This is a preprint version of chapter four of

```
@incollection{Oheimb-Nipkow-Java-LNCS,
author     = {Oheimb, David von and Nipkow, Tobias},
title      = {Machine-checking the {J}ava Specification:
              Proving Type-Safety},
booktitle  = {Formal Syntax and Semantics of {J}ava},
editor     = {Jim Alves-Foss},
url        = {http://isabelle.in.tum.de/Bali/doc/Springer98.html},
publisher  = {Springer},
series     = {LNCS},
volume     = {1523},
pages      = {119-156}
year       = {1999}
}
```

Machine-checking the Java Specification: Proving Type-Safety*

David von Oheimb** and Tobias Nipkow

Fakultät für Informatik, Technische Universität München
http://www.in.tum.de/~oheimb
http://www.in.tum.de/~nipkow

Abstract. In this article we present Bali, the formalization of a large (hitherto sequential) sublanguage of Java. We give its abstract syntax, type system, well-formedness conditions, and an operational evaluation semantics. Based on these definitions, we can express soundness of the type system, an important design goal claimed to be reached by the designers of Java, and prove that Bali is indeed type-safe.
All definitions and proofs have been done formally in the theorem prover Isabelle/HOL. Thus this article demonstrates that machine-checking the design of non-trivial programming languages has become a reality.

1 Introduction

Bali is a large subset of Java [GJS96]. This article presents its formalization and the proof of a key property, namely the soundness of its type system — specified and verified in the theorem prover Isabelle/HOL [Pau94].

On the face of it, this article 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 and its implications. Although these technicalities do indeed take up much of the space, there is a meta-theme running through the article, which we consider even more 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.

* This is a completely revised and extended version of [NO98].
** Research supported by DFG SPP *Deduktion*.

Note that the type-safety of Java is not sufficient to guarantee secure execution of bytecode programs on the Java Virtual Machine, because the bytecode might be tampered with, produced by a faulty compiler, or not be related to any Java source program at all. This was the main reason for introducing the Bytecode Verifier in the JVM, which checks the integrity, in particular type-correctness, of any bytecode before execution. Of course similar security problems arise for any other high-level languages as well. Nevertheless, the investigation of type-safety at source level is worthwhile: it checks whether the language design is sound, which means that at least all the necessary conditions expressible at that level are fulfilled. In particular static typing loses much of its *raison d'être* if the language is not type-safe.

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' [Mil78] in the context of ML. Milner uses a denotational semantics, in contrast to most of the later work, including ours. The question of type-safety came to prominence with the discovery of its failure in Eiffel [Coo89]. Ever since, many designers of programming languages (especially OO ones) have been at pains to prove type-safety of their languages (see, for example, the series of papers by Bruce et al. [Bru93,BCM+93,BvGS95]).

Directly related to our work is that by Drossopoulou and Eisenbach [DE98] who prove (on paper) type-safety of a subset of Java very similar to Bali. Although we were familiar with an earlier version [DE97] 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 [Sym98] formalized the paper [DE97] as far as possible, uncovering two significant mistakes, both in connection with 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 [Dea97] 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-safety of dynamically linked programs. Cohen [Coh97] 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 [BM88]. No proofs have been reported yet.

2 Overview

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:

- class and interface declarations with instance fields and methods,
- subinterface, subclass, and implementation relations with inheritance, overriding, and hiding,
- method calls with static overloading and dynamic binding,
- some primitive types, objects (including arrays),
- exception throwing and handling.

This portion of Java is very similar to that covered by [DE98] and [Sym98].

We do not consider Java packages and separate compilation. For the moment, we also leave out several features of Java like class variables and static methods, constructors and finalizers, the visibility of names, and concurrency, but we aim to include at least part of them in later stages of our project. Some constructs are simplified without limiting the expressiveness of the language (see §4.1).

In developing the formalization of BALI and investigating its properties, we aim to meet the following design goals:

- faithfulness to the official language specification,
- succinctness and simplicity,
- maintainability and extendibility,
- adequacy for the theorem prover.

It might be interesting to keep these goals in mind while reading §4 on the formalization of BALI and §5 on our proofs and check how far we have reached them. We comment on our experience in pursuing these goals in §6.

3 The basics of Isabelle/HOL

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

Isabelle/HOL is the instantiation of the generic interactive theorem prover Isabelle [Pau94] with Church's version of Higher-Order Logic and is very close to Gordon's HOL system [GM93]. In this article HOL is short for Isabelle/HOL.

The appearance of formulas is standard, e.g. '$\longrightarrow$' is the (right-associative) infix implication symbol. Predicates are functions with Boolean result. Function application is written in curried style. For descriptions we apply Hilbert's choice operator ε, where $\varepsilon x.\ P\ x$ denotes some value x satisfying P, or an arbitrary value if no such x exists.

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 '$\Rightarrow$'. Type abbreviations are introduced simply as equations. A free datatype is defined by listing its constructors together with their argument types, separated by '|'.

There are the basic types *bool* and *int*, as well as the polymorphic type $(\alpha)set$ of homogeneous sets for any type α. Occasionally we apply the infix 'image' operator lifting a function over a set, defined as $f``S \stackrel{\mathsf{def}}{=} \{y.\ \exists x{\in}S.\ y = f\,x\}$.

The product type $\alpha \times \beta$ comes with the projection functions fst and snd. Tuples are pairs nested to the right, e.g. $(a,b,c) = (a,(b,c))$.

As the list type $(\alpha)list$ is defined via its constructors [] denoting the empty list and the infix 'cons' operator '#', it can be introduced by the datatype declaration

$$(\alpha)list = [] \mid \alpha\#(\alpha)list$$

The concatenation operator on lists is written as the infix symbol '@'. There is a functional map :: $(\alpha \Rightarrow \beta) \Rightarrow (\alpha)list \Rightarrow (\beta)list$ applying a function to all elements of a list, as well as a conversion function set :: $(\alpha)list \Rightarrow (\alpha)set$.

We frequently use the datatype

$$(\alpha)option = \mathsf{None} \mid \mathsf{Some}\ \alpha$$

It has an unpacking function the :: $(\alpha)option \Rightarrow \alpha$ such that the (Some x) $= x$ and the None = arbitrary, where arbitrary is an unknown value defined as $\varepsilon x.\,\mathsf{False}$. There is a simple function mapping o2s :: $(\alpha)option \Rightarrow (\alpha)set$ converting an optional value to a set, with the characteristic equations o2s (Some x) $= \{x\}$ and o2s None $= \{\}$.

Most of the HOL text shown in this article 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.

We adopt the following typographic conventions: Java keywords like `catch` appear in typewriter font, the names of logical constants like cfield appear in sans serif, while type names like *state* and meta-variables like v appear in italics.

4 The formalization of Bali

This section presents all aspects of our formalization of Bali[1].

As far as Bali is a subset of Java, it strictly adheres to the Java language specification [GJS96], with several 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.
- if a class or an interface inherits more than one method with the same signature, the methods need not have identical return types.
- no check of result types in dynamic method lookup.
- the type of an assignment is determined by the right-hand side, which can be more specific than the left-hand side.

[1] The Isabelle sources are available from the Bali project page `http://www.in.tum.de/~isabelle/bali/`

We found several issues concerning exceptions not specified in [GJS96] and therefore define a reasonable behavior that seems to be consistent with current implementations:

- given a Null reference, the `throw` statement raises a `NullPointer` exception.
- each system exception thrown yields a fresh exception object.
- if there is not enough memory even to allocate an `OutOfMemory` error, program execution simply halts. (Our experiments showed erratic behavior of some implementation in this case, ranging from sudden termination without executing `finally` blocks, over hangup, to repeated invocation of a single exception handler!)

To illustrate our approach, we use the following (artificial) example.

```
class Base {
  boolean vee;
  Base foo(Base x) {
    return x;
  }
}

class Ext extends Base{
  int vee;
  Ext foo(Base x) {
    ((Ext)x).vee=1;
    return null;
  }
}

Base e;
e=new Ext();
try {e.foo(null); }
catch (NullPointerException x) {throw x;}
```

This program fragment consists of two simple but complete class declarations and a block of statements that might occur in any method that has access to these declarations. It contains the following features of Bali:

- class declarations with inheritance, hiding of fields, and overriding of methods (with refined result type),
- return expressions, parameter access,
- sequential composition, expression statements, field assignment, type cast, local accesses, literal values, exception propagation,
- local assignment, instance creation,
- try & catch statement, method call (with dynamic binding), throw statement

4.1 Abstract syntax

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

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

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

Throughout the article, the symbol 'Γ' denotes a BALI program, as we use programs as part of the static type context defined in §4.2.

Each declaration is a pair of a name and the defined entity. Some names, like those of predefined classes (including those of system exceptions *xname*), have a predefined meaning and are therefore handled extra. We do not specify the structure of names further, but use the opaque HOL types *tname0*, *mname*, and *ename0* for user-defined type names, method names, and "expression names" (i.e. field and variable identifiers).

$xname =$ `Throwable`
$\quad \mid$ `NullPointer` $\mid$ `OutOfMemory` $\mid$ `ClassCast`
$\quad \mid$ `NegArrSize` $\mid$ `IndOutBound` $\mid$ `ArrStore`

$tname =$ `Object`	name of the top of the class hierarchy
$\mid$ SXcpt *xname*	name of a system exception
$\mid$ TName *tname0*	other class or interface name

$ename =$ `this`	special name for `this` pointer
$\mid$ EName *ename0*	expression name

An interface (*iface*) contains lists of superinterface names and method declarations. A *class* specifies the names of an optional superclass and implemented interfaces, and lists of field and method declarations.

$iface = (tname)list \times (sig \times mhead)list$
$idecl = tname \times iface$
$class = (tname)option \times (tname)list \times (fdecl)list \times (mdecl)list$
$cdecl = tname \times class$

A field declaration (*fdecl*) simply gives the field type (*ty*, see §4.2). A method declaration ($sig \times mhead$ for interfaces or *mdecl* for classes) consists of a "signature" [GJS96, 8.4.2] (i.e. the method name and the list of parameter types, excluding the result type) followed by *mhead*, consisting of the list of parameter names and the result type, and (if it appears within a class) the method body (*mbody*). The latter consists of the list of local variables, a statement *stmt* as body, and a return expression *expr* (see below). As in [DE98], 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, up to now each method has exactly one parameter; multiple parameters can be simulated by a single parameter object with multiple fields.

field	$= ty$	field type
fdecl	$= ename \times field$	
sig	$= mname \times ty$	method name and parameter type
mhead	$= ename \times ty$	parameter name and result type
lvar	$= ename \times ty$	local variable name and type
mbody	$= (lvar)list \times stmt \times expr$	local vars, block, and return expression
methd	$= mhead \times mbody$	method (of a class)
mdecl	$= sig \times methd$	

In the abstract syntax given above, the formalization of our example program fragment looks like this:

```
BaseC ≝ (Base, (Some Object,
                []),
                [(vee, PrimT boolean)],
                [((foo,Class Base),(x,Class Base), ([],Skip,x))]))
ExtC  ≝ (Ext,  (Some Base,
                []),
                [(vee, PrimT int)],
                [((foo,Class Base),(x,Class Ext), ([],
                   Expr({ClassT Ext}(Class Ext)x.vee:=Lit (Intg 1)),
                   Lit Null))]))
classes ≝ [ObjectC,
           SXcptC Throwable,
           SXcptC NullPointer, SXcptC OutOfMemory, SXcptC ClassCast,
           SXcptC NegArrSize, SXcptC IndOutBound, SXcptC ArrStore,
           BaseC, ExtC]
tprg ≝ ([],classes)
test ≝ Expr(e:=new Ext);
       try Expr(e.foo({Class Base}Lit Null))
       catch((SXcpt NullPointer) x) (throw x)
```

where *Base* stands for TName *Base_*, *Ext* for TName *Ext_*, and similarly for *vee*, *x*, and *e*. The constants *Base_*, *Ext_*, etc. are all distinct. The sequence of statements *test* could have been embedded in *tprg*, which we have left out for simplicity.

Representation of lookup tables The representation of declarations as lists gives an implicit finiteness constraint, which turns out to be necessary for the well-foundedness of the subclass and subinterface relation. The list representation also enables an explicit check whether the declared entities are named uniquely, implemented with the function unique given below. This function does not check for duplicate definitions, which is harmless.

$\mathsf{unique} :: (\alpha \times \beta)\mathit{list} \Rightarrow \mathit{bool}$
$\mathsf{unique}\ t \stackrel{\mathsf{def}}{=} \forall (x_1,y_1) \in \mathsf{set}\ t.\ \forall (x_2,y_2) \in \mathsf{set}\ t.\ x_1 = x_2 \longrightarrow y_1 = y_2$

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 defined easily:

$\mathsf{empty} :: (\alpha,\beta)\mathit{table}$
$\_[\_\mapsto\_] :: (\alpha,\beta)\mathit{table} \Rightarrow \alpha \Rightarrow \beta \Rightarrow (\alpha,\beta)\mathit{table}$
$\_ \oplus \_ :: (\alpha,\beta)\mathit{table} \Rightarrow (\alpha,\beta)\mathit{table} \Rightarrow (\alpha,\beta)\mathit{table}$
$\mathsf{table\_of} :: (\alpha\times\beta)\mathit{list} \Rightarrow (\alpha,\beta)\mathit{table}$
$\_\ \mathsf{hiding}\ \_$
$\quad \mathsf{entails}\ \_ :: (\alpha,\beta)\mathit{table} \Rightarrow (\alpha,\gamma)\mathit{table} \Rightarrow (\beta \Rightarrow \gamma \Rightarrow \mathit{bool}) \Rightarrow \mathit{bool}$

$\mathsf{empty} \stackrel{\mathsf{def}}{=} \lambda k.\ \mathsf{None}$
$t[x\mapsto y] \stackrel{\mathsf{def}}{=} \lambda k.\ \mathsf{if}\ k = x\ \mathsf{then}\ \mathsf{Some}\ y\ \mathsf{else}\ t\ k$
$s \oplus t \stackrel{\mathsf{def}}{=} \lambda k.\ \mathsf{case}\ t\ k\ \mathsf{of}\ \mathsf{None} \Rightarrow s\ k \mid \mathsf{Some}\ x \Rightarrow \mathsf{Some}\ x$

$\mathsf{table\_of}\ [] = \mathsf{empty}$
$\mathsf{table\_of}\ ((k,x)\#t) = (\mathsf{table\_of}\ t)[k\mapsto x]$

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

For the union of tables, we also need the type of non-unique tables,

$(\alpha,\beta)\mathit{tables} = \alpha \Rightarrow (\beta)\mathit{set}$

together with a union operator and straightforward variants of two of the notions defined above:

$\_ \oplus\oplus \_ :: (\alpha,\beta)\mathit{tables} \Rightarrow (\alpha,\beta)\mathit{tables} \Rightarrow (\alpha,\beta)\mathit{tables}$
$\mathsf{Un\_tables} :: ((\alpha,\beta)\mathit{tables})\mathit{set} \Rightarrow (\alpha,\beta)\mathit{tables}$
$\_\ \mathsf{hidings}\ \_$
$\quad \mathsf{entails}\ \_ :: (\alpha,\beta)\mathit{tables} \Rightarrow (\alpha,\gamma)\mathit{tables} \Rightarrow (\beta \Rightarrow \gamma \Rightarrow \mathit{bool}) \Rightarrow \mathit{bool}$

$$\mathsf{Un\_tables}\ ts \stackrel{\mathsf{def}}{=} \lambda k.\ \bigcup t{\in}ts.\ t\ k$$
$$s \oplus\oplus t \stackrel{\mathsf{def}}{=} \lambda k.\ \mathsf{if}\ t\ k = \{\}\ \mathsf{then}\ s\ k\ \mathsf{else}\ t\ k$$
$$t\ \mathsf{hidings}\ s\ \mathsf{entails}\ R \stackrel{\mathsf{def}}{=} \forall k.\ \forall x{\in}t\ k.\ \forall y{\in}s\ k.\ R\ x\ y$$

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

$$\mathsf{iface} :: prog \Rightarrow (tname,\ iface)table \stackrel{\mathsf{def}}{=} \mathsf{table\_of} \circ \mathsf{fst}$$
$$\mathsf{class} :: prog \Rightarrow (tname,\ class)table \stackrel{\mathsf{def}}{=} \mathsf{table\_of} \circ \mathsf{snd}$$

More interesting are the following functions that traverse the type hierarchy of a program, collecting the methods and fields into a table (the types *tname* and *ref_ty* are defined in §4.2):

$$\begin{array}{lll}\mathsf{imethds} & :: prog \Rightarrow tname \Rightarrow (sig, & ref\_ty\ \times mhead)tables\\ \mathsf{cmethd} & :: prog \Rightarrow tname \Rightarrow (sig, & ref\_ty\ \times methd)table\\ \mathsf{fields} & :: prog \Rightarrow tname \Rightarrow ((ename \times ref\_ty) \times field\ )list\end{array}$$

Note that imethds collects a non-unique table of method declarations allowing for inheritance of more than one method with the same signature.

As Syme [Sym98] points out, a naive recursive definition of these functions is not possible in HOL because the class hierarchy might be cyclic, which is ruled out for well-formed programs (see §4.3) 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 [Sli96]. We do not give their definitions, but only the recursion equations, which we derive as easy consequences:

$$\begin{array}{l}\mathsf{wf\_prog}\ \Gamma \wedge \mathsf{iface}\ \Gamma\ I = \mathsf{Some}\ (is,ms) \longrightarrow\\ \mathsf{imethds}\ \Gamma\ I = \mathsf{Un\_tables}\ ((\lambda J.\ \mathsf{imethds}\ \Gamma\ J)\ \text{“}\ \mathsf{set}\ is) \oplus\oplus\\ \qquad\qquad (\mathsf{o2s} \circ \mathsf{table\_of}\ (\mathsf{map}\ (\lambda(s,mh).\ (s,\mathsf{IfaceT}\ I,mh))\ ms))\end{array}$$

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

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

The structure of the three equations is the same: the tables are constructed recursively from the corresponding tables of the superinterfaces or the superclass (if any), which models inheritance, augmented — with overriding — by the newly declared items. All declared items receive an extra label, namely their defining interface or class.

In our example, we obtain

$$
\begin{aligned}
&\mathsf{fields}\ \mathit{tprg}\ \mathit{Base} = [((\mathit{vee},\ \mathsf{ClassT}\ \mathit{Base}),\ \mathsf{PrimT}\ \texttt{boolean})]\\
&\mathsf{fields}\ \mathit{tprg}\ \mathit{Ext}\ \ = [((\mathit{vee},\ \mathsf{ClassT}\ \mathit{Ext}\ ),\ \mathsf{PrimT}\ \texttt{int}),\\
&\qquad\qquad\qquad ((\mathit{vee},\ \mathsf{ClassT}\ \mathit{Base}),\ \mathsf{PrimT}\ \texttt{boolean})]\\
&\mathsf{cmethd}\ \mathit{tprg}\ \mathit{Base} = \mathsf{empty}[(\mathit{foo},\ \mathsf{Class}\ \mathit{Base}) \mapsto\\
&\qquad\qquad\qquad (\mathsf{ClassT}\ \mathit{Base},\ (x,\ \mathsf{Class}\ \mathit{Base}),\ ([],\ \mathsf{Skip},\ x))]\\
&\mathsf{cmethd}\ \mathit{tprg}\ \mathit{Ext}\ \ = \mathsf{empty}[(\mathit{foo},\ \mathsf{Class}\ \mathit{Base}) \mapsto\\
&\qquad\qquad\qquad (\mathsf{ClassT}\ \mathit{Ext}\ \ ,\ (x,\ \mathsf{Class}\ \mathit{Ext}\ ),\ ([],\\
&\qquad\qquad\qquad\ \ \mathsf{Expr}(\{\mathsf{ClassT}\ \mathit{Ext}\}(\mathsf{Class}\ \mathit{Ext})x.\mathit{vee}{:=}\mathsf{Lit}\ (\mathsf{Intg}\ 1)),\\
&\qquad\qquad\qquad\ \ \mathsf{Lit}\ \mathsf{Null}))]
\end{aligned}
$$

Terms We define statements (appearing in method bodies), expressions (appearing in statements), and 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.

For a more modular description, we divide the `try _ catch _ finally _` statement into a `try _ catch _` statement and a `_ finally _` statement, which might be used in any context. Our version of the `try _ catch _` statement has exactly one `catch` clause. Multiple `catch` clauses can be simulated with cascaded `if _ else _` statements applying the `_ instanceof _` operator.

$$
\begin{aligned}
\mathit{stmt} = {}& \mathsf{Skip}\\
&\mid \mathsf{Expr}\ \mathit{expr}\\
&\mid \mathit{stmt};\ \mathit{stmt}\\
&\mid \texttt{if}\quad (\mathit{expr})\ \mathit{stmt}\ \texttt{else}\ \mathit{stmt}\\
&\mid \texttt{while}(\mathit{expr})\ \mathit{stmt}\\
&\mid \texttt{throw}(\mathit{expr})\\
&\mid \texttt{try}\ \mathit{stmt}\ \texttt{catch}(\mathit{tname}\ \mathit{ename})\ \mathit{stmt}\\
&\mid \mathit{stmt}\ \texttt{finally}\ \mathit{stmt}
\end{aligned}
$$

Skip denotes the empty statement. The "expression statement" Expr is a conversion from expressions to statements causing evaluation for side effects only. Assignments and method calls, which are expressions because they yield a value, can be turned into statements this way. In contrast to Java, for simplicity we allow this conversion to be applied to any kind of expression.

Concerning expressions, our formalization leaves out the standard unary and binary operators as their typing and semantics is straightforward. The `this` expression is modeled as a special non-assignable local variable named `this`. The `super` construct can be simulated with a `this` expression that is cast to the superclass of the current class. Creation of multi-dimensional arrays can be simulated with nested array creation, while access and assignment to multi-dimensional arrays is nested anyway.

It might be reasonable to introduce the general notion of variables (i.e. left-hand sides of assignments) in order to factor out common behavior of local variables, class instance variables, and array components. But we have chosen not to do so because the semantic treatment of these three variants of variables differs considerably. This decision leads to some redundancy between access and assignment, especially in the semantics for arrays.

$expr =$ `new` $tname$	class instance creation
$\mid$ `new` $ty[expr]$	array creation
$\mid (ty)\,expr$	type cast
$\mid expr$ `instanceof` ref_ty	type comparison operator
$\mid$ Lit val	literal
$\mid ename$	local/parameter access
$\mid ename := expr$	local/parameter assignment
$\mid \{ref\_ty\}expr.\,ename$	field access
$\mid \{ref\_ty\}expr.\,ename := expr$	field assignment
$\mid expr[expr]$	array access
$\mid expr[expr] := expr$	array assignment
$\mid expr.\,mname(\{ty\}expr)$	method call

The terms in braces $\{\ldots\}$ above are called *type annotations*. Strictly speaking, they are not part of the input language but serve as auxiliary information (computed by the type checker) that is crucial for the static binding of fields and the resolution of method overloading. Distinguishing between the actual input language and the augmented language would lead to a considerable amount of redundancy. We avoid this by assuming that the annotations are added beforehand by a kind of preprocessor. The correctness of the annotations is checked by the typing rules (see §4.2).

The definition of values is straightforward. It relies on the standard HOL types of Boolean values (*bool*) and integers (*int*), whereas the type *loc* of locations, i.e. abstract non-null addresses of objects, is not further specified.

$val =$ Unit	dummy result of `void` methods
$\mid$ Bool $bool$	
$\mid$ Intg int	
$\mid$ Null	
$\mid$ Addr loc	

The definitions below give some simple destructor functions for *val* with their characteristic properties.

$$\mathsf{the\_Bool} :: val \Rightarrow bool$$
$$\mathsf{the\_Intg} :: val \Rightarrow int$$
$$\mathsf{the\_Addr} :: val \Rightarrow loc$$

$$\mathsf{the\_Bool}\ (\mathsf{Bool}\ b) = b$$
$$\mathsf{the\_Intg}\ (\mathsf{Intg}\ i) = i$$
$$\mathsf{the\_Addr}\ (\mathsf{Addr}\ a) = a$$

4.2 Type system

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

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

```
prim_ty = void
        | boolean
        | int

ref_ty  = NullT
        | IfaceT tname
        | ClassT tname
        | ArrayT ty

ty      = PrimT prim_ty
        | RefT  ref_ty
```

`void` is used as a dummy type for methods without result. In the sequel NT stands for RefT NullT, Iface I for RefT(IfaceT I), Class C for RefT(ClassT C), and T[] for RefT(ArrayT T).

An interface or class type is considered as a proper type only if there is a corresponding declaration for its type name in the current program, which is checked by the following predicates:

```
is_iface :: prog ⇒ tname ⇒ bool
is_class :: prog ⇒ tname ⇒ bool
is_type  :: prog ⇒ ty    ⇒ bool

is_iface Γ tn ≡ iface Γ tn ≠ None
is_class Γ tn ≡ class Γ tn ≠ None
is_type  Γ (PrimT _)  = True
is_type  Γ  NT        = True
is_type  Γ (Iface  I) = is_iface Γ I
is_type  Γ (Class  C) = is_class Γ C
is_type  Γ (T[])      = is_type  Γ T
```

For all types, a default value is defined via

```
default_val :: ty ⇒ val
default_val (PrimT void   ) = Unit
default_val (PrimT boolean) = Bool False
default_val (PrimT int    ) = Intg 0
default_val (RefT  r      ) = Null
```

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 ($\_ \vdash \_ \rightsquigarrow^1 \_$) relations are of type $\mathit{prog} \times \mathit{tname} \times \mathit{tname} \Rightarrow \mathit{bool}$ and are defined as follows:

$$\begin{array}{lcl}
\Gamma \vdash I \prec_{\mathsf{i}}^1 J & \stackrel{\mathsf{def}}{=} & \mathsf{is_iface}\ \Gamma\ I \wedge \mathsf{is_iface}\ \Gamma\ J \wedge \quad J \in \mathsf{set}\ (\mathsf{fst}(\mathsf{the}(\mathsf{iface}\ \Gamma\ I))) \\
\Gamma \vdash C \prec_{\mathsf{c}}^1 D & \stackrel{\mathsf{def}}{=} & \mathsf{is_class}\ \Gamma\ C \wedge \mathsf{is_class}\ \Gamma\ D \wedge \quad \mathsf{Some}\ D = \mathsf{fst}(\mathsf{the}(\mathsf{class}\ \Gamma\ C)) \\
\Gamma \vdash C \rightsquigarrow^1 I & \stackrel{\mathsf{def}}{=} & \mathsf{is_class}\ \Gamma\ C \wedge \mathsf{is_iface}\ \Gamma\ I \wedge I \in \mathsf{set}\ (\mathsf{fst}(\mathsf{snd}(\mathsf{the}(\mathsf{class}\ \Gamma\ C))))
\end{array}$$

The transitive (but not reflexive) closures $\_ \vdash \_ \prec_{\mathsf{i}} \_$ and $\_ \vdash \_ \prec_{\mathsf{c}} \_$ can be defined inductively:

$$\frac{\Gamma \vdash I \prec_{\mathsf{i}}^1 K}{\Gamma \vdash I \prec_{\mathsf{i}} K} \qquad \frac{\Gamma \vdash I \prec_{\mathsf{i}} J;\ \ \Gamma \vdash J \prec_{\mathsf{i}} K}{\Gamma \vdash I \prec_{\mathsf{i}} K}$$

$$\frac{\Gamma \vdash C \prec_{\mathsf{c}}^1 E}{\Gamma \vdash C \prec_{\mathsf{c}} E} \qquad \frac{\Gamma \vdash C \prec_{\mathsf{c}} D;\ \ \Gamma \vdash D \prec_{\mathsf{c}} E}{\Gamma \vdash C \prec_{\mathsf{c}} E}$$

There is also a kind of transitive closure of $\_ \vdash \_ \rightsquigarrow^1 \_$ defined as

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

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 given below. Note that some rules carry the additional premise that `Object` is a proper class, which will be ensured for well-formed programs.

$$\frac{\mathsf{is\_type}\ \Gamma\ T}{\Gamma \vdash T \preceq T} \qquad \frac{\mathsf{is\_type}\ \Gamma\ (\mathsf{RefT}\ R)}{\Gamma \vdash \mathsf{NT} \preceq \mathsf{RefT}\ R}$$

$$\frac{\Gamma \vdash I \prec_{\mathsf{i}} J}{\Gamma \vdash \mathsf{Iface}\ I \preceq \mathsf{Iface}\ J} \qquad \frac{\mathsf{is\_iface}\ \Gamma\ I;\ \ \mathsf{is\_class}\ \Gamma\ \mathtt{Object}}{\Gamma \vdash \mathsf{Iface}\ I \preceq \mathsf{Class}\ \mathtt{Object}}$$

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

$$\frac{\Gamma \vdash \mathsf{RefT}\ S \preceq \mathsf{RefT}\ T}{\Gamma \vdash (\mathsf{RefT}\ S)[] \preceq (\mathsf{RefT}\ T)[]} \qquad \frac{\mathsf{is\_type}\ \Gamma\ T[];\ \ \mathsf{is\_class}\ \Gamma\ \mathtt{Object}}{\Gamma \vdash T[] \preceq \mathsf{Class}\ \mathtt{Object}}$$

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

$$\frac{\Gamma \vdash S \preceq T}{\Gamma \vdash S \preceq_? T} \qquad \frac{\Gamma \vdash C \prec_{\mathsf{c}} D}{\Gamma \vdash \mathsf{Class}\ D \preceq_? \mathsf{Class}\ C} \qquad \frac{\mathsf{is\_class}\ \Gamma\ C;\ \ \mathsf{is\_iface}\ \Gamma\ I}{\Gamma \vdash \mathsf{Class}\ C \preceq_? \mathsf{Iface}\ I}$$

$$\frac{\Gamma \vdash \mathsf{RefT}\ S \preceq_? \mathsf{RefT}\ T}{\Gamma \vdash (\mathsf{RefT}\ S)[] \preceq_? (\mathsf{RefT}\ T)[]} \qquad \frac{\mathsf{is\_class}\ \Gamma\ \mathtt{Object};\ \ \mathsf{is\_type}\ \Gamma\ T}{\Gamma \vdash \mathsf{Class}\ \mathtt{Object} \preceq_? T[]}$$

$$\frac{\begin{array}{c}\mathsf{is\_iface}\ \Gamma\ J;\ \neg\Gamma\vdash I\prec_! J;\\ \mathsf{imethds}\ \Gamma\ I\ \mathsf{hidings}\ \mathsf{imethds}\ \Gamma\ J\ \mathsf{entails}\\ (\lambda(m_1,(pn_1,rT_1))\ (m_2,(pn_2,rT_2)).\ \Gamma\vdash rT_1\preceq rT_2)\end{array}}{\Gamma\vdash \mathsf{Iface}\ I\preceq_?\mathsf{Iface}\ J}\qquad \frac{\mathsf{is\_iface}\ \Gamma\ I;\ \ \mathsf{is\_class}\ \Gamma\ C}{\Gamma\vdash \mathsf{Iface}\ I\preceq_?\mathsf{Class}\ C}$$

Typing rules Now we come to type-checking itself, which is expressed as a set of constraints on the types of expressions, relative to a type environment.

An *environment* consists of a global part, namely a program Γ, and a local part (written 'Λ'), namely the types of the local variables including the current class, i.e. the type of `this`:

$$\begin{array}{l} lenv = (ename,\ ty)\ table\\ env = prog \times lenv\end{array}$$

$$\begin{array}{l}\mathsf{prg}::\ env\Rightarrow prog \stackrel{\mathsf{def}}{=} \lambda(\Gamma,\Lambda).\ \Gamma\\ \mathsf{lcl}\ ::\ env\Rightarrow lenv \stackrel{\mathsf{def}}{=} \lambda(\Gamma,\Lambda).\ \Lambda\end{array}$$

The well-typedness of statements and the typing of expressions are defined inductively relative to an environment. The typing of expressions is unique, as can be shown easily by rule induction.

$$\begin{array}{l}\_\vdash\_::\diamond\quad ::\ env\Rightarrow stmt\qquad\Rightarrow bool\\ \_\vdash\_::\_\quad ::\ env\Rightarrow expr\Rightarrow ty\Rightarrow bool\end{array}$$

The type-checking rules for most statements are standard:

$$\frac{}{E\vdash \mathsf{Skip}::\diamond}\qquad \frac{E\vdash e::T}{E\vdash \mathsf{Expr}\ e::\diamond}\qquad \frac{E\vdash c_1::\diamond;\ \ E\vdash c_2::\diamond}{E\vdash c_1;\ c_2::\diamond}$$

$$\frac{E\vdash e::\mathsf{PrimT}\ \texttt{boolean};\ \ E\vdash c_1::\diamond;\ \ E\vdash c_2::\diamond}{E\vdash \texttt{if}(e)\ c_1\ \texttt{else}\ c_2::\diamond}$$

$$\frac{E\vdash e::\mathsf{PrimT}\ \texttt{boolean};\ \ E\vdash c::\diamond}{E\vdash \texttt{while}(e)\ c::\diamond}\qquad \frac{E\vdash c_1::\diamond;\ \ E\vdash c_2::\diamond}{E\vdash c_1\ \texttt{finally}\ c_2}$$

Note the use of the widening relation in the following two rules to ensure that a value thrown or caught as an exception is indeed a exception object.

$$\frac{E\vdash e::\mathsf{Class}\ tn;\ \ \mathsf{prg}\ E\vdash \mathsf{Class}\ tn\preceq\mathsf{Class}\ (\mathsf{SXcpt}\ \texttt{Throwable})}{E\vdash \texttt{throw}\ e::\diamond}$$

$$\frac{\begin{array}{c}(\Gamma,\Lambda)\vdash c_1::\diamond;\ \ \Gamma\vdash \mathsf{Class}\ tn\preceq\mathsf{Class}\ (\mathsf{SXcpt}\ \texttt{Throwable});\\ \Lambda\ vn=\mathsf{None};\ \ (\Gamma,\Lambda[vn\mapsto\mathsf{Class}\ tn])\vdash c_2::\diamond\end{array}}{(\Gamma,\Lambda)\vdash \texttt{try}\ c_1\ \texttt{catch}(tn\ vn)\ c_2::\diamond}$$

The `try _ catch _` statement is the only one that involves a change of the type environment, namely to include typing information for the exception parameter. The name of this parameter is required to be new in the local environment.

The typing rules for the first few of the expressions are straightforward, except for the confusing direction of the casting relation in the type cast rule:

$$\frac{\mathsf{is\_class}\ (\mathsf{prg}\ E)\ C}{E \vdash \mathtt{new}\ C :: \mathsf{Class}\ C} \qquad \frac{\mathsf{is\_type}\ (\mathsf{prg}\ E)\ T;\ \ E \vdash i :: \mathsf{PrimT}\ \mathtt{int}}{E \vdash \mathtt{new}\ T[i] :: T[]}$$

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

$$\frac{E \vdash e :: \mathsf{RefT}\ T;\ \ \mathsf{prg}\ E \vdash \mathsf{RefT}\ T \preceq_? \mathsf{RefT}\ T'}{E \vdash e\ \mathtt{instanceof}\ T' :: \mathsf{PrimT}\ \mathtt{boolean}}$$

The rule for Lit prohibits addresses as literal values, which is implemented by supplying $\lambda a.$ None as the "dynamic type" argument in the call of the function

```
typeof :: (loc ⇒ ty option) ⇒ val ⇒ ty option
typeof dt  Unit     = Some (PrimT void)
typeof dt (Bool b)  = Some (PrimT boolean)
typeof dt (Intg i)  = Some (PrimT int)
typeof dt  Null     = Some (RefT  NullT)
typeof dt (Addr a)  = dt a
```

This function is reused below with a more interesting value for the parameter dt, namely a function to compute the dynamic type of a reference.

The typings of all three assignment variants are quite similar, except that for local variables additionally an assignment to `this` is forbidden. In any case, as a generalization to the Java specification, the type of the assignment is determined by the right-hand (as opposed to the left-hand) side.

$$\frac{\mathsf{lcl}\ E\ vn = \mathsf{Some}\ T;\ \mathsf{is\_type}\ (\mathsf{prg}\ E)\ T}{E \vdash vn :: T}$$

$$\frac{E \vdash vn :: T;\ \ E \vdash v :: T';\ \ \mathsf{prg}\ E \vdash T' \preceq T;\ \ vn \neq \mathtt{this}}{E \vdash vn := v :: T'}$$

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

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

$$\frac{E \vdash a :: T[];\ \ E \vdash i :: \mathsf{PrimT}\ \mathtt{int}}{E \vdash a[i] :: T}$$

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

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

The function $\mathsf{cfield} :: \mathit{prog} \Rightarrow \mathit{tname} \Rightarrow (\mathit{ename}\ ,\ \mathit{ref\_ty} \times \mathit{field})\mathit{table}$, defined as $\mathsf{cfield}\ \Gamma\ C \stackrel{\mathsf{def}}{=} \mathsf{table\_of}\ ((\mathsf{map}\ (\lambda((n,d),t).\ (n,(d,t))))\ (\mathsf{fields}\ \Gamma\ C))$, is a variant of fields. It implements a field lookup that is based on the field name alone in contrast to a combination of field name and defining class. Thus 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.

The type annotations $\{\ldots\}$ in the above rules for field access and method call are used to implement static binding for fields and to resolve overloaded method names statically. Technically speaking, the typing rules serve as constraints on these annotations during type-checking, but one can also think of the annotations being filled with schematic variables that are instantiated with their correct values in the type-checking process, as is demonstrated in the example overleaf. The value of each annotation is uniquely determined by the value of a function in the premise of the field access and method call rule:

A field access $\{fd\}e.fn$ is annotated with the defining class of the field found when searching the class hierarchy for the name fn (using cfield), starting from the static type $\mathsf{Class}\ C$ of e. The annotation $\{fd\}$ will be used at run-time to access the field 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" [GJS96, 15.11.2] methods for reference type T (as defined below) yields exactly one method entry. In this case, the method 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. Thus any static overloading of the method name mn has been resolved and the dynamic method lookup at run-time will be based on the signature (mn,pT').

$$
\begin{array}{lll}
\mathsf{max\_spec} & :: \mathit{prog} \Rightarrow \mathit{ref\_ty} \Rightarrow \mathit{sig} & \Rightarrow((\mathit{ref\_ty} \times \mathit{mhead}) \times \mathit{ty})\ \ \mathit{set} \\
\mathsf{appl\_methds} & :: \mathit{prog} \Rightarrow \mathit{ref\_ty} \Rightarrow \mathit{sig} & \Rightarrow((\mathit{ref\_ty} \times \mathit{mhead}) \times \mathit{ty})\ \ \mathit{set} \\
\mathsf{mheads} & :: \mathit{prog} \Rightarrow \mathit{ref\_ty} \Rightarrow \mathit{sig} & \Rightarrow (\mathit{ref\_ty} \times \mathit{mhead})\ \ \mathit{set} \\
\mathsf{more\_spec} & :: \mathit{prog} \Rightarrow (\mathit{ref\_ty} \times \mathit{mhead}) \times \mathit{ty} \Rightarrow (\mathit{ref\_ty} \times \mathit{mhead}) \times \mathit{ty} \Rightarrow \mathit{bool}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{max\_spec}\ \Gamma\ T\ \mathit{sig} \stackrel{\mathsf{def}}{=} \{m \mid m \in \mathsf{appl\_methds}\ \Gamma\ T\ \mathit{sig} \wedge \\
\qquad\qquad (\forall m' \in \mathsf{appl\_methds}\ \Gamma\ T\ \mathit{sig}. \\
\qquad\qquad\qquad \mathsf{more\_spec}\ \Gamma\ m'\ m \longrightarrow m' = m)\} \\
\mathsf{appl\_methds}\ \Gamma\ T\ (mn,\ pT) \stackrel{\mathsf{def}}{=} \{(m,pT') \mid m \in \mathsf{mheads}\ \Gamma\ T\ (mn,\ pT') \wedge \\
\qquad\qquad\qquad\qquad \Gamma \vdash pT \preceq pT'\} \\
\mathsf{mheads}\ \Gamma\ \ \mathsf{NullT}\ \ = \lambda \mathit{sig}.\ \{\} \\
\mathsf{mheads}\ \Gamma\ (\mathsf{IfaceT}\ I) = \mathsf{imethds}\ \Gamma\ I \\
\mathsf{mheads}\ \Gamma\ (\mathsf{ClassT}\ C) = \mathsf{o2s} \circ \mathsf{option\_map}\ (\lambda(d,(h,b)).(d,h)) \circ \mathsf{cmethd}\ \Gamma\ C \\
\mathsf{mheads}\ \Gamma\ (\mathsf{ArrayT}\ T) = \lambda \mathit{sig}.\ \{\} \\
\mathsf{more\_spec}\ \Gamma\ ((md,mh),pT)\ ((md',mh'),pT') \stackrel{\mathsf{def}}{=} \Gamma \vdash \mathsf{RefT}\ md \preceq \mathsf{RefT}\ md' \wedge \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \Gamma \vdash pT \preceq pT' \\
\textit{where} \\
\quad \mathsf{option\_map} :: (\alpha \Rightarrow \beta) \Rightarrow (\alpha\ \mathit{option} \Rightarrow \beta\ \mathit{option}) \\
\quad \mathsf{option\_map}\ f \stackrel{\mathsf{def}}{=} \lambda y.\ \mathsf{case}\ y\ \mathsf{of}\ \mathsf{None} \Rightarrow \mathsf{None} \mid \mathsf{Some}\ x \Rightarrow \mathsf{Some}\ (f\ x)
\end{array}
$$

The well-typedness of our example code *test* is derived as given below. For formatting reasons, the derivation tree is cut at several positions, whereby the positions are marked with the labels of the cut subtrees. Irrelevant values in formulas are replaced by _. We use the following abbreviations:

Γ = *tprg*
Λ = empty[e↦Class *Base*]
SNP = SXcpt `NullPointer`

During the derivation, the schematic variable $?pT'$ is instantiated with Class *Base*, as a result of the function max_spec.

$$\dfrac{\dfrac{\Lambda\ e = \mathsf{Some}\ (\mathsf{Class}\ \mathit{Base}) \quad \mathsf{is\_type}\ \Gamma\ (\mathsf{Class}\ \mathit{Base})}{(\Gamma,\Lambda) \vdash e :: \mathsf{Class}\ \mathit{Base}} \quad e \neq \texttt{this} \quad \dfrac{\mathsf{is\_class}\ \Gamma\ \mathit{Ext}}{(\Gamma,\Lambda) \vdash \texttt{new}\ \mathit{Ext} :: \mathsf{Class}\ \mathit{Ext}} \quad \Gamma \vdash \mathsf{Class}\ \mathit{Ext} \preceq \mathsf{Class}\ \mathit{Base}}{\dfrac{(\Gamma,\Lambda) \vdash \quad (e\texttt{:=new}\ \mathit{Ext}) :: \_}{(\Gamma,\Lambda) \vdash \mathsf{Expr}(e\texttt{:=new}\ \mathit{Ext}) :: \Diamond}\ (\mathit{LAss})}$$

$$\dfrac{\dfrac{\Lambda\ e = \mathsf{Some}\ (\mathsf{Class}\ \mathit{Base}) \quad \mathsf{is\_type}\ \Gamma\ (\mathsf{Class}\ \mathit{Base})}{(\Gamma,\Lambda) \vdash e :: \mathsf{Class}\ \mathit{Base}} \quad \dfrac{\mathsf{typeof}\ (\lambda a.\ \mathsf{None})\ \mathsf{Null} = \mathsf{Some}\ \mathsf{NT}}{(\Gamma,\Lambda) \vdash \mathsf{Lit}\ \mathsf{Null} :: \mathsf{NT}} \quad \mathsf{max\_spec}\ \Gamma\ (\mathsf{ClassT}\ \mathit{Base})\ (\mathit{foo}, \mathsf{NT}) = \{((\_, \_, \mathsf{Class}\ \mathit{Base}), ?pT')\}}{\dfrac{(\Gamma,\Lambda) \vdash \quad (e.\mathit{foo}(\{?pT'\}\mathsf{Lit}\ \mathsf{Null})) :: \mathsf{Class}\ \mathit{Base}}{(\Gamma,\Lambda) \vdash \mathsf{Expr}(e.\mathit{foo}(\{?pT'\}\mathsf{Lit}\ \mathsf{Null})) :: \Diamond}\ (\mathit{Call})}$$

$$\dfrac{\dfrac{\Lambda[x\mapsto\mathsf{Class}\ \mathit{SNP}]\ x = \mathsf{Some}\ (\mathsf{Class}\ \mathit{SNP}) \quad \mathsf{is\_type}\ \Gamma\ (\mathsf{Class}\ \mathit{SNP})}{(\Gamma,\ \Lambda[x\mapsto\mathsf{Class}\ \mathit{SNP}]) \vdash x :: \mathsf{Class}\ \mathit{SNP}} \quad \Gamma \vdash \mathsf{Class}\ \mathit{SNP} \preceq \mathsf{Class}\ (\mathsf{SXcpt}\ \texttt{Throwable})}{(\Gamma,\ \Lambda[x\mapsto\mathsf{Class}\ \mathit{SNP}]) \vdash \texttt{throw}\ x :: \Diamond}\ (\mathit{Throw})$$

$$\dfrac{(\mathit{LAss}) \qquad \dfrac{(\mathit{Call}) \quad \Gamma \vdash \mathsf{Class}\ \mathit{SNP} \preceq \mathsf{Class}\ (\mathsf{SXcpt}\ \texttt{Throwable}) \quad \Lambda\ x = \mathsf{None} \quad (\mathit{Throw})}{(\Gamma,\Lambda) \vdash \texttt{try}\ \mathsf{Expr}(e.\mathit{foo}(\{?pT'\}\mathsf{Lit}\ \mathsf{Null}))\ \texttt{catch}(\mathit{SNP}\ x)\ (\texttt{throw}\ x) :: \Diamond}}{(\Gamma,\Lambda) \vdash \mathsf{Expr}(e\texttt{:=new}\ \mathit{Ext})\texttt{;}\ \texttt{try}\ \mathsf{Expr}(e.\mathit{foo}(\{?pT'\}\mathsf{Lit}\ \mathsf{Null}))\ \texttt{catch}(\mathit{SNP}\ x)\ (\texttt{throw}\ x) :: \Diamond}$$

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, class, and whole program declarations.

$$
\begin{array}{lllll}
\mathsf{wf\_fdecl} & :: prog \Rightarrow & & fdecl \Rightarrow bool \\
\mathsf{wf\_mhead} & :: prog \Rightarrow sig & \times mhead \Rightarrow bool \\
\mathsf{wf\_mdecl} & :: prog \Rightarrow tname \Rightarrow & mdecl \Rightarrow bool \\
\mathsf{wf\_idecl} & :: prog \Rightarrow & & idecl \Rightarrow bool \\
\mathsf{wf\_cdecl} & :: prog \Rightarrow & & cdecl \Rightarrow bool \\
\mathsf{wf\_prog} & :: prog \Rightarrow & & & bool
\end{array}
$$

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

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

A method declaration is well-formed only if its argument and result types are defined and the name of the parameter is not `this`. Additionally, if the declaration appears in a class, the names of the local variables must be unique and may not contain the special name `this` nor hide the parameter, all types of the local variables must exist, the method body has to be well-typed (in the static context of its parameter type and the current class), and its result expression must have a type that widens to the result type:

$$
\begin{array}{l}
\mathsf{wf\_mhead}\ \Gamma \quad ((mn,pT),(pn,rT)) \stackrel{\mathsf{def}}{=} \mathsf{is\_type}\ \Gamma\ pT \wedge \mathsf{is\_type}\ \Gamma\ rT \wedge pn \neq \texttt{this} \\
\mathsf{wf\_mdecl}\ \Gamma\ C\ ((mn,pT),(pn,rT),lvars,blk,res) \stackrel{\mathsf{def}}{=} \\
\quad \mathsf{let}\ ltab = \mathsf{table\_of}\ lvars;\ E = (\Gamma,ltab[\texttt{this}\mapsto\mathsf{Class}\ C][pn\mapsto pT]) \\
\quad \mathsf{in}\ \ \mathsf{wf\_mhead}\ \Gamma\ ((mn,pT),(pn,rT)) \wedge \\
\qquad \mathsf{unique}\ lvars \wedge ltab\ \texttt{this} = \mathsf{None} \wedge ltab\ pn = \mathsf{None} \wedge \\
\qquad (\forall (vn,T) \in \mathsf{set}\ lvars.\ \mathsf{is\_type}\ \Gamma\ T) \wedge \\
\qquad E \vdash blk :: \diamond \wedge \exists T.\ E \vdash res :: T \wedge \Gamma \vdash T \preceq rT
\end{array}
$$

Even 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, any method overriding a set of methods defined in some superinterfaces has a result type that widens to all their result types:

$$
\begin{array}{l}
\mathsf{wf\_idecl}\ \Gamma\ (I,(is,ms)) \stackrel{\mathsf{def}}{=} \neg\ \mathsf{is\_class}\ \Gamma\ I \wedge \\
\quad (\forall J \in \mathsf{set}\ is.\ \mathsf{is\_iface}\ \Gamma\ J \wedge \neg\ \Gamma \vdash J \prec_{\mathsf{i}} I) \wedge \\
\quad \mathsf{unique}\ ms \wedge (\forall m \in \mathsf{set}\ ms.\ \mathsf{wf\_mhead}\ \Gamma\ m \wedge \\
\quad \mathsf{let}\ mtab = \mathsf{Un\_tables}\ ((\lambda J.\ \mathsf{imethds}\ \Gamma\ J)\text{``}\mathsf{set}\ is)\ \mathsf{in} \\
\quad (\mathsf{o2s} \circ \mathsf{table\_of}\ ms)\ \mathsf{hidings}\ mtab\ \mathsf{entails} \\
\qquad (\lambda (pn,rT)\ (m,(pn',rT')).\ \Gamma \vdash rT \preceq rT')
\end{array}
$$

Similarly, the name of a well-formed class declaration is not an interface name. All implemented interfaces exist, and for any method of such an 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 a subclass of the current class. Furthermore, any method overriding a method of the superclass has a compatible result type:

$$
\begin{array}{l}
\mathsf{wf\_cdecl}\ \Gamma\ (C,(sc,si,fs,ms)) \stackrel{\mathsf{def}}{=} \neg\ \mathsf{is\_iface}\ \Gamma\ C \wedge \\
\quad (\forall I{\in}\mathsf{set}\ si.\ \mathsf{is\_iface}\ \Gamma\ I \wedge \\
\qquad \forall s.\ \forall(m_1,(pn_1,rT_1)) \in \mathsf{imethds}\ \Gamma\ I\ s. \\
\qquad\quad \exists m_2\ \ pn_2\ rT_2\ b.\ \ \mathsf{cmethd}\ \ \Gamma\ C\ s = \mathsf{Some}\ (m_2,(pn_2,rT_2),b) \wedge \\
\qquad\qquad\qquad\qquad \Gamma \vdash rT_2 \preceq rT_1) \wedge \\
\quad \mathsf{unique}\ fs\ \wedge (\forall f\ \in\mathsf{set}\ fs.\ \ \mathsf{wf\_fdecl}\ \ \Gamma\ \ \ f\ ) \wedge \\
\quad \mathsf{unique}\ ms \wedge (\forall m{\in}\mathsf{set}\ ms.\ \mathsf{wf\_mdecl}\ \Gamma\ C\ m) \wedge \\
\quad (\mathsf{case}\ sc\ \mathsf{of}\ \mathsf{None}\ \ \ \Rightarrow C = \texttt{Object} \\
\qquad\qquad \mid \mathsf{Some}\ D \Rightarrow \mathsf{is\_class}\ \Gamma\ D \wedge \neg\ \Gamma \vdash D \prec_{\mathsf{c}} C \wedge \\
\qquad\qquad\qquad\qquad \mathsf{table\_of}\ ms\ \mathsf{hiding}\ \mathsf{cmethd}\ \Gamma\ D\ \mathsf{entails} \\
\qquad\qquad\qquad\qquad\quad (\lambda((pn_1,rT_1),b)\ (m,((pn_2,rT_2),b')).\ \Gamma \vdash rT_1 \preceq rT_2)
\end{array}
$$

Finally, all interfaces and classes declared in a well-formed program are named uniquely and are in turn well-formed. For uniformity, this includes the predefined class declarations of `Object` and the (flat) hierarchy of system exceptions.

$$
\begin{array}{l}
\mathsf{ObjectC}\ \ \stackrel{\mathsf{def}}{=}\ (\texttt{Object}\ \ ,\ (\mathsf{None}\ \ \ ,\ [],\ [],\ [])) \\
\mathsf{SXcptC}\ xn \stackrel{\mathsf{def}}{=}\ \mathsf{let}\ sc = \mathsf{if}\ xn{=}\texttt{Throwable}\ \mathsf{then}\ \texttt{Object}\ \mathsf{else}\ \mathsf{SXcpt}\ \texttt{Throwable}\ \mathsf{in} \\
\qquad\qquad (\mathsf{SXcpt}\ xn,\ (\mathsf{Some}\ sc,\ [],\ [],\ []))
\end{array}
$$

$$
\begin{array}{l}
\mathsf{wf\_prog}\ \Gamma \stackrel{\mathsf{def}}{=}\ \mathsf{let}\ is = \mathsf{set}\ (\mathsf{fst}\ \Gamma);\ cs = \mathsf{set}\ (\mathsf{snd}\ \Gamma) \\
\qquad\qquad \mathsf{in}\ \mathsf{ObjectC} \in cs \wedge \forall xn.\ \mathsf{SXcptC}\ xn \in cs \wedge \\
\qquad\qquad\quad \mathsf{unique}\ (\mathsf{fst}\ \ \Gamma) \wedge \forall i{\in}is.\ \ \mathsf{wf\_idecl}\ \ \Gamma\ i) \wedge \\
\qquad\qquad\quad \mathsf{unique}\ (\mathsf{snd}\ \Gamma) \wedge \forall c{\in}cs.\ \mathsf{wf\_cdecl}\ \Gamma\ c)
\end{array}
$$

Our example program *tprg* is well-formed. Here is a heavily abstracted derivation tree of our proof of this fact.

$$
\frac{\begin{array}{l}\mathsf{wf\_mdecl}\ tprg\ Base\ ((foo,\ \mathsf{Class}\ Base),\\ \qquad (x,\ \mathsf{Class}\ Base),\ [],\ \mathsf{Skip},\ x)\end{array} \qquad \neg(tprg \vdash \texttt{Object} \prec_{\mathsf{c}} Base)}{\mathsf{wf\_cdecl}\ tprg\ BaseC}
$$

$$
\frac{\begin{array}{l}\mathsf{wf\_mdecl}\ tprg\ Ext\ ((foo,\ \mathsf{Class}\ Base),\\ (x,\ \mathsf{Class}\ Ext),\ [],\ \mathsf{Expr}\ (\{\mathsf{ClassT}\ Ext\}(\mathsf{Class}\ Ext)\\ \qquad x.vee{:=}\mathsf{Lit}\ (\mathsf{Intg}\ 1)),\ \mathsf{Lit}\ \mathsf{Null})\end{array} \qquad \neg(tprg \vdash Base \prec_{\mathsf{c}} Ext)}{\mathsf{wf\_cdecl}\ tprg\ BaseC}
$$

$$
\frac{\mathsf{wf\_cdecl}\ tprg\ BaseC \quad \mathsf{wf\_cdecl}\ tprg\ ExtC \quad Base \neq Ext}{\mathsf{wf\_}tprg\ tprg}
$$

4.4 Operational semantics

We formalize the semantics of Java in operational style with evaluation rules. This is the natural choice since the language specification itself is given in an operational evaluation-oriented style, which allows for a direct formalization and its straightforward validation. Furthermore, a denotational semantics would require much more difficult mathematical tools, and an axiomatic semantics would be problematic to validate and to use for reasoning on the language as a whole. We prefer an evaluation semantics to a transition semantics in order to obtain a concise description, because we consider a transition semantics less readable and rather low-level, which in particular holds for a formulation as an Abstract State Machine (ASM) like in [BS98].

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

State A *state* consists of an optional exception (of type *xcpt*), a heap, and a current invocation frame, which is the values of the local variables (including method and exception parameters and the `this` pointer):

$state = (xcpt)option \times st$
$st \quad = heap \times locals$
$\mathsf{heap} :: st \Rightarrow heap \quad \stackrel{\mathsf{def}}{=} \lambda(h,l).\ h$
$\mathsf{locals} :: st \Rightarrow locals \quad \stackrel{\mathsf{def}}{=} \lambda(h,l).\ l$

Remember that tuples associative to the right, so if for some state σ we have an equation like $\sigma = (x, \sigma')$, then x is the (optional) exception component alone, while the second projection σ' of the state has (tuple) type *st*, i.e. represents a "small" state excluding the exception entry.

An *exception* is a reference to an instance of some exception class, which is a subclass of `Throwable`. Normally, when an exception is thrown, a fresh exception object is allocated and its location returned to represent the exception. But in the case of system exceptions, we defer their allocation (and just record their names) until an enclosing `catch` block references it. This helps to avoid the subtleties of (conditional) side effects on the heap and out-of-memory conditions. Thus we model exceptions as follows.

$xcpt = \mathsf{XcptLoc}\ loc$
$\quad\ \mid \mathsf{SysXcpt}\ xname$

A *heap* maps locations to objects, while local variables map names to values:

$heap \ = (loc \quad , obj)table$
$locals = (ename, val)table$

In our model there is no need to explicitly maintain a stack of invocation frames containing local variables and return addresses for method calls. In this way we also abstract over the finiteness of stack space. On the other hand, we explicitly model the possibility of memory allocation on the heap to fail if there is no free location (i.e. some a with $(\mathsf{heap}\ \sigma)\ a = \mathsf{None}$) available. Memory allocation is loosely, yet deterministically, defined by the function

$$\begin{aligned}
&\mathsf{new_Addr} :: \mathit{heap} \Rightarrow (\mathit{loc} \times (\mathit{xcpt})\mathit{option})\mathit{option}\\
&\mathsf{new_Addr}\ h \stackrel{\mathsf{def}}{=} \varepsilon y.\ (\quad y = \mathsf{None} \wedge\ (\forall a.\ h\ a \neq \mathsf{None})) \vee\\
&\qquad\qquad (\exists a\ x.\ y = \mathsf{Some}\ (a,x) \wedge h\ a = \mathsf{None} \wedge\\
&\qquad\qquad\quad (x = \mathsf{None} \vee x = \mathsf{Some}\ (\mathsf{SysXcpt}\ \texttt{OutOfMemory})))
\end{aligned}$$

This function fails, i.e. returns None, iff there is no free location on the heap, and otherwise gives an unused location. At the latest when there is only one free address left, it returns an `OutOfMemory` exception. In this way it is guaranteed that when an `OutOfMemory` exception is thrown for the first time, there is a free location on the heap to allocate it. Note that we do not consider garbage collection.

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:

$$\begin{aligned}
\mathit{fields} &= (\mathit{ename} \times \mathit{ref_ty},\ \mathit{val})\mathit{table}\\
\mathit{components} &= (\mathit{int}\qquad\quad ,\ \mathit{val})\mathit{table}\\
\mathit{obj} &= \mathsf{Obj}\ \mathit{tname}\ \mathit{fields}\\
&\ \ |\ \mathsf{Arr}\ \mathit{ty}\ \mathit{components}
\end{aligned}$$

$$\begin{aligned}
&\mathsf{the_Obj} :: (\mathit{obj})\mathit{option} \Rightarrow \mathit{tname} \times \mathit{fields}\\
&\mathsf{the_Arr} :: (\mathit{obj})\mathit{option} \Rightarrow \mathit{ty} \times \mathit{components}\\
&\mathsf{obj_ty} :: \mathit{obj} \Rightarrow \mathit{ty}
\end{aligned}$$

$$\begin{aligned}
&\mathsf{the_Obj}\ (\mathsf{Some}\ (\mathsf{Obj}\ C\ fs)) = (C,fs)\\
&\mathsf{the_Arr}\ (\mathsf{Some}\ (\mathsf{Arr}\ T\ cs)) = (T,cs)\\
&\mathsf{obj_ty}\ (\mathsf{Obj}\ C\ fs) = \mathsf{Class}\ C\\
&\mathsf{obj_ty}\ (\mathsf{Arr}\ T\ cs) = T[]
\end{aligned}$$

Using obj_ty we define the predicate $\Gamma,\sigma \vdash v$ fits T, meaning that in the context of Γ and state σ, the value v is assignable to a variable of type T. This proposition, which is computed at run-time for type casts and array assignments, is a weaker version of the notion of conformance introduced in §5.3.

$$\begin{aligned}
&_\ ,_ \vdash _\ \mathsf{fits}\ _ :: \mathit{prog} \Rightarrow \mathit{st} \Rightarrow \mathit{val} \Rightarrow \mathit{ty} \Rightarrow \mathit{bool}\\
&\Gamma,\sigma \vdash v\ \mathsf{fits}\ T \stackrel{\mathsf{def}}{=} (\exists \mathit{pt}.\ T = \mathsf{PrimT}\ \mathit{pt}) \vee v = \mathsf{Null} \vee\\
&\qquad\qquad \Gamma \vdash \mathsf{obj_ty}\ (\mathsf{the}\ (\mathsf{heap}\ \sigma\ (\mathsf{the_Addr}\ v))) \preceq T
\end{aligned}$$

There is a number of auxiliary functions for constructing and updating the state, namely:

$$\begin{aligned}
&\mathsf{lupd}[_\mapsto_]\ _ :: \mathit{ename} \Rightarrow \mathit{val} \Rightarrow \mathit{st} \Rightarrow \mathit{st}\\
&\mathsf{hupd}[_\mapsto_]\ _ :: \mathit{loc} \Rightarrow \mathit{obj} \Rightarrow \mathit{st} \Rightarrow \mathit{st}\\
&\mathsf{x_case} :: \mathit{xcpt}\ \mathit{option} \Rightarrow \mathit{st} \Rightarrow \mathit{st} \Rightarrow \mathit{state}\\
&\mathsf{lupd}[v\mapsto x\ ]\ (h,l) \stackrel{\mathsf{def}}{=} (h,l[v\mapsto x])\\
&\mathsf{hupd}[a\mapsto \mathit{obj}]\ (h,l) \stackrel{\mathsf{def}}{=} (h[a\mapsto \mathit{obj}],l)\\
&\mathsf{x_case}\quad x\ \sigma'\ \sigma \stackrel{\mathsf{def}}{=} (x, \mathsf{if}\ x = \mathsf{None}\ \sigma'\ \mathsf{else}\ \sigma)
\end{aligned}$$

$$
\begin{array}{lll}
\mathsf{init\_vars} & :: & (\alpha \times ty)\,list \Rightarrow (\alpha,val)\,table \\
\mathsf{init\_Obj} & :: & prog \Rightarrow tname \Rightarrow obj \\
\mathsf{init\_Arr} & :: & ty \Rightarrow int \Rightarrow obj \\
\mathsf{init\_vars} & \stackrel{\mathsf{def}}{=} & \mathsf{table\_of} \circ \mathsf{map}\ (\lambda(n,T).\ (n,\mathsf{default\_val}\ T)) \\
\mathsf{init\_Obj}\ \ \Gamma\ C & \stackrel{\mathsf{def}}{=} & \mathsf{Obj}\ C\ (\mathsf{init\_vars}\ (\mathsf{fields}\ \Gamma\ C)) \\
\mathsf{init\_Arr}\ \ T\ i & \stackrel{\mathsf{def}}{=} & \mathsf{Arr}\ T\ (\lambda j.\ \mathsf{if}\ 0 \le j \wedge j < i\ \mathsf{then\ Some}\ (\mathsf{default\_val}\ T)\ \mathsf{else\ None}) \\
\mathsf{raise\_if} & :: & bool \Rightarrow xname \Rightarrow (xcpt)\,option \Rightarrow (xcpt)\,option \\
\mathsf{np} & :: & val \Rightarrow (xcpt)\,option \Rightarrow (xcpt)\,option \\
\mathsf{raise\_if}\ c\ xn\ xo & \stackrel{\mathsf{def}}{=} & \mathsf{if}\ c \wedge (xo = \mathsf{None})\ \mathsf{then\ Some}\ (\mathsf{SysXcpt}\ xn)\ \mathsf{else}\ xo \\
\mathsf{np}\ v & \stackrel{\mathsf{def}}{=} & \mathsf{raise\_if}\ (v = \mathsf{Null})\ \texttt{NullPointer}
\end{array}
$$

The definition of raise_if deserves a comment: raise_if c xn xo either propagates an already thrown exception xo or raises the system exception xn if c is true. As an application, np v checks for a null pointer access through the value v and throws a `NullPointer` exception in this case, but any other exception that has already occurred takes precedence.

Evaluation rule format Internally, the evaluation rules are given as mutually inductive sets of tuples. These sets define relations, which we present as predicates of the following form.

- $\Gamma \vdash \sigma\ -c\rightarrow\ \sigma' :: prog \Rightarrow state \Rightarrow stmt \Rightarrow state \Rightarrow bool$
  means that the execution of statement c transforms state σ into σ'.
- $\Gamma \vdash \sigma\ -e\triangleright v\rightarrow\ \sigma' :: prog \Rightarrow state \Rightarrow expr \Rightarrow val \Rightarrow state \Rightarrow bool$
  means that expression e evaluates to v, transforming σ into σ'.

Although defined as relations (for technical reasons), the semantics given below can be shown to be functional, i.e. deterministic.

Strictly speaking it is not necessary to include an exception in the start state of a computation. 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, we can avoid case distinctions on whether exceptions occur in intermediate states, which would cause the rules to be split. Suppose for example that $\Gamma \vdash \sigma\ -c\rightarrow\ \sigma'$ had the signature $prog \Rightarrow st \Rightarrow stmt \Rightarrow state \Rightarrow bool$, i.e. all rules assume that there is no exception in the start state. Then the rule(s) for sequential composition would look like

$$\frac{\Gamma \vdash \sigma_0\ -c_1\ \rightarrow (\mathsf{None},\sigma_1);\ \ \Gamma \vdash \sigma_1\ -c_2\rightarrow\ \sigma_2}{\Gamma \vdash \sigma_0\ -c_1;\ \ c_2\rightarrow\ \sigma_2}$$

$$\frac{\Gamma \vdash \sigma_0\ -c_1\ \rightarrow (\mathsf{Some}\ xs,\sigma_1)}{\Gamma \vdash \sigma_0\ -c_1;\ \ c_2\rightarrow (\mathsf{Some}\ xs,\sigma_1)}$$

As a consequence of the design decisions just mentioned, there is exactly one rule for each syntactic construct. Additionally there are general rules defining that exceptions simply propagate when a series of statements is executed or a series of expressions is evaluated:

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

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

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

Execution of statements The rules for the statements not explicitly involving exceptions are obvious:

$$\frac{}{\Gamma \vdash \mathsf{Norm}\ \sigma\ -\mathsf{Skip}\rightarrow\ \mathsf{Norm}\ \sigma} \qquad \frac{\Gamma \vdash \mathsf{Norm}\ \sigma_0\ -c_1\ \rightarrow \sigma_1;\ \ \Gamma \vdash \sigma_1\ -c_2\rightarrow\ \sigma_2}{\Gamma \vdash \mathsf{Norm}\ \sigma_0\ -c_1\,;\ c_2\rightarrow\ \sigma_2}$$

$$\frac{\Gamma \vdash \mathsf{Norm}\ \sigma_0\ -e\triangleright v\rightarrow\ \sigma_1}{\Gamma \vdash \mathsf{Norm}\ \sigma_0\ -\mathsf{Expr}\ e\rightarrow\ \sigma_1} \qquad \frac{\begin{array}{c}\Gamma \vdash \mathsf{Norm}\ \sigma_0\ -e\triangleright v\rightarrow\ \sigma_1;\\ \Gamma \vdash \sigma_1\ -\mathsf{if\ the\_Bool}\ v\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\rightarrow\ \sigma_2\end{array}}{\Gamma \vdash \mathsf{Norm}\ \sigma_0\ -\mathtt{if}(e)\ c_1\ \mathtt{else}\ c_2\rightarrow\ \sigma_2}$$

$$\frac{\Gamma \vdash \mathsf{Norm}\ \sigma_0\ -\mathtt{if}(e)\ (c;\ \mathtt{while}(e)\ c)\ \mathtt{else}\ \mathsf{Skip}\rightarrow\ \sigma_1}{\Gamma \vdash \mathsf{Norm}\ \sigma_0\ -\mathtt{while}(e)\ c\rightarrow\ \sigma_1}$$

If no other exceptions have occurred while evaluating its argument and testing for a null reference (using np), the `throw` statement copies the evaluated location into the exception component of the state:

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{Norm}\ \sigma_0\ -e\triangleright a'\rightarrow\ (x_1,\sigma_1);\ \ {x_1}' = \mathsf{np}\ a'\ x_1;\\ {x_1}''=(\mathsf{if}\ {x_1}'=\mathsf{None\ then\ (Some\ (XcptLoc\ (the\_Addr}\ a'))))\ \mathsf{else}\ {x_1}')\end{array}}{\Gamma \vdash \mathsf{Norm}\ \sigma_0\ -\mathtt{throw}\ e\rightarrow\ ({x_1}'',\sigma_1)}$$

For the semantics of the `try _ catch _` statement we have to distinguish whether some exception is thrown and then caught by the `catch` clause or not. In the first case, i.e. there is an exception of appropriate dynamic type to be handled, the `catch` clause is executed with its exception parameter set to the caught exception. In the second case the `catch` clause is skipped. Because of technical limitations of the inductive definition package of Isabelle/HOL, even in this case we have to provide an occurrence of the execution relation, which in effect simply sets σ_2 to $({x_1}',{\sigma_1}')$.

$$\frac{\begin{array}{l}\Gamma \vdash \mathsf{Norm}\ \sigma_0\ -c_1\rightarrow\ \sigma_1;\ \ \Gamma \vdash \sigma_1\ -\mathsf{salloc}\rightarrow\ ({x_1}',{\sigma_1}');\\ \mathsf{case}\ {x_1}'\ \mathsf{of\ None}\quad\ \Rightarrow {\sigma_1}'' = ({x_1}',{\sigma_1}') \wedge {c_2}' = \mathsf{Skip}\\ \qquad\quad \mid \mathsf{Some}\ xc \Rightarrow \mathsf{let}\ a = \mathsf{Addr\ (the\_XcptLoc}\ xc)\ \mathsf{in}\\ \qquad\qquad\qquad\quad\ \mathsf{if}\ \Gamma,{\sigma_1}' \vdash a\ \mathsf{fits\ Class}\ tn\\ \qquad\qquad\qquad\quad\ \mathsf{then}\ {\sigma_1}'' = \mathsf{Norm}\ (\mathsf{lupd}[vn\mapsto a]{\sigma_1}') \wedge {c_2}' = c_2\\ \qquad\qquad\qquad\quad\ \mathsf{else}\ \ {\sigma_1}'' = ({x_1}',{\sigma_1}') \wedge {c_2}' = \mathsf{Skip};\\ \Gamma \vdash {\sigma_1}''\ -{c_2}'\rightarrow\ \sigma_2\end{array}}{\Gamma \vdash \mathsf{Norm}\ \sigma_0\ -(\mathtt{try}\ c_1\ \mathtt{catch}(tn\ vn)\ c_2)\rightarrow\ \sigma_2}$$

On the one hand, the exception parameter of the `catch` clause must represent the exception thrown in the `try` block by a reference to its exception object. As on the other hand we defer the allocation of system exceptions when evaluating expressions, we have to ensure that even for such exceptions a suitable exception object is allocated on the heap of $\sigma_1{}'$, replacing the SysXcpt entry by an XcptLoc entry in $x_1{}'$. This is achieved by the auxiliary relation $\Gamma \vdash \sigma\ {-}\mathsf{salloc}{\rightarrow}\ \sigma' :: \mathit{prog} \Rightarrow \mathit{state} \Rightarrow \mathit{state} \Rightarrow \mathit{bool}$. If no system exception has been thrown, the relation behaves like the identity on the state, and otherwise allocates an exception object and modifies the state accordingly. Note that this allocation step is impossible — and therefore program execution halts — if there is no free address left.

$$\frac{}{\Gamma \vdash \mathsf{Norm}\ \sigma\ {-}\mathsf{salloc}{\rightarrow}\ \mathsf{Norm}\ \sigma}$$

$$\frac{}{\Gamma \vdash (\mathsf{Some}\ (\mathsf{XcptLoc}\ a),\sigma)\ {-}\mathsf{salloc}{\rightarrow}\ (\mathsf{Some}\ (\mathsf{XcptLoc}\ a),\sigma)}$$

$$\frac{\begin{array}{c}\mathsf{new\_Addr}\ (\mathsf{heap}\ \sigma) = \mathsf{Some}\ (a,x);\\ xobj = \mathsf{init\_Obj}\ \Gamma\ (\mathsf{SXcpt}\ (\mathsf{if}\ x = \mathsf{None}\ \mathsf{then}\ xn\ \mathsf{else}\ \texttt{OutOfMemory}))\end{array}}{\Gamma \vdash (\mathsf{Some}\ (\mathsf{SysXcpt}\ xn),\sigma)\ {-}\mathsf{salloc}{\rightarrow}\ (\mathsf{Some}\ (\mathsf{XcptLoc}\ a),\mathsf{hupd}[a{\mapsto}xobj]\sigma)}$$

The `finally` statement is similar to the sequential composition, but executes its second clause regardless whether an exception has been thrown in its first clause or not. If an exception occurs in either clause, it is (re-)raised after the statement, and if both parts throw an exception, the first one takes precedence.

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{Norm}\ \sigma_0\ {-}c_1{\rightarrow}\ (x_1,\sigma_1);\\ \Gamma \vdash \mathsf{Norm}\ \sigma_1\ {-}c_2{\rightarrow}\ (x_2,\sigma_2);\\ x_2{}' = (\mathsf{if}\ x_1 \neq \mathsf{None} \wedge x_2 = \mathsf{None}\ \mathsf{then}\ x_1\ \mathsf{else}\ x_2)\end{array}}{\Gamma \vdash \mathsf{Norm}\ \sigma_0\ {-}(c_1\ \texttt{finally}\ c_2){\rightarrow}\ (x_2{}',\sigma_2)}$$

Evaluation of expressions In contrast to the statement rules, almost all evaluation rules for expressions deserve some comments.

Creating a new class instance means picking a free address a and updating the heap at that address with an object, the fields of which are initialized with default values according to their types. Note that the rule is not applicable — and therefore execution halts — if new_Addr fails.

$$\frac{\mathsf{new\_Addr}\ (\mathsf{heap}\ \sigma) = \mathsf{Some}\ (a,x)}{\Gamma \vdash \mathsf{Norm}\ \sigma\ {-}\mathtt{new}\ C{\triangleright}\mathsf{Addr}\ a{\rightarrow}\ \mathsf{x\_case}\ x\ (\mathsf{hupd}[a{\mapsto}\mathsf{init\_Obj}\ \Gamma\ C]\sigma)\ \sigma}$$

The same applies for the creation of a new array, where additionally an exception is raised if the length of the array is negative:

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{Norm}\ \sigma_0\ {-}e{\triangleright}i'{\rightarrow}\ (x_1,\sigma_1);\ \ i = \mathsf{the\_Intg}\ i';\\ \mathsf{new\_Addr}\ (\mathsf{heap}\ \sigma_1) = \mathsf{Some}\ (a,x);\\ x_1{}' = \mathsf{raise\_if}\ (i{<}0)\ \texttt{NegArrSize}\ (\mathsf{if}\ x_1 = \mathsf{None}\ \mathsf{then}\ x\ \mathsf{else}\ x_1)\end{array}}{\Gamma \vdash \mathsf{Norm}\ \sigma_0\ {-}\mathtt{new}\ T[e]{\triangleright}\mathsf{Addr}\ a{\rightarrow}\ \mathsf{x\_case}\ x_1{}'\ (\mathsf{hupd}[a{\mapsto}\mathsf{init\_Arr}\ T\ i]\sigma_1)\ \sigma_1}$$

A type cast merely returns its argument value, but raises an exception if the dynamic type happens to be unsuitable:

$$\frac{\Gamma \vdash \mathsf{Norm}\ \sigma_0\ -e\triangleright v\rightarrow\ (x_1,\sigma_1);\quad x_1' = \mathsf{raise\_if}(\neg\ \Gamma,\sigma_1 \vdash v\ \mathsf{fits}\ T)\ \mathtt{ClassCast}\ x_1}{\Gamma \vdash \mathsf{Norm}\ \sigma_0\ -(T)\,e\triangleright v\rightarrow\ (x_1',\sigma_1)}$$

The type comparison operator checks if the type of its argument is assignable to the given reference type:

$$\frac{\Gamma \vdash \mathsf{Norm}\ \sigma_0\ -e\triangleright v\rightarrow\ \sigma_1;\quad b = (v{\neq}\mathsf{Null} \wedge \Gamma,\mathsf{snd}\ \sigma_1 \vdash v\ \mathsf{fits}\ \mathsf{RefT}\ T)}{\Gamma \vdash \mathsf{Norm}\ \sigma_0\ -e\ \mathtt{instanceof}\ T\triangleright\mathsf{Bool}\ b\rightarrow\ \sigma_1}$$

The result of a literal expression is simply the given value:

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

An access to a local variable (or the `this` pointer) reads from the local state component:

$$\frac{}{\Gamma \vdash \mathsf{Norm}\ \sigma\ -vn\triangleright\mathsf{the}\ (\mathsf{locals}\ \sigma\ vn)\rightarrow\ \mathsf{Norm}\ \sigma}$$

An assignment to a local variable updates the state, unless the evaluation of the subexpression raises an exception:

$$\frac{\Gamma \vdash \mathsf{Norm}\ \sigma_0\ -e\triangleright v\rightarrow\ \ (x,\sigma_1);\quad \sigma_1' = (\mathsf{if}\ x = \mathsf{None}\ \mathsf{then}\ \mathsf{lupd}[vn{\mapsto}v]\ \sigma_1\ \mathsf{else}\ \sigma_1)}{\Gamma \vdash \mathsf{Norm}\ \sigma_0\ -vn{:=}e\triangleright v\rightarrow\ (x,\sigma_1')}$$

A field access reads from a field of the given object, taking into account the type annotation which yields the defining class of the field as determined statically. It also checks for null pointer access.

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

A field assignment acts accordingly:

$$\frac{\Gamma \vdash \mathsf{Norm}\ \sigma_0\ -e_1\triangleright a'\rightarrow\ (x_1,\sigma_1);\quad a = \mathsf{the\_Addr}\ a';\quad \Gamma \vdash (\mathsf{np}\ a'\ x_1,\sigma_1)\ -e_2\triangleright v\ \rightarrow\ (x_2,\sigma_2);\quad (c,fs)\ =\ \mathsf{the\_Obj}\ (\mathsf{heap}\ \sigma_2\ a);\quad obj = \mathsf{Obj}\ c\ (fs[(fn,T){:=}v])}{\Gamma \vdash \mathsf{Norm}\ \sigma_0\ -(\{T\}e_1.fn{:=}e_2)\triangleright v\rightarrow\ \mathsf{x\_case}\ x_2\ (\mathsf{hupd}[a{\mapsto}obj]\sigma_2)\ \sigma_2}$$

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

$$\frac{\Gamma \vdash \mathsf{Norm}\ \sigma_0\ -e_1\triangleright a'\rightarrow\ \sigma_1;\quad \Gamma \vdash \sigma_1\ -e_2\triangleright i'\rightarrow\ (x_2,\sigma_2);\quad vo\ =\ \mathsf{snd}\ (\mathsf{the\_Arr}\ (\mathsf{heap}\ \sigma_2\ (\mathsf{the\_Addr}\ a')))\ (\mathsf{the\_Intg}\ i');\quad x_2' = \mathsf{raise\_if}\ (vo = \mathsf{None})\ \mathtt{IndOutBound}\ (\mathsf{np}\ a'\ x_2)}{\Gamma \vdash \mathsf{Norm}\ \sigma_0\ -e_1[e_2]\triangleright\mathsf{the}\ vo\rightarrow\ (x_2',\sigma_2)}$$

Similarly, an array assignment updates the appropriate component, but first has to check the type of the value to be assigned. Note one subtle difference to field assignment: null pointer access is checked after evaluating the right-hand side, whereas in field assignment the check occurs immediately after calculating the reference.

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{Norm}\ \sigma_0\ -e_1 \triangleright a' \rightarrow\ \sigma_1;\ \ a = \mathsf{the\_Addr}\ a';\\ \Gamma \vdash \sigma_1\ -e_2 \triangleright i' \rightarrow\ \sigma_2;\ \ i = \mathsf{the\_Intg}\ i';\\ \Gamma \vdash \sigma_2\ -e_3 \triangleright v \rightarrow\ (x_3,\sigma_3);\\ (T,cs) = \mathsf{the\_Arr}\ (\mathsf{heap}\ \sigma_3\ a);\ \ obj = \mathsf{Arr}\ T\ (cs[i \mapsto v]);\\ x_3' = \mathsf{raise\_if}\ (\neg\ \Gamma,\sigma_3 \vdash v\ \mathsf{fits}\ T)\ \texttt{ArrStore}\ (\\ \mathsf{raise\_if}\ (cs\ i = \mathsf{None})\ \texttt{IndOutBound}\ (\mathsf{np}\ a'\ x_3))\end{array}}{\Gamma \vdash \mathsf{Norm}\ \sigma_0\ -(e_1[e_2] := e_3) \triangleright v \rightarrow\ \mathsf{x\_case}\ x_3'\ (\mathsf{hupd}[a \mapsto obj]\sigma_3)\ \sigma_3}$$

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'). For simplicity, we require local variables to be initialized with default values, as the expensive rules for "definite assignment" [GJS96, Ch. 16] merely enable the run-time optimization that variables need not be initialized before being explicitly assigned to. After executing blk and res in the new invocation frame built from the local variables, the parameter pv and a' as the value of `this`, the old invocation frame is restored and the result value v returned:

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{Norm}\ \sigma_0\ -e \triangleright a' \rightarrow\ \sigma_1;\\ \Gamma \vdash \sigma_1\ -p \triangleright pv \rightarrow\ (x_2,\sigma_2);\\ dynT = \mathsf{fst}\ (\mathsf{the\_Obj}\ (\mathsf{heap}\ \sigma_2\ (\mathsf{the\_Addr}\ a')));\\ (md,(pn,rT),lvars,blk,res) = \mathsf{the}\ (\mathsf{cmethd}\ \Gamma\ dynT\ (mn,pT));\\ \Gamma \vdash (\mathsf{np}\ a'\ x_2,(\mathsf{heap}\ \sigma_2,\mathsf{init\_vars}\ lvars[\texttt{this} \mapsto a'][pn \mapsto pv]))\ -blk \rightarrow\ \sigma_3;\\ \Gamma \vdash \sigma_3\ -res \triangleright v \rightarrow\ (x_4,\sigma_4)\end{array}}{\Gamma \vdash \mathsf{Norm}\ \sigma_0\ -(e.mn(\{pT\}p)) \triangleright v \rightarrow\ (x_4,(\mathsf{heap}\ \sigma_4,\mathsf{locals}\ \sigma_2))}$$

Note that all rules are defined carefully in order to be applicable even in not type-correct situations. For example, in any context where a value v is expected to be an address, we do not use a premise like $v = \mathsf{Addr}\ a$ as this will disable the rule if v happens to be, for example, a null pointer or a Boolean value. Instead, we use an expression like $a = \mathsf{the\_Addr}\ v$, which will yield an arbitrary value if v is not an address, yet will leave the rule applicable. In such cases we could not prove anything useful about a, but during the type soundness proof itself it emerges that for well-formed programs (and statically well-typed statements and expressions) such situations cannot occur. A "defensive" evaluation throwing some artificial exception in case of type mismatches, which would require additional overhead, is therefore not necessary.

Below we give a derivation for the execution of our example code *test*, under the assumptions new_Addr empty = Some (a, None) and $\forall obj.$ new_Addr (empty[$a \mapsto obj$]) = Some (b, None), which guarantee that there are at least two free locations on the heap.

We use the following abbreviations:

Γ = *tprg*
NP = `NullPointer`
blk = Expr({ClassT *Ext*}(Class *Ext*)x.*vee*:=Lit (Intg 1))
obj1 = Obj *Ext* (empty[(*vee*, ClassT *Base*)$\mapsto$Bool False] [(*vee*, ClassT *Ext*)$\mapsto$Intg 0])
obj2 = Obj (SXcpt *NP*) empty
h = empty[$a \mapsto$*obj1*]
l = empty[$e \mapsto$Addr a]

Given only the start state σ_0, all other states are computed during the derivation, which results in the sequence

σ_0 = Norm (empty , empty)
σ_1 = Norm (h , l)
σ_2 = Norm (h , empty[`this`$\mapsto$Addr a][$x \mapsto$Null])
σ_3 = (Some (SysXcpt *NP*), h , empty[`this`$\mapsto$Addr a][$x \mapsto$Null])
σ_4 = (Some (SysXcpt *NP*), h , l)
σ_5 = (Some (XcptLoc b), h[$b \mapsto$*obj2*], l)
σ_6 = Norm (h[$b \mapsto$*obj2*], l[$x \mapsto$Addr b])
σ_7 = (Some (XcptLoc b), h[$b \mapsto$*obj2*], l[$x \mapsto$Addr b])

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash \sigma_2 -x\triangleright\text{Null}\rightarrow \sigma_2 \quad \Gamma,\text{snd } \sigma_2 \vdash \text{Null fits Class } Ext}{\Gamma \vdash \sigma_2 -(\text{Class } Ext)x\triangleright\text{Null}\rightarrow \sigma_2} \quad \sigma_3 = (\text{Some (SysXcpt } NP), \text{snd } \sigma_2) \quad \Gamma \vdash \sigma_3 -\text{Lit (Intg 1)}\triangleright\_\rightarrow \sigma_3}{\Gamma \vdash \sigma_2 - (\{\text{ClassT } Ext\}(\text{Class } Ext)x.\,vee\text{:=Lit (Intg 1)})\triangleright\_\rightarrow \sigma_3}}{\Gamma \vdash \sigma_2 -\text{Expr}(\{\text{ClassT } Ext\}(\text{Class } Ext)x.\,vee\text{:=Lit (Intg 1)}) \rightarrow \sigma_3}\ (Blk)$$

$$\dfrac{\dfrac{\Gamma \vdash \sigma_1 -e\triangleright\text{Addr } a\rightarrow \sigma_1 \quad \text{cmethd } \Gamma\ Ext\ (foo, \text{Class } Base) = \text{Some } (\_, (x, \_), [], blk, \text{Lit Null}) \quad \Gamma \vdash \sigma_1 -\text{Lit Null}\triangleright\text{Null}\rightarrow \sigma_1 \quad \sigma_2 = \text{Norm } (h, \text{empty}[\texttt{this}\mapsto\text{Addr } a][x\mapsto\text{Null}]) \quad (Blk) \quad \Gamma \vdash \sigma_3 -\text{Lit Null}\triangleright\_\rightarrow \sigma_3}{\Gamma \vdash \sigma_1 - (e.foo(\{\text{Class } Base\}\text{Lit Null}))\triangleright\_\rightarrow \sigma_4}}{\Gamma \vdash \sigma_1 -\text{Expr}(e.foo(\{\text{Class } Base\}\text{Lit Null})) \rightarrow \sigma_4}\ (Call')$$

$$\dfrac{\dfrac{\text{new\_Addr empty} = \text{Some } (a, \text{None})}{\Gamma \vdash \sigma_0 -\texttt{new } Ext\triangleright a\rightarrow \text{Norm } (h, \text{empty})}}{\dfrac{\Gamma \vdash \sigma_0 - (e\text{:=}\texttt{new } Ext)\triangleright\_\rightarrow \sigma_1}{\Gamma \vdash \sigma_0 -\text{Expr}(e\text{:=}\texttt{new } Ext) \rightarrow \sigma_1}} \qquad \dfrac{\dfrac{\dfrac{\Gamma \vdash \sigma_6 -x\triangleright\text{Addr } b\rightarrow \sigma_6 \quad \sigma_7 = (\text{Some (XcptLoc } b), \text{snd } \sigma_6)}{\Gamma \vdash \sigma_6 -\texttt{throw } x\rightarrow \sigma_7}\ (Throw) \quad \dfrac{\text{new\_Addr } h = \text{Some } (b, \text{None})}{\Gamma \vdash \sigma_4 -\text{salloc}\rightarrow \sigma_5} \quad \Gamma,\text{snd } \sigma_5 \vdash \text{Addr } b \text{ fits Class (SXcpt } NP)}{\Gamma \vdash \sigma_1 -(\texttt{try } \text{Expr}(e.foo(\{\text{Class } Base\}\text{Lit Null}))\ \texttt{catch}((\text{SXcpt } NP)\ x)\ (\texttt{throw } x))\rightarrow \sigma_7}\ (Call')\ (Throw)}{\Gamma \vdash \sigma_0 -(\text{Expr}(e\text{:=}\texttt{new } Ext); \texttt{ try } \text{Expr}(e.foo(\{\text{Class } Base\}\text{Lit Null}))\ \texttt{catch}((\text{SXcpt } NP)\ x)\ (\texttt{throw } x))\rightarrow \sigma_7}$$

5 The proof of type soundness

In this section we discuss our type soundness theorem together with its crucial lemmas. As we spent almost half of the proof effort deriving properties of the type relations and the structure of well-formed programs, we dedicate to them subsections of their own before introducing helpful notions concerning type soundness, the main theorem itself, and interesting corollaries.

It is not surprising that many of them are similar to those given by Drossopoulou and Eisenbach [DE98] since the necessity of certain lemmas emerges quite naturally. On the other hand, the proof principles we use are sometimes rather different from those outlined in their earlier paper [DE97], some of which were inadequate.

5.1 Lemmas on 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

$$\begin{aligned}\mathsf{wf\_prog}\ \Gamma \longrightarrow\ & \mathsf{wf}\ (\lambda(J,I).\ \Gamma \vdash I \prec_{\mathsf{i}} J) \\ \wedge\ & \mathsf{wf}\ (\lambda(D,C).\ \Gamma \vdash C \prec_{\mathsf{c}} D)\end{aligned}$$

and the frequently used transitivity of the widening relation:

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

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. the rule

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

means that for a well-formed program, if some property hold for class `Object` and is preserved by the direct subclass relation, it holds for all classes.

Most lemmas, 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. For example, the transitivity of $\_ \vdash \_ \preceq \_$ is proved by rule induction on the widening relation. It requires a well-formed program because it uses the properties that every class widens to `Object` and that `Object` has neither a superclass nor a superinterface.

5.2 Lemmas on 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 $\{T\}e.fn$, where e is of type Class C, statically refers to a field of type fT defined in the reference type T, then within an instance of some class C', which may be a subclass of C, the field can be (dynamically) referred to using the same name and its defining class. In particular, there is no dynamic binding for fields. This fact requires the following lemma:

wf_prog $\Gamma \wedge$ cfield Γ C fn = Some $(T,fT) \wedge \Gamma \vdash$ Class $C' \preceq$ Class $C \longrightarrow$
table_of (fields Γ C') (fn,T) = Some fT

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 $E \vdash e::$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: for a well-formed program, a reference type T, and any class type T_1 that widens to T, if T (statically) supports a method with a given signature, then the (dynamic) type T_1 supports a method with the same signature and whose result type widens to the result type of the first method:

wf_prog $\Gamma \wedge (m_1,(pn_1,rT_1)) \in$ mheads Γ T $sig \wedge \Gamma \vdash$ Class $T_1 \preceq$ RefT $T \longrightarrow$
$\exists m_2$ pn_2 rT_2 b. cmethd Γ T_1 sig = Some $(m_2,(pn_2,rT_2),b) \wedge \Gamma \vdash rT_2 \preceq rT_1$

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. We motivate how we express type soundness, comment on the proof of the main theorem, and discuss it consequences.

Goal Type soundness is a relation between the type system and the semantics of a language meaning that all values produced during any program execution respect their static types. This can be formulated as a preservation property: For all state transformations caused by executing a statement or evaluating an expression, if in the original state the contents of all variables "conform" to their respective types, this holds also for any final state. Additionally, if an expression yields some result, this value "conforms" to the type of the expression. Of course, we can only expect all this to hold if we assume a well-formed program and well-typed statements and expressions.

It remains to specify what we mean exactly by 'conforms', which is inspired by [DE98]. Relative to a given program Γ and a state σ, a value v conforms to a type T, written $\Gamma,\sigma \vdash v::T$, iff the dynamic type of v widens to T. Via two auxiliary conformance concepts, this can be lifted to the notion of a whole state σ conforming to an environment E. The proposition $\sigma ::\preceq E$ means that the value of any accessible variable within the state is compatible with its static type. Formally, these four concepts

- $\_,\_ \vdash \_ ::\preceq \_ :: prog \Rightarrow st \Rightarrow val \Rightarrow ty \Rightarrow bool$
  of a value conforming to a type,
- $\_,\_ \vdash \_[::\preceq]\_ :: prog \Rightarrow st \Rightarrow (\alpha,val)table \Rightarrow (\alpha,ty)table \Rightarrow bool$
  of all values in a table conforming to their respective types,
- $\_,\_ \vdash \_ ::\preceq \diamond :: prog \Rightarrow st \Rightarrow obj \Rightarrow bool$
  of all components of an object conforming to their respective types, and
- $\_ ::\preceq \_ :: state \Rightarrow env \Rightarrow bool$
  of a state conforming to an environment

are defined as follows:

$$\Gamma,\sigma \vdash v \ ::\preceq \ T \ \stackrel{\text{def}}{=} \ \mathsf{let}\ dyn\_ty = \mathsf{option\_map}\ \mathsf{obj\_ty} \circ \mathsf{heap}\ \sigma \ \mathsf{in}\ \exists T'.\ \mathsf{typeof}\ dyn\_ty\ v = \mathsf{Some}\ T' \wedge \Gamma \vdash T' \preceq T$$

$$\Gamma,\sigma \vdash vs[::\preceq]\,Ts \ \stackrel{\text{def}}{=} \ \forall n\ T.\ Ts\ n = \mathsf{Some}\ T \longrightarrow (\exists v.\ vs\ n = \mathsf{Some}\ v \wedge \Gamma,\sigma \vdash v::\preceq T)$$

$$\Gamma,\sigma \vdash \mathsf{Obj}\ C\ fs ::\preceq \diamond \ = \ \Gamma,\sigma \vdash fs[::\preceq]\mathsf{table\_of}\ (\mathsf{fields}\ \Gamma\ C$$

$$\Gamma,\sigma \vdash \mathsf{Arr}\ T\ cs ::\preceq \diamond \ = \ \Gamma,\sigma \vdash cs[::\preceq]\mathsf{option\_map}\ (\lambda i.\ T) \circ cs$$

$$(x,\sigma) ::\preceq (\Gamma,\Lambda) \ \stackrel{\text{def}}{=} \ \Gamma,\sigma \vdash \mathsf{locals}\ \sigma[::\preceq]\Lambda \wedge (\forall a\ obj.\ \mathsf{heap}\ \sigma\ a = \mathsf{Some}\ obj \longrightarrow \Gamma,\sigma \vdash obj \quad ::\preceq \diamond) \wedge (\forall a.\quad x = \mathsf{Some}(\mathsf{XcptLoc}\ a) \longrightarrow \Gamma,\sigma \vdash \mathsf{Addr}\ a ::\preceq \mathsf{Class}(\mathsf{SXcpt}\ \texttt{Throwable}))$$

The expression $(\mathsf{option\_map}\ \mathsf{obj\_ty} \circ \mathsf{heap}\ \sigma)\ a$ calculates the dynamic type of the object (if any) at address a on the heap. Note that the conformance relation is defined such that it does not take into account inaccessible variables, i.e. values that occur in the state but not in the corresponding component of the static environment. Among others, this frees us from explicitly deallocating exception parameters after a catch clause.

With the help of the notions just introduced, we can express the propositions we aim to prove as follows. 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:

$$E = (\Gamma,\Lambda) \wedge \mathsf{wf\_prog}\ \Gamma \wedge E \vdash s::\diamond \wedge \sigma ::\preceq E \wedge \Gamma \vdash \sigma\ -s\rightarrow\ \sigma' \ \longrightarrow \ \sigma' ::\preceq E$$

Analogously, the evaluation of a well-typed expression preserves the conformance of the state to the environment where, unless an exception has occurred, the value of the expression conforms to its static type:

$$E = (\Gamma,\Lambda) \wedge \mathsf{wf\_prog}\ \Gamma \wedge E \vdash e::T \wedge \sigma ::\preceq E \wedge \Gamma \vdash \sigma\ -e\triangleright v\rightarrow\ (x',\sigma') \ \longrightarrow \ (x',\sigma') ::\preceq E \wedge (x' = \mathsf{None} \longrightarrow \Gamma,\sigma' \vdash v::\preceq T)$$

The validity of these two formulas will result as trivial corollaries from the main theorem, given next.

Main theorem and proof To prove the intended type soundness theorems given above, we utilize rule induction on the derivation on the execution of statements and the evaluation of expressions. As these depend on each other, we must deal with statements and expressions simultaneously. In addition, in order to obtain a suitable induction hypothesis, we have to strengthen the propositions by adding the auxiliary "heap extension" predicate $\_ \trianglelefteq \_$ (defined below) and introducing universal quantifications explicitly at several positions. As a result, the main theorem looks quite formidable, yet we attempt to cast it into words:

$$
\begin{array}{l}
\mathsf{wf\_prog}\ \Gamma \longrightarrow \\
\quad (\Gamma \vdash (x,\sigma)\ -c\ \rightarrow (x',\sigma') \longrightarrow \\
\qquad \forall \Lambda.\ (x,\sigma) :: \preceq (\Gamma,\Lambda) \longrightarrow \\
\qquad\quad (\Gamma,\Lambda) \vdash c :: \diamond \longrightarrow \\
\qquad\quad (x',\sigma') :: \preceq (\Gamma,\Lambda) \wedge \sigma \trianglelefteq \sigma' \\
\wedge \\
\quad (\Gamma \vdash (x,\sigma)\ -e \triangleright v \rightarrow (x',\sigma') \longrightarrow \\
\qquad \forall \Lambda.\ (x,\sigma) :: \preceq (\Gamma,\Lambda) \longrightarrow \\
\qquad \forall T.\ (\Gamma,\Lambda) \vdash e :: T \longrightarrow \\
\qquad\quad (x',\sigma') :: \preceq (\Gamma,\Lambda) \wedge \sigma \trianglelefteq \sigma' \wedge (x' = \mathsf{None} \longrightarrow \Gamma,\sigma' \vdash v :: \preceq T))
\end{array}
$$

For a well-formed program Γ, if the execution of a statement transforms one state into another then for all local environments Λ, if the the statement is well-typed according to the environment (Γ,Λ) and the first state conforms to it, so does the second state, and the new heap is an extension of the old one. The same holds for expressions, but additionally the value of the expression conforms to its type, in case there is no exception.

The "heap extension" is a pre-order on states of type $st \Rightarrow st \Rightarrow bool$, where $\sigma \trianglelefteq \sigma'$ means that any object existing on the heap of σ also exists on σ' and has the same type there. (If we considered garbage collection, we would have to restrict this proposition to accessible objects.) The heap extension property holds for any transition of the operational semantics, which turns out to be necessary in our inductive proof.

$$
\begin{array}{l}
\sigma \trianglelefteq \sigma' \stackrel{\mathsf{def}}{=} \forall a\ obj.\ \ \mathsf{heap}\ \sigma\ \ a = \mathsf{Some}\ obj \longrightarrow \\
\qquad\qquad \exists obj'.\ \mathsf{heap}\ \sigma'\ a = \mathsf{Some}\ obj' \wedge \mathsf{obj\_ty}\ obj' = \mathsf{obj\_ty}\ obj
\end{array}
$$

The proof of the main type soundness theorem is by far the heaviest. At the top level, it consists of currently 21 cases, one for each evaluation rule, where

- 8 cases can be solved rather directly (e.g. from the induction hypothesis),
- 7 cases require just simple lemmas on the structure of the state, and
- the remaining 6 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 tackled separately, which keeps the main proof manageable.

Consequences A corollary of type soundness is that method calls always execute a suitable method, i.e. a 'method not understood' run-time error is impossible. This property 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 for that (static) type and a given signature a method is available, the dynamic method lookup for the same signature according to the class instance pointed at by the reference value yields a proper method body:

$E = (\Gamma,\Lambda) \wedge \mathsf{wf\_prog}\ \Gamma \wedge E \vdash e::\mathsf{RefT}\ T \wedge \sigma ::\preceq E \wedge \Gamma \vdash \sigma\ -e\triangleright a'\rightarrow \mathsf{Norm}\ \sigma' \wedge$
$a' \neq \mathsf{Val\ Null} \wedge dynT = \mathsf{fst}\ (\mathsf{the\_Obj}\ (\mathsf{heap}\ \sigma'\ (\mathsf{the\_Addr}\ a'))) \wedge$
$\mathsf{mheads}\ \Gamma\ T\ sig \neq \{\} \longrightarrow \exists m.\ \mathsf{cmethd}\ \Gamma\ dynT\ sig = \mathsf{Some}\ m$

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.

As it stands, the type soundness theorem does not directly say anything about non-terminating computations, which might lead to the conclusion that it is useless for the type-safety of reactive systems and looping programs. Fortunately, the theorem guarantees type-safety even in such cases if one accepts the following meta-level reasoning. An infinite computation can be interrupted after any finite number of computation steps, for example by introducing a counter of steps and raising an exception when a given value has been reached. The theorem implies that the state resulting from interrupting the computation after any finite number of statements executed conforms to the environment. Together with the fact that there is no single non-terminating statement, the whole (infinite) computation can be concluded to be type-safe.

In addition to the evaluation semantics, we plan to define a transition semantics and prove both styles equivalent (for finite computations). The transition semantics will be less concise and abstract, but allows type soundness to be formulated as a subject reduction property, which is more natural for infinite computations. More importantly, it seems to be unavoidable to describe concurrency (and I/O).

6 Experience and statistics

Recalling our design goals stated at the end of §2, we comment how far we have reached them and share some of the lessons learned during the project.

Faithfulness to the official language specification HOL's expressiveness enables us to formalize the Java specification quite naturally and directly, without facing any severe obstacles. There is almost a one-to-one correspondence between the concepts given in the specification and those defined in Bali. As far as we could tell, all the messy well-formedness conditions inherited from the language specification are actually needed somewhere in the proofs. This inspires confidence in the adequacy of both the specification and our formalization.

We do not yet have tools for automatically generating executable code from our theories, which would be an additional help in validating our formalization. The importance of such a mechanism became very obvious when we uncovered a mistake in our formalization (which was not present in [NO98] but was introduced by modifications) when symbolically executing the example in this article in Isabelle: the list returned by function fields was in reverse order. Although the type soundness proof itself was an excellent debugging mechanism which caught many minor and some major mistakes, it failed to detect the wrong order because type soundness is independent of the order in which fields are inherited. In the original language specification we did not find any significant errors, but some omissions and unneeded restrictions, which we lifted.

Succinctness and simplicity Our policy to restrict the number of features considered and to make straightforward simplifications that do not diminish the expressiveness of the language has lead to a clear and straightforward formalization. Mixfix syntax and mathematical fonts as offered by Isabelle also contribute greatly to moderately readable definitions and theorems.

The facility to conduct concise proofs strongly depends on the formalization. In our case, the use of the (also more elegant) evaluation semantics saved us from a lot of trouble, while the intricacies of a transition semantics faced by Drossopoulou and Eisenbach [DE97] lead to several mistakes that were finally corrected during Syme's machine-checked proof [Sym97b], but at the expense of additional concepts.

Maintainability and extendibility Unless the language changes drastically, modifications tend to be of a local nature, but only if both the formalization and the proofs are reasonably structured. As always, modularity is the key issue. But when the formalization is extended, even well-structured proofs need to be modified, which remains a tedious job. Higher-level proof scripts and more automation are some of the answers. A dedicated mechanism for change management exploring and fixing the impact of modifications would also help.

We are reasonably happy with the modularity of our work. For instance, Martin Büchi [BW98] has adpoted the formalization (including the proofs), extended it to handle compound types, and proved the type-safety of the augmented language, all of which worked very smoothy.

Adequacy for the theorem prover Theorem provers are notoriously sensitive to the precise formulation of definitions and theorems. Thus the two goals of maximal automation of proofs and maximal abstractness of definitions are sometimes in conflict. In a number of cases this meant that although we could start with an abstract definition, we had to derive consequences which were better suited for the available proof procedures. 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 by extending the capabilities of Isabelle itself.

Statistics We spent two months (estimated net time) developing and maintaining our formalization, and the Isabelle theory files produced add up to about 1200 lines of well-documented definitions. To conduct and maintain the type soundness proof with all necessary lemmas, it took us roughly three months of work and about 2400 lines of proof scripts.

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 at the beginning of §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 [Dea97] 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 and extensions, 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. This has shown to be important, for example when we extended Bali with full exception handling. It will continue to help us further: apart from adding the last important Java features missing from Bali, e.g. threads, we also plan to use Bali as a vehicle for experimental extensions of Java such as parameterized types [MBL97,OW97,AFM97].

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 Java type system is, despite subclassing, simpler than that of your average functional language: whereas the type checking rules of Java are almost directly executable, the verification of ML's type inference algorithm against the type system requires a significant effort [NN98]. The key complication there is the presence of free and bound type variables, which requires complex reasoning about substitutions. VanInwegen [Van97] 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 [Sym97a] promise to lower this hurdle. However, as Harrison [Har97] points out, both proof styles have their merits, and we are currently investigating a combination.

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

Acknowledgments We thank Sophia Drossopoulou, Donald Syme and Egon Börger for the very helpful discussions about their related work. We also thank Wolfgang Naraschewski, Markus Wenzel, Andrew Gordon and several anonymous referees for their comments on earlier reports on our project.

References

[AFM97] Ole Agesen, Stephen N. Freund, and John C. Mitchell. Adding type parameterization to the Java language. In *ACM Symp. Object-Oriented Programming: Systems, Languages and Applications*, 1997.

[BCM+93] Kim B. Bruce, Jon Crabtree, Thomas P. Murtagh, Robert van Gent, Allyn Dimock, and Robert Muller. Safe and decidable type checking in an object-oriented language. In *Proc. OOPSLA'93*, volume 18 of *ACM SIGPLAN Notices*, pages 29–46, October 1993.

[BM88] Robert S. Boyer and J Strother Moore. *A Computational Logic Handbook*. Academic Press, 1988.

[Bru93] Kim 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.

[BS98] Egon Börger and Wolfram Schulte. A programmer friendly modular definition of the dynamic semantics of Java. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, Lect. Notes in Comp. Sci. Springer-Verlag, 1998. Chapter 11 of this volume.

[BvGS95] Kim B. Bruce, Robert van Gent, and Angela 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.

[BW98] Martin Büchi and Wolfgang Weck. Java needs compound types. Technical Report 182, Turku Center for Computer Science, May 1998. http://www.abo.fi/~mbuechi/publications/CompoundTypes.html.

[Coh97] Richard M. Cohen. The defensive Java Virtual Machine specification. Technical report, Computational Logic Inc., 1997. Draft version.

[Coo89] William Cook. A proposal for making Eiffel type-safe. In *Proc. ECOOP'89*, pages 57–70. Cambridge University Press, 1989.

[DE97] Sophia Drossopoulou and Susan Eisenbach. Is the Java type system sound? In *Proc. 4th Int. Workshop Foundations of Object-Oriented Languages*, January 1997.

[DE98] Sophia Drossopoulou and Susan Eisenbach. Java is type safe — probably. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, Lect. Notes in Comp. Sci. Springer-Verlag, 1998. Chapter 3 of this volume.

[Dea97] Drew Dean. The security of static typing with dynamic linking. In *Proc. 4th ACM Conf. Computer and Communications Security*. ACM Press, 1997.

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

[GM93] M.J.C. Gordon and T.F. Melham, editors. *Introduction to HOL: a theorem-proving environment for higher order logic*. Cambridge University Press, 1993.

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

[MBL97] Andrew C. Myers, Joseph A. Bank, and Barbara Liskov. Parameterized types for Java. In *Proc. 24th ACM Symp. Principles of Programming Languages*, pages 132–145, 1997.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *J. Comp. Sys. Sci.*, 17:348–375, 1978.

[NN98] Wolfgang Naraschewski and Tobias Nipkow. Type inference verified: Algorithm W in Isabelle/HOL. In E. Giménez and C. Paulin-Mohring, editors, *Types for Proofs and Programs: Intl. Workshop TYPES '96*, volume 1512 of *Lect. Notes in Comp. Sci.*, pages 317–332. Springer-Verlag, 1998.

[NO98] Tobias Nipkow and David von Oheimb. Java$_{light}$ is type-safe — definitely. In *Proc. 25th ACM Symp. Principles of Programming Languages*, pages 161–170. ACM Press, 1998.

[OW97] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *Proc. 24th ACM Symp. Principles of Programming Languages*, pages 146–159, 1997.

[Pau94] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 1994.

[Sli96] Konrad 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.

[Sym97a] Donald Syme. DECLARE: A prototype declarative proof system for higher order logic. Technical Report 416, University of Cambridge Computer Laboratory, 1997.

[Sym97b] Donald Syme. Proving Java type soundness. Technical Report 427, University of Cambridge Computer Laboratory, 1997.

[Sym98] Donald Syme. Proving Java type soundness. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, Lect. Notes in Comp. Sci. Springer-Verlag, 1998. Chapter 4 of this volume.

[Van97] Myra VanInwegen. Towards type preservation for core SML. University of Cambridge Computer Laboratory, 1997.

Index