All other rules can assume that in their concerning initial state no exception has been thrown. For such states, we define the abbreviation Norm σ, which stands for (None,σ).

4.4.3 Execution of statements

The rules for statements are obvious:

$$\frac{}{\Gamma \vdash \mathsf{Norm}\ \sigma\ -\mathsf{Skip}\rightarrow\ \mathsf{Norm}\ \sigma} \qquad \frac{\Gamma \vdash \mathsf{Norm}\ \sigma_0\ -e\rhd v\rightarrow\ \sigma_1}{\Gamma \vdash \mathsf{Norm}\ \sigma_0\ -\mathsf{Expr}\ e\rightarrow\ \sigma_1}$$

$$\frac{\Gamma \vdash \mathsf{Norm}\ \sigma_0\ -s_1\ \rightarrow \sigma_1;\ \ \Gamma \vdash \sigma_1\ -s_2\rightarrow\ \sigma_2}{\Gamma \vdash \mathsf{Norm}\ \sigma_0\ -s_1;\ s_2\rightarrow\ \sigma_2}$$

$$\frac{\Gamma \vdash \mathsf{Norm}\ \sigma_0\ -e\rhd v\rightarrow\ \sigma_1;\quad \Gamma \vdash \sigma_1\ -\mathsf{if\ the\_Bool}\ v\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2\rightarrow\ \sigma_2}{\Gamma \vdash \mathsf{Norm}\ \sigma_0\ -\mathsf{If}(e)\ s_1\ \mathsf{Else}\ s_2\rightarrow\ \sigma_2}$$

$$\frac{\Gamma \vdash \mathsf{Norm}\ \sigma_0\ -\mathsf{If}(e)\ (s;\ \mathsf{While}(e)\ s)\ \mathsf{Else\ Skip}\rightarrow\ \sigma_1}{\Gamma \vdash \mathsf{Norm}\ \sigma_0\ -\mathsf{While}(e)\ s\rightarrow\ \sigma_1}$$

4.4.4 Evaluation of expressions

In contrast, the evaluation rules for expressions deserve some comments.

The value of This is a component of the state:

$$\frac{}{\Gamma \vdash \mathsf{Norm}\ \sigma\ -\mathsf{This}\triangleright \mathsf{Addr}\ (\mathsf{this}\ \sigma)\rightarrow\ \mathsf{Norm}\ \sigma}$$

Creating a new class instance means picking a new address a (i.e. $h\ a = \mathsf{None}$) and updating the heap at that address with an object, the fields of which are initialized with default values according to their types:

$$\frac{\begin{array}{c} h = \mathsf{heap}\ \sigma;\ \ h\ a = \mathsf{None};\ \ h' = h[a{:=}\ \mathsf{Obj}\ C\ (\mathsf{table} \\ (\mathsf{map}\ (\lambda(f,fT).\ (f,\mathsf{default\_val}\ fT))\ (\mathsf{fields}\ (\Gamma,C))))] \end{array}}{\Gamma \vdash \mathsf{Norm}\ \sigma\ -\mathsf{New}\ C\triangleright \mathsf{Addr}\ a\rightarrow\ \mathsf{c\_hupd}\ h'\ (\mathsf{Norm}\ \sigma)}$$

Creating a new array means picking a new address, updating the heap with an array, the components of which are initialized with default values, and raising an exception if the length of the array is negative:

$$\frac{\begin{array}{c} \Gamma \vdash \mathsf{Norm}\ \sigma_0\ -e\triangleright i'\rightarrow\ (x_1,\sigma_1);\ \ i = \mathsf{the\_Int}\ i'; \\ h = \mathsf{heap}\ \sigma_1;\ \ h\ a = \mathsf{None};\ \ h'{=}h[a{:=}\mathsf{Arr}\ T\ (\lambda j. \\ \mathsf{if}\ 0{\le}j \wedge j{<}i\ \mathsf{then\ Some}\ (\mathsf{default\_val}\ T)\ \mathsf{else\ None})]; \\ x_1'{=}\mathsf{raise\_if}\ (i{<}0)\ \mathsf{NegArrSizeXcpt}\ x_1 \end{array}}{\Gamma \vdash \mathsf{Norm}\ \sigma_0\ -\mathsf{New}\ T[e]\triangleright \mathsf{Addr}\ a\rightarrow\ \mathsf{c\_hupd}\ h'\ (x_1',\sigma_1)}$$

Type casts simply return their argument value, but raise an exception if its dynamic type happens to be unsuitable:

$$\frac{\begin{array}{c} \Gamma \vdash \mathsf{Norm}\ \sigma_0\ -e\triangleright v\rightarrow\ (x_1,s_1); \\ x_1' = \mathsf{raise\_if}(\neg \mathsf{cast\_ok}\ \Gamma\ T\ (\mathsf{heap}\ s_1)\ v)\ \mathsf{ClassCastXcpt}\ x_1 \end{array}}{\Gamma \vdash \mathsf{Norm}\ \sigma_0\ -(T)e\triangleright v\rightarrow\ (x_1',s_1)}$$

A literal value is simply returned:

$$\frac{}{\Gamma \vdash \mathsf{Norm}\ \sigma\ -\mathsf{Lit}\ v\triangleright v\rightarrow\ \mathsf{Norm}\ \sigma}$$

An access to LVar reads from the corresponding state component:

$$\frac{}{\Gamma \vdash \mathsf{Norm}\ \sigma\ -\mathsf{LVar}\triangleright \mathsf{local}\ \sigma\rightarrow\ \mathsf{Norm}\ \sigma}$$

An assignment to LVar updates the state, in case the subexpression does not raise an exception:

$$\frac{\begin{array}{c} \Gamma \vdash \mathsf{Norm}\ \sigma\ -e\triangleright v\rightarrow\ \ (x,(h,l,t)); \\ l' = (\mathsf{if}\ x = \mathsf{None\ then}\ v\ \mathsf{else}\ l) \end{array}}{\Gamma \vdash \mathsf{Norm}\ \sigma\ -\mathsf{LVar}{:=}e\triangleright v\rightarrow\ (x,(h,l',t))}$$

A field access reads from a field of the given object, checking for null pointer access:

$$\frac{\begin{array}{c} \Gamma \vdash \mathsf{Norm}\ \sigma_0\ -e\triangleright a'\rightarrow\ (x_1,\sigma_1); \\ v = \mathsf{the}\ (\mathsf{snd}\ (\mathsf{the\_Obj}\ (\mathsf{heap}\ \sigma_1\ (\mathsf{the\_Addr}\ a')))\ (fn,T)) \end{array}}{\Gamma \vdash \mathsf{Norm}\ \sigma_0\ -e\{T\}.fn\triangleright v\rightarrow\ (\mathsf{np}\ a'\ x_1,\sigma_1)}$$

A field assignment acts accordingly:

$$\frac{\begin{array}{c} \Gamma \vdash \mathsf{Norm}\ \sigma_0\ -e_1\triangleright a'\rightarrow\ (x_1,\sigma_1);\ \ a = \mathsf{the\_Addr}\ a'; \\ \Gamma \vdash (\mathsf{np}\ a'\ x_1,\sigma_1)\ -e_2\triangleright v\ \rightarrow\ (x_2,\sigma_2); \\ h = \mathsf{heap}\ \sigma_2;\ \ (c,fs) = \mathsf{the\_Obj}\ (h\ a); \\ h' = h[a{:=}\mathsf{Obj}\ c\ (fs[(fn,T){:=}v])] \end{array}}{\Gamma \vdash \mathsf{Norm}\ \sigma_0\ -(e_1\{T\}.fn{:=}e_2)\triangleright v\rightarrow\ \mathsf{c\_hupd}\ h'\ (x_2,\sigma_2)}$$

An array access reads a component from the given array, but raises an exception if the index is invalid:

$$\frac{\begin{array}{c} \Gamma \vdash \mathsf{Norm}\ \sigma_0\ -e_1\triangleright a'\rightarrow\ \sigma_1;\ \ \Gamma \vdash \sigma_1\ -e_2\triangleright i'\rightarrow\ (x_2,\sigma_2); \\ vo = \mathsf{snd}\ (\mathsf{the\_Arr}\ (\mathsf{heap}\ \sigma_2\ (\mathsf{the\_Addr}\ a')))\ (\mathsf{the\_Int}\ i'); \\ x_2' = \mathsf{raise\_if}\ (vo = \mathsf{None})\ \mathsf{IndOutBoundXcpt}\ (\mathsf{np}\ a'\ x_2) \end{array}}{\Gamma \vdash \mathsf{Norm}\ \sigma_0\ -e_1[e_2]\triangleright \mathsf{the}\ vo\rightarrow\ (x_2',\sigma_2)}$$

Similarly, an array assignment updates the appropriate component, but has to check the typing:

$$\frac{\begin{array}{c} \Gamma \vdash \mathsf{Norm}\ \sigma_0\ -e_1\triangleright a'\rightarrow\ \sigma_1;\ \ a = \mathsf{the\_Addr}\ a'; \\ \Gamma \vdash \sigma_1\ -e_2\triangleright i'\rightarrow\ \sigma_2;\ \ i = \mathsf{the\_Int}\ i'; \\ \Gamma \vdash \sigma_2\ -e_3\triangleright v\ \rightarrow\ (x_3,\sigma_3);\ \ h = \mathsf{heap}\ \sigma_3; \\ (T,cs) = \mathsf{the\_Arr}\ (h\ a);\ \ h' = h[a{:=}\mathsf{Arr}\ T\ (cs[i{:=}v])]; \\ x_3' = \mathsf{raise\_if}\ (\neg \mathsf{cast\_ok}\ \Gamma\ T\ h\ v)\ \mathsf{ArrStoreXcpt}\ (\\ \mathsf{raise\_if}\ (cs\ i = \mathsf{None})\ \mathsf{IndOutBoundXcpt}\ (\mathsf{np}\ a'\ x_3)) \end{array}}{\Gamma \vdash \mathsf{Norm}\ \sigma_0\ -(e_1[e_2]{:=}e_3)\triangleright v\rightarrow\ \mathsf{c\_hupd}\ h'\ (x_3',\sigma_3)}$$

The most complex rule is the one for method invocation: after evaluating e to the target location a and p to the parameter value pv, the block blk and the result expression res of method mn with argument type T are extracted from the program Γ (using the dynamic type $dynT$ of the object stored at a). After executing blk and res in the new invocation frame built from pv and a, the old invocation frame is restored and the result value v returned:

$$\frac{\begin{array}{c} \Gamma \vdash \mathsf{Norm}\ \sigma_0\ -e\triangleright a'\rightarrow\ \sigma_1;\ \ a = \mathsf{the\_Addr}\ a'; \\ \Gamma \vdash \sigma_1\ -p\triangleright pv\rightarrow\ (x_2,(h,l,t));\ \ dynT = \mathsf{fst}(\mathsf{the\_Obj}\ (h\ a)); \\ (md,mh,blk,res) = \mathsf{the}\ (\mathsf{cmethd}\ (\Gamma,dynT)\ (mn,T)); \\ \Gamma \vdash (\mathsf{np}\ a'\ x_2,(h,pv,a))\ -blk\rightarrow \sigma_3; \\ \Gamma \vdash \sigma_3\ -res\triangleright v\ \rightarrow\ (x_4,\sigma_4) \end{array}}{\Gamma \vdash \mathsf{Norm}\ \sigma_0\ -(e.mn\{T\}(p))\triangleright v\rightarrow\ (x_4,(\mathsf{heap}\ \sigma_4,l,t))}$$

Note that the rules are defined carefully in order to be applicable in all situations, even not type-correct ones (e.g. the_Addr (Val (Bool b)) yields an arbitrary value). A "defensive" evaluation throwing some artificial exception in case of type mismatches, which would require additional overhead, is not necessary.

5 Proof of type soundness

This section discusses the type soundness theorem and its crucial lemmas. As the necessity of certain lemmas emerges quite naturally, it is not surprising that many of them are similar to those given by Drossopoulou and Eisenbach [10]. On the other hand, the proof principles we use are sometimes rather different from those outlined in their earlier paper [9], some of which are inadequate.

5.1 Lemmas about the type relations

There are two non-trivial lemmas concerning the type relations of BALI, namely the well-foundedness wf of the (converse) subinterface and subclass relations

$$\begin{array}{l} \mathsf{wf\_prog}\ \Gamma \longrightarrow \mathsf{wf}\ (\lambda(J\ ,I\ ).\ \Gamma \vdash I\ \prec_{\mathsf{i}}\ J\ ) \\ \qquad\qquad\quad \wedge\ \mathsf{wf}\ (\lambda(D,C).\ \Gamma \vdash C{\prec_{\mathsf{C}}}\,D) \end{array}$$

and the frequently used transitivity of the widening relation:

$$\mathsf{wf\_prog}\ \Gamma \longrightarrow \Gamma \vdash S{\preceq}\ U \wedge \Gamma \vdash U{\preceq}\ T \longrightarrow \Gamma \vdash S{\preceq}\ T$$

The two relations are well-founded because they are finite and acyclic, where the former is a consequence of representing class and interface declarations as lists, and the latter follows from the irreflexivity of the relations, which in turn follows from the well-formedness of the classes and interfaces implied by the well-formedness of the whole program.

The well-foundedness facts are necessary for deriving the recursion equations for the functions that traverse the type hierarchy of a program (see §4.1) and also give rise to induction principles for the (direct) subinterface and subclass relations, e.g.

$$\frac{\begin{array}{c}\mathsf{wf\_prog}\ \Gamma;\ P\ \mathsf{Object};\\ \forall C\ D.\ C \neq \mathsf{Object} \wedge \Gamma \vdash C \prec^1_{\mathsf{C}} D \wedge \ldots \wedge P\ D \longrightarrow P\ C\end{array}}{\forall E.\ \mathsf{is\_class}\ \Gamma\ E \longrightarrow P\ E}$$

Most lemmas like transitivity of $\vdash\ \preceq$, as well as auxiliary properties for deriving them, typically rely on several well-formedness conditions and are usually proved by rule induction on the type relation involved, or by applying the induction principles just mentioned.

5.2 Lemmas about fields and methods

For the type-safety of field accesses and method calls, characteristic lemmas concerning the field lookup and method lookup are required. They are used to relate the (static) types of fields and methods, as determined at compile-time, to the actual (dynamic) types that occur at run-time.

For example, fields correctly referred to at compile-time must be found at run-time. More formally, if a field access $e\{T\}.fn$ with $E \vdash e{::}\mathsf{Class}\ C$ statically refers to a field of type fT defined in the reference type (some class) T, within an instance of some class C' which may be a subclass of C the field can be referred to (dynamically) using the same name and its defining class. In particular, there is no dynamic binding for fields. This fact requires the following lemma:

$$\begin{array}{l}\mathsf{wf\_prog}\ \Gamma\ \wedge\ \mathsf{cfield}\ (\Gamma,C)\ fn = \mathsf{Some}\ (fd,fT) \wedge\\ \Gamma \vdash \mathsf{Class}\ C' \preceq \mathsf{Class}\ C \longrightarrow\\ \quad \mathsf{table}\ (\mathsf{fields}\ (\Gamma,C'))\ (fn,fd) = \mathsf{Some}\ fT\end{array}$$

Concerning method calls, a similar requirement preventing 'method not understood' errors can be formalized: if a method call of the form $e.mn\{pT\}(p)$ with $E \vdash e{::}\mathsf{RefT}\ T$ refers to a method that is statically available for the reference e, the dynamic lookup of the object pointed at by e should yield a method with a compatible result type. The lemma that helps to establish this behavior reads as follows:

$$\begin{array}{l}\mathsf{wf\_prog}\ \Gamma \wedge \mathsf{mhead}\ \Gamma\ T\ sig = \mathsf{Some}\ (m_1,rT_1) \wedge\\ \Gamma \vdash \mathsf{Class}\ T_1 \preceq \mathsf{RefT}\ T \longrightarrow\\ \quad \exists m_2\ rT_2\ b.\ \ \mathsf{cmethd}\ (\Gamma,T_1)\ sig = \mathsf{Some}\ (m_2,rT_2,b) \wedge\\ \qquad\qquad\qquad \Gamma \vdash rT_2 \preceq rT_1\end{array}$$

The proofs of these lemmas are lengthy and require many auxiliary theorems that are proved by induction on the direct subclass relation, by case splitting on the right-hand argument of the widening relation and by rule induction on the subinterface, subclass, and implementation relation.

5.3 Type soundness

Finally, we state and prove the type soundness theorem.

5.3.1 Notions

In order to express the type soundness theorem, we introduce the notion of a state σ conforming to an environment E, written $\sigma {::}{\preceq} E$, which intuitively means that the value of any variable within the state is compatible with its static type. The conformance relation is based on the two auxiliary concepts $\Gamma,h \vdash v{::}{\preceq}\ T$ of a value v conforming to a type T and $\Gamma,h \vdash obj{::}{\preceq} \diamond$ of all components of an object obj conforming to their respective types, both with reference to a given program Γ and heap h:

$$\begin{array}{l}\Gamma,h \vdash v{::}{\preceq}\ T \stackrel{\mathsf{def}}{=} \exists T'.\ \mathsf{typeof}\ (\mathsf{option\_map}\ \mathsf{obj\_ty} \circ h)\ v =\\ \qquad\qquad\qquad\qquad \mathsf{Some}\ T' \wedge G \vdash T' \preceq T\end{array}$$

$$\begin{array}{l}\Gamma,h \vdash obj{::}{\preceq} \diamond \stackrel{\mathsf{def}}{=} \mathsf{case}\ obj\ \mathsf{of}\\ \quad \mathsf{Obj}\ C\ fs \Rightarrow \forall T\ f.\ \mathsf{table}\ (\mathsf{fields}\ (\Gamma,C))\ f = \mathsf{Some}\ T \longrightarrow\\ \qquad\qquad\qquad \exists v.\ fs\ f = \mathsf{Some}\ v\ \wedge\ \ \Gamma,h \vdash v{::}{\preceq}\ T\\ \mid \mathsf{Arr}\ T\ cs \Rightarrow \forall i\ v.\ cs\ i = \mathsf{Some}\ v \longrightarrow \Gamma,h \vdash v{::}{\preceq}\ T\end{array}$$

$$\begin{array}{l}s{::}{\preceq} E \stackrel{\mathsf{def}}{=} \mathsf{let}\ \Gamma = \mathsf{prg}\ E;\ h = \mathsf{heap}\ s;\ t = \mathsf{this}\ s\ \mathsf{in}\\ \qquad (\forall a\ obj.\ h\ a = \mathsf{Some}\ obj \longrightarrow \Gamma,h \vdash obj{::}{\preceq} \diamond) \wedge\\ \qquad \Gamma,h \vdash \mathsf{local}\ s{::}{\preceq}\mathsf{localT}\ E \wedge\\ \qquad \Gamma,h \vdash \mathsf{Addr}\ t{::}{\preceq}\mathsf{Class}\ (\mathsf{thisT}\ E)\end{array}$$

Another helpful notion used below is a pre-order on heaps: $h \trianglelefteq h'$ means that any object existing on heap h also exists on h' and has the same type there. This property holds for any transition of the operational semantics, which turns out to be necessary in our proof of type soundness.

$$\begin{array}{l}h \trianglelefteq h' \stackrel{\mathsf{def}}{=} \forall a.\ (\forall C\ fs.\ h\ \ a = \mathsf{Some}\ (\mathsf{Obj}\ C\ fs\ ) \longrightarrow\\ \qquad\qquad\qquad \exists fs'.\ h'\ \ a = \mathsf{Some}\ (\mathsf{Obj}\ C\ fs')) \wedge\\ \qquad\qquad (\forall T\ cs.\ h\ \ a = \mathsf{Some}\ (\mathsf{Arr}\ T\ cs\ ) \longrightarrow\\ \qquad\qquad\qquad \exists cs'.\ h'\ \ a = \mathsf{Some}\ (\mathsf{Arr}\ T\ cs'))\end{array}$$

5.3.2 Main theorem

Next, we give the key type soundness theorem. It is proved by simultaneous rule induction on the evaluation of expressions and statements and therefore has to be formulated in a way that gives a strong enough induction hypothesis. We do not attempt to cast it into words. Instead, we discuss some of its corollaries below, which are surprisingly clear.

$$\begin{array}{l}\mathsf{wf\_prog}\ \Gamma \longrightarrow\\ (\Gamma \vdash (x,(h\ ,l\ ,t))\ -s\rightarrow\ \ (x',(h',l',t')) \longrightarrow\\ \ \forall lT\ tT.\ (h\ ,l\ ,t\ ){::}{\preceq}(\Gamma,lT,tT) \longrightarrow\\ \qquad\qquad\qquad\qquad (\Gamma,lT,tT) \vdash s{::}\diamond \longrightarrow\\ \qquad\qquad (h',l',t'){::}{\preceq}(\Gamma,lT,tT) \wedge h \trianglelefteq h') \wedge\\ (\Gamma \vdash (x,(h\ ,l\ ,t))\ -e{\rhd}v\rightarrow\ (x',(h',l',t')) \longrightarrow\\ \ \forall lT\ tT.\ (h\ ,l\ ,t\ ){::}{\preceq}(\Gamma,lT,tT) \longrightarrow\\ \ \forall T.\qquad\qquad\qquad (\Gamma,lT,tT) \vdash e{::}\ T \longrightarrow\\ \qquad\qquad (h',l',t'){::}{\preceq}(\Gamma,lT,tT) \wedge h \trianglelefteq h' \wedge\\ \qquad\qquad (x' = \mathsf{None} \longrightarrow \Gamma,h' \vdash v{::}{\preceq}\ T))\end{array}$$

The proof of this theorem is by far the heaviest. At its top level, it consists of (currently) 19 cases, one per syntactic construct, where

- 7 cases can be solved rather directly (e.g. from the induction hypothesis),
- 4 cases require just simple lemmas on the structure of the state,
- and the remaining 8 cases require extensive reasoning on the characteristic properties of the constructs concerned.

Most of this reasoning is independent of the operational semantics itself and can be factored out, which keeps the main proof manageable.

5.3.3 Corollaries

For a discussion of its consequences, we state two immediate corollaries of the main theorem. In the context of a well-formed program, the execution of a well-typed statement transforms a state conforming to the environment into another state that again conforms to the environment:

$$\begin{array}{l}\Gamma = \mathsf{fst}\ E \wedge \mathsf{wf\_prog}\ \Gamma \wedge \\ \Gamma \vdash (x,\sigma)\ -s\rightarrow\ (x',\sigma') \wedge \sigma :: \preceq E \wedge E \vdash s :: \diamond \longrightarrow \\ \quad \sigma' :: \preceq E\end{array}$$

The same holds for the evaluation of well-typed expression, where additionally we have that, unless an exception occurs during evaluation, the resulting value conforms to the static type of the expression:

$$\begin{array}{l}\Gamma = \mathsf{fst}\ E \wedge \mathsf{wf\_prog}\ \Gamma \wedge \\ \Gamma \vdash (x,\sigma)\ -e\rhd v\rightarrow\ (x',\sigma') \wedge \sigma :: \preceq E \wedge E \vdash e :: T \longrightarrow \\ \quad \sigma' :: \preceq E \wedge (x' = \mathsf{None} \longrightarrow \Gamma,\mathsf{fst}\ \sigma' \vdash v :: \preceq T)\end{array}$$

This is what type soundness actually means.

A corollary of type soundness is that method calls always find a suitable method, i.e. a 'method not understood' runtime error is impossible. This can be stated more formally: for a well-formed program and a state that conforms to the environment, if an expression of reference type (which plays the role of the target expression for the method call considered) evaluates without an exception to a non-null reference, and if there is a method available for that (static) type and a given signature, the dynamic method lookup for the same signature according to the class instance pointed at by the reference value yields a method body:

$$\begin{array}{l}\Gamma = \mathsf{fst}\ E \wedge \mathsf{wf\_prog}\ \Gamma \wedge \\ \Gamma \vdash (x,\sigma)\ -e\rhd a'\rightarrow\ \mathsf{Norm}\ \sigma' \wedge a' \neq \mathsf{Val\ Null} \wedge \\ dynT = \mathsf{fst}\ (\mathsf{the\_Obj}\ (\mathsf{fst}\ \sigma'\ (\mathsf{the\_Addr}\ a'))) \wedge \\ \sigma :: \preceq E \wedge E \vdash e :: \mathsf{RefT}\ T \wedge \mathsf{mhead}\ \Gamma\ T\ sig \neq \mathsf{None} \longrightarrow \\ \quad \mathsf{cmethd}\ (\Gamma, dynT)\ sig \neq \mathsf{None}\end{array}$$

This implies that in a well-formed context, in every instance of the evaluation rule for method calls, the function cmethd returns a proper method body.

6 Experience and statistics

Because of the expressiveness of HOL, our formalization of BALI is quite natural and direct. Isabelle's mixfix syntax and mathematical font are indispensable for writing moderately readable definitions and theorems. The theory files add up to about 1100 lines of well-documented specifications. It took us roughly two months of work and about 2400 lines of proof scripts to show the type soundness theorem with all necessary lemmas.

Although we are far from satisfied with the current status of Isabelle's proof procedures (for example, the handling of assumptions during simplification, or the necessity to expand tuples and similar datatypes by hand), they are basically adequate for the task at hand. Nevertheless, more automation is necessary and feasible.

The adaption of old proofs after changing the formalization is a tedious job. Although the changes in the proofs are usually quite local, there tend to be many. Higher-level proof scripts and more automation are some of the answers. A dedicated mechanism for exploring and fixing the impact of modifications would also help.

7 Conclusion

The reader has been exposed to large chunks of a formal language specification and a proof of type soundness and may need to be reminded of the benefits. Even including the slight generalizations mentioned in §4, we did not discover a loop-hole in the type system. But we had not seriously expected this either. So what have we gained over and above a level of certainty far beyond any paper-and-pencil proof?

We view our work primarily as an investment for the future. For a start, it can serve as the basis for many other mechanized proofs about Java, e.g. as a foundation for the work by Dean [8] or for compiler correctness. More importantly, we see machine-checked proofs as an invaluable aid in maintaining large language designs (or formal documents of any kind). It is all very well to perform a detailed proof on paper once, but in the face of changes of the formalization, the reliability of such proofs begins to crumble. In contrast, we developed the design incrementally, and Isabelle reminded us where proofs needed to be modified. Unless the language changes drastically, such modifications of proofs tend to be of a local nature. This change management will continue to be of great importance when we extend BALI further: apart from adding the last important Java features missing from BALI, full exception handling and threads, we also plan to use BALI as a vehicle for experimental extensions of Java such as parameterized classes [15, 17, 1].

Despite our general enthusiasm for machine-checked language designs, a few words of warning are in order:

- BALI is still a half-way house: not a toy language any more, but missing many details and some important features of Java.
- The type system of BALI is, despite subclassing, simpler than that of your average functional language: whereas BALI's type checking rules are almost directly executable, the verification of ML's type inference algorithm against the type system requires a significant effort [16]. The key complication there is the presence of free and bound type variables, which requires complex reasoning about substitutions. VanInwegen [22] reports similar difficulties in her formalization of the type system and the semantics of ML.
- Theorem provers, and Isabelle is no exception, require a certain learning effort due to the machine-oriented proof style. Recent moves towards a more human-oriented proof style like Syme's DECLARE system [20] promise to lower this barrier. However, as Harrison [13] points out, both proof styles have their merits, and we are currently investigating a combination of both.

In a nutshell: although machine-checked language designs for the masses are still some way off, this paper demonstrates that they have definitely become a viable option for the expert.

Acknowledgments. We thank Sophia Drossopoulou and Donald Syme for the very helpful discussions about their work. We also thank Wolfgang Naraschewski, Markus Wenzel, Andrew Gordon, and two anonymous referees for their comments on draft versions of this paper.

References

[1] O. Agesen, S. N. Freund, and J. C. Mitchell. Adding type parameterization to the Java language. In *ACM Symp. Object-Oriented Programming: Systems, Languages and Applications*, 1997.

[2] R. S. Boyer and J. S. Moore. *A Computational Logic Handbook.* Academic Press, 1988.

[3] K. B. Bruce. Safe type checking in a statically-typed object-oriented programming language. In *Proc. 20th ACM Symp. Principles of Programming Languages*, pages 285–298. ACM Press, 1993.

[4] K. B. Bruce, J. Crabtree, T. P. Murtagh, R. van Gent, A. Dimock, and R. Muller. Safe and decidable type checking in an object-oriented language. In *Proc. OOPSLA'93*, volume 18 of *ACM SIGPLAN Notices*, pages 29–46, Oct. 1993.

[5] K. B. Bruce, R. van Gent, and A. Schuett. PolyTOIL: A type-safe polymorphic object-oriented language. In W. Olthoff, editor, *ECOOP '95*, volume 952 of *Lect. Notes in Comp. Sci.*, pages 27–51. Springer-Verlag, 1995.

[6] R. M. Cohen. The defensive Java Virtual Machine specification. Technical report, Computational Logic Inc., 1997. Draft version.

[7] W. Cook. A proposal for making Eiffel type-safe. In *Proc. ECOOP'89*, pages 57–70. Cambridge University Press, 1989.

[8] D. Dean. The security of static typing with dynamic linking. In *Proc. 4th ACM Conf. Computer and Communications Security.* ACM Press, 1997.

[9] S. Drossopoulou and S. Eisenbach. Is the Java type system sound? In *Proc. 4th Int. Workshop Foundations of Object-Oriented Languages*, Jan. 1997.

[10] S. Drossopoulou and S. Eisenbach. Java is type safe — probably. In *ECOOP'97 — Object-Oriented Programming*, volume 1241 of *Lect. Notes in Comp. Sci.*, pages 389–418. Springer-Verlag, 1997.

[11] M. Gordon and T. Melham. *Introduction to HOL: a theorem-proving environment for higher order logic.* Cambridge University Press, 1993.

[12] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison-Wesley, 1996.

[13] J. Harrison. Proof style. Technical Report 410, University of Cambridge Computer Laboratory, 1997.

[14] R. Milner. A theory of type polymorphism in programming. *J. Comp. Sys. Sci.*, 17:348–375, 1978.

[15] A. C. Myers, J. A. Bank, and B. Liskov. Parameterized types for Java. In *Proc. 24th ACM Symp. Principles of Programming Languages*, pages 132–145, 1997.

[16] W. Naraschewski and T. Nipkow. Type inference verified: Algorithm W in Isabelle/HOL. In C. Paulin-Mohring, editor, *Proc. Int. Workshop TYPES'96*, volume 1??? of *Lect. Notes in Comp. Sci.* Springer-Verlag, 1997. To appear.

[17] M. Odersky and P. Wadler. Pizza into Java: Translating theory into practice. In *Proc. 24th ACM Symp. Principles of Programming Languages*, pages 146–159, 1997.

[18] L. C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 1994.

[19] K. Slind. Function definition in higher order logic. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics*, volume 1125 of *Lect. Notes in Comp. Sci.*, pages 381–397. Springer-Verlag, 1996.

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

[21] D. Syme. Proving Java type soundness. Technical Report 427, University of Cambridge Computer Laboratory, 1997.

[22] M. VanInwegen. Towards type preservation for core SML. University of Cambridge Computer Laboratory, 1997.