μJava: Embedding a Programming Language in a Theorem Prover

Tobias Nipkow and David von Oheimb and Cornelia Pusch
Institut für Informatik
Technische Universität München
80290 München, Germany
`http://www.in.tum.de/~{nipkow,oheimb,pusch}`

Abstract. This paper introduces the subset μJava of Java, essentially by omitting everything but classes. The type system and semantics of this language (and a corresponding abstract Machine μJVM) are formalized in the theorem prover Isabelle/HOL. Type safety both of μJava and the μJVM are mechanically verified.

To make the paper self-contained, it begins with introductions to Isabelle/HOL and the art of embedding languages in theorem provers.

1 Introduction

Embedding a programming language in a theorem prover means to describe (parts of) the language in the logic of the theorem prover, for example the abstract syntax, the semantics, the type system, a Hoare logic, a compiler, etc. One could call this *applied machine-checked semantics*.

Why should we want to do this? We have to distinguish two possible applications:

- Proving theorems about programs. This is usually called *program analysis* or *verification* and will not concern us very much in this paper.
- Proving theorems about the programming language. This is meta-theory and could be called *language analysis*. It is the very focus of our work.

Since programming languages are complex entities, the machine, i.e. the theorem prover, helps us to stay honest and also forces us to look for the simplest possible formalization of all concepts.

The purpose of this paper is to describe a particular such embedding, namely the definition of μJava, a fragment of Java, in the theorem prover Isabelle/HOL which is based on higher-order logic. The main analysis that we discuss is the proof of type safety, i.e. the absence of (unchecked) run-time type errors.

The plan of the paper is as follows. In the first third, we survey the necessary background information: after a few general remarks about higher-order logic (§2) we introduce the theorem prover Isabelle/HOL (§3) and discuss the main principles of language embeddings (§4). The other two thirds are dedicated to μJava and the corresponding virtual machine, the μJVM. For both levels, the type system and semantics of the language are presented and type safety is proved. Discussion of related work is found in the individual sections.

2 Higher-order logic

The term *higher-order logic* traditionally means a typed logic that permits quantification over functions or sets. A very specific example of such a logic is Church's *simple theory of types* [Chu40, And86]. In the theorem proving community, higher-order logic is often abbreviated to HOL and refers to the simple theory of types. We follow this convention which is due to one of the first theorem provers for this logic, Mike Gordon's HOL system [Gor85, GM93], a descendant of LCF [Pau87].

The work reported in this paper has been conducted with the help of Isabelle/HOL: Isabelle [Pau94] is a generic interactive theorem prover, and Isabelle/HOL an instance supporting HOL. There are many other systems supporting HOL, and many other higher-order logics. Mike Gordon's HOL system has spawned many closely related implementations. There is even an automatic theorem prover for HOL, the TPS system [ABI+96]—most other theorem provers for higher-order logics are interactive. Then there is the area of *type theories*, which are constructive higher-order logics with sophisticated type systems. This area is marked by its proliferation of different although related logics, many of which are supported by their own theorem prover. The most prominent of these provers are Coq [BBC+97], Nuprl [C+86] and Lego [Pol94]. Also based on type theory are the Elf and Twelf systems [Pfe91, PS99]. Strictly speaking the latter are not fully fledged theorem provers but logical frameworks specifically designed for prototyping but also (automatic) reasoning about deductive systems, in particular operational descriptions of programming languages.

3 Modeling and proving in Isabelle/HOL

What is HOL? In a nutshell, it is a classical (i.e. two-valued) logic with equality and total polymorphic higher-order functions. Therefore familiarity with classical predicate logic and functional programming is all one needs when reading this paper, since we will not be concerned with the low level details in proofs. The system Isabelle/HOL is more than just a theorem prover for HOL, it is a fully fledged specification and programming language. Coming from a programming perspective, one could characterize Isabelle/HOL as combination of functional programming, logic programming and quantifiers. The remainder of this section describes the main features of Isabelle/HOL from an abstract perspective. For more details see the Isabelle/HOL tutorial [Nip99a].

3.1 Terms, types, formulae and theories

The type system is similar to that of typed functional programming languages like ML, Haskell and Gofer. There are basic types like *bool*, *nat* and *int*, and type constructors (written postfix) like $(\tau)list$ and $(\tau)set$ (where τ is the argument type). Function types are written $\tau_1 \Rightarrow \tau_2$ and represent the type of all *total* functions. Type variables, which are used to express polymorphism, are written α, β etc.

Terms are formed as in λ-calculus by application and abstraction. The constructions $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$, $\mathsf{if}\ b\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$ and $\mathsf{case}\ e\ \mathsf{of}\ p_1 \to e_1 \mid \ldots$ familiar from functional programming are also supported.

Formulae are terms of type *bool*. HOL offers the usual logical vocabulary. The notation

$$\frac{A_1 \quad \ldots \quad A_n}{C}$$

means $A_1 \wedge \ldots \wedge A_n \longrightarrow C$.

Modules in Isabelle/HOL are called *theories* to emphasize their mathematical content. They contain collections of declarations and definitions of types and constants (which include functions). You could also call them specifications or programs, depending on your point of view. Although you can in principle add new axioms as well, this is strongly discouraged because of the following variant of Murphy's law:

What can be inconsistent will be inconsistent.

Therefore the HOL dogma (going back to Gordon's HOL system) is never to add new axioms, only definitions, because the latter preserve consistency. However, working only with basic non-recursive definitions of the form

$$name \equiv term$$

can be very cumbersome. Therefore Isabelle/HOL additionally provides several derived application-oriented definition principles: recursive datatypes, recursive functions and inductively defined sets. We discuss them in turn.

3.2 Functional programming

Functional programming is supported by constructs for the definition of recursive datatypes and recursive functions. A simple example is the theory of lists:

datatype α *list* = Nil ([])
| Cons α (α *list*) (**infixr** #)

consts app :: α *list* $\to$ α *list* $\to$ α *list* (**infixr** @)

primrec
[] @ *ys* = *ys*
(*x* # *xs*) @ *ys* = *x* # (*xs* @ *ys*)

It defines the recursive datatype of lists together with some syntactic sugar (Nil can be written [] and Cons *x xs* can be written *x*#*xs*), declares a function *app* (with infix syntax @), and defines *app* by primitive recursion. The key proof technique in this setting is structural induction on datatypes. For example, associativity of @

$$(xs \mathbin{@} ys) \mathbin{@} zs = xs \mathbin{@} (ys \mathbin{@} zs)$$

is proved by (structural) induction on *xs*:

Nil: ([] @ *ys*) @ *zs* = *ys* @ *zs* = [] @ (*ys* @ *zs*)

Cons: ((*x* # *xs*) @ *ys*) @ *zs* = (*x* # (*xs* @ *ys*)) @ *zs* =
x # ((*xs* @ *ys*) @ *zs*) = *x* # (*xs* @ (*ys* @ *zs*)) = (*x*#*xs*) @ (*ys* @ *zs*)

Such proofs are performed automatically by Isabelle/HOL.

More complex recursion patterns can be expressed by *well-founded recursion*, which requires a termination ordering to convince Isabelle/HOL of the totality of the defined function. More precisely, the ordering is used in a non-recursive definition of the function from which the desired recursion equations are proved as theorems. For details see the work by Slind [Sli96, Sli97, Sli99].

Totality is always the key requirement when defining a function in HOL since HOL is a logic of total functions, and the introduction of a truly partial function would cause an inconsistency.

3.3 Inductively defined sets

Isabelle/HOL comes with a type $(\tau)set$ of (finite and infinite) sets over type τ and together with the usual operations. Set comprehension is written $\{x\ .\ P\}$ instead of $\{x \mid P\}$.

Sets can be defined inductively in Isabelle/HOL, for example the set of even numbers:

consts even :: *nat set*

inductive

$$0 \in \mathsf{even} \qquad \frac{n \in \mathsf{even}}{n+2 \in \mathsf{even}}$$

The importance of this mechanism cannot be overestimated because computer science abounds with inductive processes.

In general, an inductive definition looks like this

consts M :: τ *set*

inductive

$$\frac{t_1 \in \mathsf{M} \quad \ldots \quad t_n \in \mathsf{M}}{t \in \mathsf{M}}$$

and defines M as the *least* subset of τ which satisfies the given rules. There can be any finite number of rules of the given format. In fact, a more general rule format is supported, but this one suffices for our purposes.

Leastness is the mathematical way of saying that the given rules are the only rules that define M. From a programmer's perspective, inductive definitions can be seen as providing logic programming with closed world assumption. This gives rise to the principle of *rule induction*. For example,

$$n \in \mathsf{even} \longrightarrow n+n \in \mathsf{even}$$

is proved by induction on the derivation of $n \in$ even:

rule 1: $0+0 \in$ even

rule 2: $n+n \in \mathsf{even} \longrightarrow n+n+2 \in \mathsf{even} \longrightarrow n+n+2+2 \in \mathsf{even} \longrightarrow (n+2)+(n+2) \in \mathsf{even}$

In general, $x \in \mathsf{M} \longrightarrow P(x)$ is proved by rule induction on $x \in$ M, i.e. by showing that P is preserved by every rule:

$$\frac{P(t_1) \quad \ldots \quad P(t_n)}{P(t)}$$

In fact, one may also assume $t_i \in M$ for all $i = 1, \ldots, n$. Isabelle/HOL supports proof by rule induction.

3.4 Peeking at the library

The formalization of Java, as any large project, makes use of many predefined theories. We quickly summarize the most important ones.

3.4.1 Cartesian products

The type $\tau_1 \times \tau_2$ of Cartesian products of τ_1 and τ_2 comes with projection functions fst :: $\alpha \times \beta \to \alpha$ and snd :: $\alpha \times \beta \to \beta$, and with (postfix) operations for constructing the transitive closure R^+ and transitive reflexive closure R^* of a relation $R :: (\alpha \times \alpha)set$.

3.4.2 Lists

In addition to the basic list constructs shown above, the list library contains the following relevant functions:

length :: α *list* $\to$ *nat*
set :: α *list* $\to$ α *set*
map :: $(\alpha \to \beta) \to \alpha$ *list* $\to \beta$ *list*
zip :: α *list* $\to \beta$ *list* $\to (\alpha \times \beta)$*list*
nodups :: α *list* $\to$ *bool*
nth :: α *list* $\to$ *nat* $\to \alpha$

The meaning of length, set, map and zip should be obvious; nodups *xs* is true iff *xs* contains no duplicates, and nth *xs i* selects the *i*-th element (starting from 0) and is abbreviated by *xs*!*i*. The usual notation [*x*,*y*,*z*] instead of *x*#*y*#*z*#[] is also supported.

3.4.3 Options

The datatype of optional values

datatype α *option* = None | Some α

consts the :: α *option* $\to \alpha$

primrec the (Some x) = x
the None = ...

is used to add a new element None to a type and wrap the remaining elements up in Some; function the unwraps them again; on None it can be defined arbitrarily.

3.4.4 Mappings

One frequently needs partial functions where one can determine if an entry is defined or not. We call them mappings and define them as functions with optional range type:

types $\alpha \rightharpoonup \beta$ = $\alpha \to \beta$ *option*

Typical applications are symbol tables (where declared names are mapped to some information) or heap storage (where allocated addresses are mapped to their content).

For the convenient manipulation of mappings the following functions are provided:

empty :: $\alpha \rightharpoonup \beta$
_(_$\mapsto$_) :: $(\alpha \rightharpoonup \beta) \to \alpha \to \beta \to (\alpha \rightharpoonup \beta)$
_(_[$\mapsto$]_) :: $(\alpha \rightharpoonup \beta) \to \alpha$ *list* $\to \beta$ *list* $\to (\alpha \rightharpoonup \beta)$
$\oplus$ :: $(\alpha \rightharpoonup \beta) \to (\alpha \rightharpoonup \beta) \to (\alpha \rightharpoonup \beta)$
map_of :: $(\alpha \times \beta)$ *list* $\to (\alpha \rightharpoonup \beta)$

They represent the empty mapping, updating in one place and in a list of places, overwriting of one map with another, and turning an association list into a mapping. The definitions are unsurprising:

empty $\equiv \lambda k.$ None
$m(x \mapsto y)$ $\equiv \lambda k.$ if k=x then Some y else $m\ k$

$s \oplus t$ $\equiv \lambda k.$ case $t\ k$ of None $\to s\ k$ | Some $y \to$ Some y

map_of [] = empty
map_of ((*x*,*y*)#*l*) = (map_of l)($x \mapsto y$)

m([] [$\mapsto$] []) = m
$m(a\#as[\mapsto]b\#bs)$ = $m(a \mapsto b)(as[\mapsto]bs)$

4 The basics of programming language embeddings

Historically, this work has its roots in research on generating programming environments, especially compilers and interpreters, from formal language descriptions. Prominent systems of this kind are PSG [Sne85] (based on denotational semantics), ASF+SDF [Kli93] (based on algebraic specifications) and Centaur [BCD+88]. Centaur is quite close to our framework because it is based on operational semantics expressed by inference rules, a connection that has already been exploited by Bertot and Fraer [BF95].

However, the first embedding of a semantics in a theorem prover is due to Gordon [Gor89], who defined the semantics of a simple `while`-language in the HOL system and derived the rules of Hoare logic as theorems. The attractiveness of this approach soon lead other researchers to embed various programming languages in various theorem provers: CCS [Nes94], CSP [TW97], Java [Sym99, ON99, JHvB+98], JVM [Coh97, Pus99], ML [Sym94, VG94, MG94], UNITY [APP94, Pra95, HC96, Pau99] and Hoare logics [Nip98, Kle98, Ohe99]. This is just a random selection and not a complete list. We will now discuss the basic principles of language embeddings.

The literature [BGG+92] distinguishes two different kinds of embeddings:

deep embeddings represent the abstract syntax of the language as a separate datatype and define the semantics using the syntax (for example as a function from syntax to semantics).

shallow embeddings define the semantics directly, i.e. each construct in the language is represented directly by some function on a semantic domain.

The difference is best explained by an example.

Assume we want to embed boolean expressions consisting of variables, conjunctions and disjunctions in Isabelle/HOL. Let us call them *positive boolean expressions*. The semantics of a boolean expression is a function from environments to boolean values, where environments map variables (we assume we are given a type *var* of variables) to boolean values:

types $env = var \to bool$
$sem = env \to bool$

A shallow embedding represents the semantics only, i.e. it identifies the positive boolean expressions with type *sem*. Each language construct is directly defined in terms of semantics:

$\mathsf{var}\ p \equiv \lambda e.\ e\ p, \qquad \mathsf{and}\ b_1\ b_2 \equiv \lambda e.\ b_1\ e \wedge b_2\ e, \qquad \mathsf{or}\ b_1\ b_2 \equiv \lambda e.\ b_1\ e \vee b_2\ e$

Thus the boolean expression "$p \wedge q$" (where p and q are variables) is represented directly by the HOL function $\lambda e.\ e(p) \wedge e(q)$ of type *sem*.

A deep embedding requires a definition of the syntax, usually as an inductive type:

datatype $pbex = \mathsf{Var}\ var \mid \mathsf{And}\ pbex\ pbex \mid \mathsf{Or}\ pbex\ pbex$

The semantics is defined by an explicit function $eval :: pbex \to sem$:

$\mathsf{eval}\ (\mathsf{Var}\ p) = \lambda e.\ e(p)$
$\mathsf{eval}\ (\mathsf{And}\ b_1\ b_2) = \lambda e.\ (\mathsf{eval}\ b_1\ e \wedge \mathsf{eval}\ b_2\ e)$
$\mathsf{eval}\ (\mathsf{Or}\ b_1\ b_2) = \lambda e.\ (\mathsf{eval}\ b_1\ e \vee \mathsf{eval}\ b_2\ e)$

The boolean expression "$p \wedge q$" is represented by And (Var p) (Var p) and applying $eval$ to it yields its semantics $\lambda e.\ e(p) \wedge e(q)$.

Both embeddings allow proofs about individual boolean expressions, for example "$p \wedge p = p$". In a shallow embedding this reduces (via extensionality of functions) to the idempotence of $\wedge$ in HOL:

$((\lambda e.\ e(p) \wedge e(p)) = (\lambda e.\ e(p))) \ = \ (e(p) \wedge e(p) = e(p)) \ = \ \mathsf{True}$

The deep embedding has to be reduced to the shallow one first before the truth emerges:

$(\mathsf{eval}(\mathsf{And}\ (\mathsf{Var}\ p)\ (\mathsf{Var}\ p)) = \mathsf{eval}(\mathsf{Var}\ p)) \ = \ ((\lambda e.\ e(p) \wedge e(p)) = (\lambda e.\ e(p))) \ = \ \mathsf{True}$

This indicates that deep embeddings are often harder to work with than shallow ones because the syntax has to be translated into semantics first.

However, the tide turns against shallow embeddings when we consider general statements about the embedded language as a whole, i.e. meta-theory. For example, we would like to prove monotonicity of positive boolean expressions. To this end we order *bool* and *env* in the canonical way, i.e. False $<$ True and *env* is ordered pointwise:

$x \le y \ \equiv \ x \longrightarrow y$
$e_1 \le_e e_2 \equiv \ \forall p.\ e_1(p) \le e_2(p)$

In the deep embedding, monotonicity of $pb :: pbex$ is expressed as

$$e_1 \le_e e_2 \ \longrightarrow \ \mathsf{eval}\ pb\ e_1 \le \mathsf{eval}\ pb\ e_2$$

and proved by induction on pb. The Var-case is straightforward by assumption:

$\mathsf{eval}\ (\mathsf{Var}\ p)\ e_1 = e_1(p) \le e_2(p) = \mathsf{eval}\ (\mathsf{Var}\ p)\ e_2$.

The And-case (and similarly the Or-case) follows from the monotonicity of $\wedge$ in HOL using the induction hypotheses:

$\mathsf{eval}\ (\mathsf{And}\ x\ y)\ e_1 = \mathsf{eval}\ x\ e_1 \wedge \mathsf{eval}\ y\ e_1 \le \mathsf{eval}\ x\ e_2 \wedge \mathsf{eval}\ y\ e_2 = \mathsf{eval}\ (\mathsf{And}\ x\ y)\ e_2$

In the shallow embedding however, monotonicity of $f :: sem$, i.e.

$e_1 \le_e e_2 \ \longrightarrow \ f\ e_1 \le f\ e_2$

is not true because *sem* contains the semantics of *all* boolean expressions, not just the positive ones. The restriction to variables, conjunctions and disjunctions is not part of the specification and thus not available for an inductive proof. Hence it is folklore that shallow embeddings do not allow meta-theory. However, this is not true as Felty *et al.* [FHR99] have shown: one can often define the required subset of the semantics via an inductive definition that follows the syntax. In our case we define a subset Pbex of denotations of positive boolean expressions:

consts Pbex :: *sem set*

inductive

$$\frac{}{\mathsf{var}\ p \in \mathsf{Pbex}} \qquad \frac{b_1 \in \mathsf{Pbex} \qquad b_2 \in \mathsf{Pbex}}{\mathsf{and}\ b_1\ b_2 \in \mathsf{Pbex}} \qquad \frac{b_1 \in \mathsf{Pbex} \qquad b_2 \in \mathsf{Pbex}}{\mathsf{or}\ b_1\ b_2 \in \mathsf{Pbex}}$$

Monotonicity of Pbex is expressed as $b \in \mathsf{Pbex} \longrightarrow \ e_1 \le_e e_2 \longrightarrow \ b\ e_1 \le b\ e_2$ and proved by rule induction on $b \in \mathsf{Pbex}$. The proof follows the one above for *pbex* very closely.

It should be clear from our discussion that a shallow embedding is advantageous for reasoning about individual elements of the embedded language. For meta-theory a deep embedding is the natural choice, but combining a shallow embedding with an inductive definition may also work. Our Java formalization presented below is a deep embedding because of our interest in meta-theory and also because the semantics is defined operationally rather than denotationally. Thus there is no direct semantic counterpart for each syntactic construct.

5 μJava

This section describes the μJava formalization and one of its applications, a proof of type-safety. After motivating the subset of Java that we chose and giving a short survey of related work, we present the abstract syntax and operational semantics of μJava along with the corresponding well-formedness and well-typedness conditions. Subsequently we introduce the notions required for the type-safety proof and state the main results.

5.1 Why μJava?

Being a general-purpose programming language, Java has (almost) all features a state-of-the-art language is supposed to offer. Formalizing all of them is possible, but the amount of detail and the complexity of several aspects like concurrency would make it involved and difficult to handle. Thus a typical formalization like Bali [ON99] on the one hand tries to cover the main features of Java, but on the other hand intentionally leaves out many bulky but uninteresting details. For didactic purposes, the result is still too involved. So we further reduced Bali to the bare essentials of Java, namely an imperative language with classes: μJava.

5.2 Other approaches to formalizing Java

There are several formalizations of Java, the pioneering one being of Drossopoulou and Eisenbach [DE97]. It gives a transition ("small-step") semantics of the core object-oriented features of Java and a proof of type-safety, which has been extended later [DE99] to include exception handling. Syme has embedded this paper-and-pencil work in his theorem prover DECLARE [Sym99], correcting several flaws that came up thanks to the rigorous machine-checked treatment. In parallel, we have developed the first version of our embedding in Isabelle/HOL [NO98] covering a similar fraction of Java but using an evaluation ("big-step") semantics. Börger and Schulte have formalized (on paper) almost the full Java language as an Abstract State Machine [BS99]. Jacobs el al. translate Java code directly into the PVS higher-order logic (as a shallow embedding) in order to conduct program verification [JHvB+98].

5.3 Design goals

For any formalization, it is important to aim at the following general design goals.

- readability (this is the basic requirement, as otherwise handling and applying the formalization would be severely hampered),
- faithfulness to the original language specification,
- succinctness and simplicity,
- maintainability and extendibility,
- adequacy for applications like theorem proving.

The reader is invited to keep them in mind while reading the subsequent sections and to judge herself how far we have reached them.

5.4 Abstract syntax of μJava

5.4.1 Programs

A key idea of this formalization is to separate declarations from code. Not only is their structure inherently different, they can also be described in isolation from each other. Since furthermore the structures of declarations for μJava and its μJVM are identical (from an abstract point of view), it is profitable to formalize the declarations in a generic style and supplying the actual method bodies (i.e., code) as a parameter.

As a μJava program consists of a series of class declarations, we model with a (parameterized) type that stands for a list of class declarations:

types $\alpha\ prog = \alpha\ cdecl\ list$

The parameter α is the type of method bodies. The list representation is not the most abstract one possible, since it retains the immaterial order of the class declarations. Yet its advantages are that it implies a finiteness constraint on the number of declarations and that the canonical conversion to a mapping yields a simple lookup mechanism.

A class declaration consists of the class name (of type *cname* which does not need to be specified further), the name of the superclass (for all classes except `Object`), and the lists of field and method declarations:

types $\alpha\ cdecl = cname \times cname\ option \times fdecl\ list \times \alpha\ mdecl\ list$

Field declarations simply give the field name and type,

types $fdecl = vname \times ty$

whereas method declarations give the method signature (consisting of the method name and the list of parameter types), the result type, and the method body, which is the parameter motivated above.

types $sig = mname \times ty\ list$
$\alpha\ mdecl = sig \times ty \times \alpha$

5.4.2 Values and types

The variety of values and their corresponding types in μJava is limited to the most important ones, basically Booleans, integers, and class references:

datatype *val* =
| Unit
| Bool *bool*
| Intg *int*
| Null
| Addr *loc*

datatype *ty* =
`void`
| `boolean`
| `int`
| NT
| Class *cname*

Here NT stands for the null type, i.e. the common type of all Null references. We have invented the type `void` (with the single dummy value Unit) for convenience in modeling the (non-existing) result of void methods.

Function default_val, defined by cases on *ty*, is used for variable initialization. Function typeof (of HOL type $(loc \to ty\ option) \to val \to ty\ option$), determining the dynamic type of a value, is used for the definition of conformance in §5.8.2, where the argument *dt* is used to dermine the existence and the (class) types of objects on the heap.

```
primrec default_val void      = Unit
        default_val boolean   = Bool False
        default_val int       = Intg 0
        default_val NT        = Null
        default_val (Class C) = Null
```

```
primrec typeof dt (Unit)    = Some void
        typeof dt (Bool b)  = Some boolean
        typeof dt (Intg i)  = Some int
        typeof dt (Null)    = Some NT
        typeof dt (Addr a)  = dt a
```

5.4.3 Looking up method and field declarations

The method lookup function

method :: α *prog* $\times$ *cname* $\to$ (*sig* $\rightsquigarrow$ *cname* $\times$ *ty* $\times$ α)

serves as a typical example of a definition by well-founded recursion. The intended result of method (Γ, C) *sg* is Some (D, T, b) where class D is the first class (traversing the subclass hierarchy upwards starting from C) that declares a method with signature *sg*, which has return type T and body b. Exploiting the fact that the inverse subclass relation (see §5.5.1) is well-founded, the following characteristic equation can be derived from the definition of method:

```
wf ((subcls1 Γ)⁻¹) ⟶
  method (Γ,C) = case map_of Γ C of None → empty
      | Some (sc,fs,ms) →
          (case sc of None → empty | Some D → method (Γ,D)) ⊕
          map_of (map (λ(s,m). (s,C,m)) ms)
```

A similar definition is used for

fields :: α *prog* $\times$ *cname* $\to$ ((*vname* $\times$ *cname*) $\times$ *ty*) *list*

The formalization given up to here is identical for μJava and our μJVM. Now comes the part of the abstract syntax that is μJava-specific.

5.4.4 μJava statements and expressions

The μJava statements are just the canonical ones for imperative languages, except that any expression may be used as statement and assignments are considered as expressions since they yield a result. Next to literals, local variables and type casts, the μJava expressions contain the actual object-oriented features, namely class creation, field access and assignment, and method call. We model the `this` expression by a special non-assignable local variable of that name. The Isabelle/HOL datatype definition of the abstract syntax intentionally looks pretty much like a BNF specification:

```
datatype  stmt = Skip
               | expr
               | stmt; stmt
               | if (expr) stmt else stmt
               | while (expr) stmt

and       expr = new cname
               | (ty)expr
               | val
               | vname
               | vname := expr
               | {cname}expr.vname
               | {cname}expr.vname := expr
               | expr.mname({ty list}expr list)
```

5.4.5 Program annotations

The parts in braces $\{\ldots\}$ in the above definition of expressions are called *type annotations*. Strictly speaking, they are not part of the source language but are normally computed by the compiler during type checking. Their purpose is to serve as auxiliary information that is crucial for the resolution of method overloading and the static binding of fields, with the following meaning:

Static overloading of methods: in the method call $obj.m(\{formal\text{-}param\text{-}tys\}params)$, *formal-param-tys* stands for the formal parameter types of the maximally specific method applicable w.r.t. the static types of *obj* and *params*. During execution, (*m*,*formal-param-tys*) is used for method lookup.

Static binding of fields: in $\{C\}obj.f$, C is the class declaring the field f w.r.t. the static type of *obj*. During execution, (*C*,*f*) is used for field selection.

In our language embedding, we do not distinguish between the source language and the augmented internal form since this would lead to a considerable amount of redundancy. Instead, we assume that the annotations are added beforehand (by some preprocessing step) and checked by the typing rules given in §5.5.3 below.

5.4.6 Method bodies

Finally, the definitions of statements and expressions being available, we can formalize μJava method body declarations. We define them as a tuple of parameter names, local variable declarations, a statement comprising the actual code block, and an expression to be returned—there is no return statement in μJava.

types $java\_mb = vname\ list \times (vname \times ty)\ list \times stmt \times expr$

5.5 Type system of μJava

The type system consists of the types already introduced, some basic relations between types (in particular, classes) and the actual typing rules for expressions and statements.

5.5.1 Type relations

Being an object-oriented language, μJava of course features the subclass relation, which is extracted w.r.t. a given program Γ:

- $\mathsf{subcls1}\ \Gamma \equiv \{(C,D)\ .\ \exists r.\ (C,\mathsf{Some}\ D,r) \in \mathsf{set}\ \Gamma\}$
- $\Gamma \vdash C \preceq_{\mathsf{c}} D \equiv (C,D) \in (\mathsf{subcls1}\ \Gamma)^*$

Built on the subclass relation we define the important *widening* relation, where $\Gamma \vdash S \preceq T$ means that in the context Γ value of type S may be used in a place where a value of type T is expected. We give this relation via an inductive definition:

inductive

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

5.5.2 *Typing judgments*

There are three forms of typing judgments, namely for well-typed statements, expressions, and expression lists:
$java\_mb\ env \vdash stmt\ \surd$
$java\_mb\ env \vdash expr :: ty$
$java\_mb\ env \vdash expr\ list\ [::]\ ty\ list$

The judgments include a context called *environment* that consists of the program and type bindings for local variables currently in scope:

types $\alpha\ env = \alpha\ prog \times (vname \rightsquigarrow ty)$

The actual type parameter for μJava environments is $java\_mb$.

5.5.3 *Typing rules*

The typing rules for most μJava terms are straightforward, thus we give only one typical example, namely the well-typedness of `while` loops:

$$\frac{E\vdash e :: \texttt{boolean} \qquad E\vdash s\ \surd}{E\vdash \texttt{while}\ (e)\ s\ \surd}$$

More interesting are the rules for field access and method call.

The field access rule enables static binding by calculating the annotation C with the help of the auxiliary function $\mathsf{field} \equiv \mathsf{map\_of} \circ (\mathsf{map}\ (\lambda((fn,fd),ft).\ (fn,(fd,ft)))) \circ \mathsf{fields}$ as follows:

$$\frac{E\vdash a :: \mathsf{Class}\ D \qquad \mathsf{field}\ (\mathsf{fst}\ E,D)\ fn = \mathsf{Some}\ (C,fT)}{E\vdash \{C\}a.fn :: fT}$$

In a similar fashion, method overloading is resolved in the method call rule. This involves the requirement that the max_spec auxiliary function (given below) yields a most specific applicable method.

$$\frac{E\vdash e :: \mathsf{Class}\ C \qquad E\vdash ps\ [::]\ pTs \qquad \mathsf{max\_spec}\ (\mathsf{fst}\ E)\ C\ (mn,pTs) = \{((\_,rT),fpTs)\}}{E\vdash e.mn(\{fpTs\}ps) :: rT}$$

5.5.4 *Maximally specific methods*

Following the involved definition in the Java specification, $\mathsf{max\_spec}\ \Gamma\ C\ sig$ determines the set of all maximally specific applicable methods of signature sig available for class C in program Γ: $\mathsf{max\_spec}\ \Gamma\ C\ sig \equiv$
$\{m.\ m \in \mathsf{appl\_methds}\ \Gamma\ C\ sig \wedge (\forall m' \in \mathsf{appl\_methds}\ \Gamma\ C\ sig.\ \mathsf{more\_spec}\ \Gamma\ m'\ m \longrightarrow m'=m)\}$

The partial order on methods used for max_spec is given by the relation more_spec Γ, which states that the defining classes as well as all parameter types are (pointwise) in widening relation:

$\mathsf{more\_spec}\ \Gamma\ ((d,\_),pTs)\ ((d',\_),pTs') \equiv \Gamma\vdash d \preceq d' \wedge \mathsf{list\_all2}\ (\lambda T\ T'.\ \Gamma\vdash T\preceq T')\ pTs\ pTs'$
$\mathsf{list\_all2}\ P\ xs\ ys \equiv \mathsf{length}\ xs = \mathsf{length}\ ys \wedge (\forall (x,y) \in \mathsf{zip}\ xs\ ys.\ P\ x\ y)$

The set of methods (available for class C in program Γ) applicable for signature (mn,pTs) are those with name mn and fitting parameter types:

$$\begin{aligned}\mathsf{appl\_methds}\ \Gamma\ C\ (mn,pTs) \equiv \{&((\mathsf{Class}\ D,rT),pTs').\\ &\mathsf{method}\ (\Gamma,C)\ (mn,pTs') = \mathsf{Some}\ (D,rT,\_) \wedge\\ &\mathsf{list\_all2}\ (\lambda T\ T'.\ \Gamma\vdash T\preceq T')\ pTs\ pTs'\}\end{aligned}$$

5.6 Well-formedness of both μJava and μJVM programs

Now we are equipped to express the well-formedness conditions on programs, which are part of the static checks performed by the compiler.

5.6.1 Programs

A program is well-formed iff it contains class `Object` (for simplicity, we model this class as if it were user-defined) and all class declarations are well-formed and no class is declared twice:

$$\begin{array}{l}\mathsf{wf\_prog}\ wf\_mb\ \Gamma \equiv \mathsf{ObjectDecl} \in \mathsf{set}\ \Gamma\ \wedge \\ \qquad\qquad (\forall c{\in}\mathsf{set}\ \Gamma.\ \mathsf{wf\_cdecl}\ wf\_mb\ \Gamma\ c)\ \wedge\ \mathsf{nodups}\ (\mathsf{map\ fst}\ \Gamma) \\ \mathsf{ObjectDecl} \equiv (\texttt{Object}, (\mathsf{None}, [], []))\end{array}$$

Note that this definition is parameterized with *wf_mb*, which is aimed to be the well-formedness predicate for method bodies (given in §5.6.3 below).

A class declaration (*C,sc,fs,ms*) is well-formed (in Γ) iff

- all types in the field declarations *fs* are legal,
- no field is declared twice,
- all method declarations in *ms* are well-formed,
- no method is declared twice,
- only class `Object` has no superclass,
- if C extends D then
  - D must be declared in Γ,
  - D must not be a subclass of C, and
  - if a method in *ms* overwrites one higher up, its return type must widen to the return type higher up.

Formally:

$$\begin{array}{l}\mathsf{wf\_cdecl}\ wf\_mb\ \Gamma\ (C,sc,fs,ms) \equiv \\ \quad (\forall(\_,T){\in}\mathsf{set}\ fs.\ \mathsf{is\_type}\ \Gamma\ T)\ \wedge \\ \quad \mathsf{nodups}\ (\mathsf{map\ fst}\ fs)\ \wedge \\ \quad (\forall m{\in}\mathsf{set}\ ms.\ \mathsf{wf\_mdecl}\ wf\_mb\ \Gamma\ C\ m)\ \wedge \\ \quad \mathsf{nodups}\ (\mathsf{map\ fst}\ ms)\ \wedge \\ \quad \mathsf{case}\ sc\ \mathsf{of\ None} \to C = \texttt{Object} \\ \qquad \mid \mathsf{Some}\ D \to D \in \mathsf{set}\ (\mathsf{map\ fst}\ \Gamma)\ \wedge \\ \qquad\qquad\qquad \neg\ \Gamma{\vdash}D \prec_{\mathsf{c}} C\ \wedge \\ \qquad\qquad\qquad \forall(sg,T,\_){\in}\mathsf{set}\ ms.\ \forall T'.\ \mathsf{method}(\Gamma,D)\ sg = \mathsf{Some}(\_,T',\_) \longrightarrow \Gamma{\vdash}T \preceq T'\end{array}$$

5.6.2 Methods (both μJava and μJVM)

The definition of well-formed classes relies on the well-formedness conditions for method declarations, which in turn rely on the well-formedness of method heads, the well-formedness parameter for method bodies, and the simple is_type predicate testing the existence of a class:

$$\begin{array}{l}\mathsf{wf\_mdecl}\ wf\_mb\ \Gamma\ C\ (sig,rT,b) \equiv \mathsf{wf\_mhead}\ \Gamma\ sig\ rT\ \wedge\ wf\_mb\ \Gamma\ C\ (sig,rT,b) \\ \mathsf{wf\_mhead}\ \Gamma\ (mn,pTs)\ rT \quad \equiv (\forall T{\in}\mathsf{set}\ pTs.\ \mathsf{is\_type}\ \Gamma\ T) \wedge \mathsf{is\_type}\ \Gamma\ rT \\ \mathsf{is\_type}\ \Gamma\ T \equiv \mathsf{case}\ T\ \mathsf{of\ Class}\ C \to C \in \mathsf{set}\ (\mathsf{map\ fst}\ \Gamma)\ \mid\ \_ \to \mathsf{True}\end{array}$$

5.6.3 Method bodies (μJava)

A method with signature (*mn*,*pTs*), return type *rT*, and body (*pns*,*lvars*,*blk*,*res*) is well-formed within class *C* in Γ iff

- there are as many parameter names *pns* as parameter types *pTs*,
- the names of parameters and local variables are unique and do not clash,
- all types of the local variables *lvars* are legal,
- in the context of Γ, the local variables and the current class *C*, the code block *blk* is well-typed and the type of the result expression *res* widens to *rT*.

Formally:

wf_java_mdecl Γ *C* ((*mn*,*pTs*),*rT*,(*pns*,*lvars*,*blk*,*res*)) ≡
  length *pns* = length *pTs* ∧
  nodups *pns* ∧ nodups (map fst *lvars*) ∧
  (∀*pn*∈set *pns*. map_of *lvars pn* = None) ∧
  (∀(*vn*,*T*)∈set *lvars*. is_type Γ *T*) ∧
  let *E* = (Γ,map_of *lvars*(*pns*[↦]*pTs*)(this↦Class *C*))
  in *E*⊢*blk* √ ∧ (∃*T*. *E*⊢*res*::*T* ∧ Γ⊢*T*≼*rT*)

Note that in contrast to Java, it is not required that local variables are initialized explicitly: the operational semantics does so implicitly upon method invocation.

Predicate wf_java_mdecl serves as actual parameter of wf_prog for μJava programs:

wf_java_prog Γ ≡ wf_prog wf_java_mdecl Γ

5.7 Operational semantics of μJava

For the semantics of μJava we use an operational description, as this style is the same as the original specification of Java [GJS96], easy to understand, and almost immediately executable.

5.7.1 Program state

A μJava state consists of the local variables and the heap, where the heap is a mapping from locations to objects, where objects consist of a class name and the instance variables.

types *obj* = *cname* × (*vname* ⇝ *val*)
  heap = *loc* ⇝ *obj*
  locals = *vname* ⇝ *val*
  state = *heap* × *locals*

An extended state is augmented with an optional exception. Here we need only a few of the language-defined exceptions of Java.

types *xstate* = *xcpt option* × *state*

datatype *xcpt* = `NullPointer`
  | `ClassCast`
  | `OutOfMemory`

Usually the meta-variables σ and *s* are of type *state* and *xstate* respectively.

5.7.2 Evaluation judgments

We deliberately chose an evaluation semantics as opposed to a transition semantics, since it is more abstract, less verbose and more convenient for proofs.

Analogously to typing judgments, there are three forms of evaluation judgments: statements transform an initial state to a final one, expressions additionally yield a value, and expression lists yield a list of values.

$$\begin{array}{ll} \mathit{java\_mb}\ \mathit{prog} \vdash \mathit{xstate}\ -\mathit{stmt}\rightarrow & \mathit{xstate} \\ \mathit{java\_mb}\ \mathit{prog} \vdash \mathit{xstate}\ -\mathit{expr}\succ\mathit{val}\rightarrow & \mathit{xstate} \\ \mathit{java\_mb}\ \mathit{prog} \vdash \mathit{xstate}\ -\mathit{expr\ list}[\succ]\mathit{val\ list}\rightarrow & \mathit{xstate} \end{array}$$

5.7.3 Evaluation rules and exception propagation

For each form of judgment, there is a general rule stating that exceptions propagate, e.g.

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

All other rules assume that the initial state is exception-free, but any further (intermediate) states may be exceptional, like s_1 and s_2 in the (otherwise trivial) composition rule:

$$\frac{\Gamma\vdash(\mathsf{None},\sigma)\ -c_1\rightarrow\ s_1 \quad \Gamma\vdash s_1\ -c_2\rightarrow\ s_2}{\Gamma\vdash(\mathsf{None},\sigma)\ -c_1;\ c_2\rightarrow\ s_2}$$

5.7.4 Method call rule

Most of the evaluation rules are more or less straightforward and are omitted here. The most complex rule is that for method calls:

$$\frac{\begin{array}{c}\Gamma\vdash(\mathsf{None},\sigma)\ -e\succ a\rightarrow\ s_1 \quad \Gamma\vdash s_1\ -ps[\succ]pvs\rightarrow\ (x,h,l) \\ dynT = \mathsf{fst}\ (\mathsf{the}\ (h\ (\mathsf{the\_Addr}\ a))) \\ (\_,\_,pns,lvars,blk,res) = \mathsf{the}\ (\mathsf{method}\ (\Gamma,dynT)\ (mn,pTs)) \\ \Gamma\vdash(\mathsf{np}\ a\ x,\ h,\ (\mathsf{init\_vars}\ lvars)(pns[\mapsto]pvs)(\mathtt{this}\mapsto a))\ -blk\rightarrow\ s_3 \\ \Gamma\vdash s_3\ -res\succ v\rightarrow\ (x_4,h_4,\_)\end{array}}{\Gamma\vdash(\mathsf{None},\sigma)\ -e.mn(\{pTs\}ps)\succ v\rightarrow\ (x_4,h_4,l)}$$

where

$$\begin{array}{l} \mathsf{the\_Addr}(\mathsf{Addr}\ l) = l \\ \mathsf{np}\ v\ x\ \equiv \mathsf{if}\ \ v = \mathsf{Null} \wedge x = \mathsf{None}\ \ \mathsf{then}\ \ \mathsf{Some}\ \mathtt{NullPointer}\ \ \mathsf{else}\ \ x \\ \mathsf{init\_vars} \equiv \mathsf{map\_of} \circ \mathsf{map}\ (\lambda(n,T).\ (n,\mathsf{default\_val}\ T)) \end{array}$$

Note that local variables are initialized with their default values, in contrast to Java, where the programmer must initialize them explicitly.

5.8 Type-safety

5.8.1 The notion of type-safety

A programming language is *type-safe* iff its type system prevents type mismatches, i.e. situations where dynamically produced values do not conform to the corresponding statically determined types. A language may be not type-safe if it has no type system at all (e.g., Smalltalk) or the type system is unsound (e.g., Eiffel). A consequence of type-safety for object-oriented languages is that method calls always find an applicable method. Note that type-safety (usually) does not cover division by zero, nontermination, etc.

5.8.2 Conformance of values to types

Central to the proof of type-safety is an invariant stating that all (run-time) values *conform* to their declared types, i.e. the types of the values are subtypes of the declared types. This notion is lifted pointwise from single values to maps of values and further to the whole state conforming to a type environment. The conformance judgments have the form

$$\begin{array}{lll} \alpha\ prog,heap\vdash & val & ::\preceq\ \ ty \\ \alpha\ prog,heap\vdash(\beta\rightsquigarrow\ val) & [::\preceq] & (\beta\rightsquigarrow\ ty) \\ \qquad\qquad state & ::\preceq\ java\_mb\ env & \end{array}$$

and the definitions

$$\begin{array}{ll} \mathsf{obj\_ty}\ o & \equiv \mathsf{case}\ o\ \mathsf{of}\ \mathsf{None}\rightarrow\mathsf{None}\mid\mathsf{Some}\ (C,fm)\rightarrow\mathsf{Some}\ C \\ \Gamma,h\vdash v::\preceq T & \equiv \exists S.\ \mathsf{typeof}\ (\mathsf{obj\_ty}\circ h)\ v=\mathsf{Some}\ S\wedge\Gamma\vdash S\preceq T \\ \Gamma,h\vdash vm[::\preceq]Tm & \equiv \forall n\ T.\ Tm\ n=\mathsf{Some}\ T\longrightarrow(\exists v.\ vm\ n=\mathsf{Some}\ v\wedge\Gamma,h\vdash v::\preceq T) \\ (h,l)::\preceq(\Gamma,lT) & \equiv (\forall a\ C\ fm.\ h\ a=\mathsf{Some}(C,fm)\longrightarrow\Gamma,h\vdash fm\ [::\preceq]\ \mathsf{map\_of}(\mathsf{fields}(\Gamma,C)))\wedge \\ & \quad \Gamma,h\vdash l\ [::\preceq]\ lT \end{array}$$

5.8.3 Proof of type-safety

A simple but important lemma for type-safety (not considering garbage collection) is the invariant that during execution any object once created is never lost and retains its type. This is formalized as the heap extension relation

$$h\trianglelefteq h'\equiv\forall a\ C\ fm.\ h\ a=\mathsf{Some}(C,fm)\longrightarrow(\exists fm'.\ h'\ a=\mathsf{Some}(C,fm'))$$

The invariant is proved by simultaneous rule induction, such that the proof goal is

$$\begin{array}{ll} (\Gamma\vdash(x,h,l)\ -\ c\ \ \ \ \ \ \ \ \rightarrow(x',h',l')\longrightarrow h\trianglelefteq h' & \wedge \\ (\Gamma\vdash(x,h,l)\ -\ e\succ v\rightarrow(x',h',l')\longrightarrow h\trianglelefteq h' & \wedge \\ (\Gamma\vdash(x,h,l)\ -es[\succ]vs\rightarrow(x',h',l')\longrightarrow h\trianglelefteq h' & \end{array}$$

The main theorem states that for well-formed programs, execution of well-typed statements and expression preserves conformance:

$$\begin{array}{l} \mathsf{wf\_java\_prog}\ \Gamma\longrightarrow \\ (\Gamma\vdash(x,\sigma)\ -c\rightarrow(x',\sigma')\longrightarrow \\ \quad \forall lT.\ \sigma::\preceq(\Gamma,lT)\longrightarrow(\Gamma,lT)\vdash c\surd\longrightarrow \\ \qquad \sigma'::\preceq(\Gamma,lT))\quad\wedge \\ (\Gamma\vdash(x,\sigma)\ -e\succ v\rightarrow(x',\sigma')\longrightarrow \\ \quad \forall lT.\ \sigma::\preceq(\Gamma,lT)\longrightarrow\forall T.\ (\Gamma,lT)\vdash e::T\longrightarrow \\ \qquad \sigma'::\preceq(\Gamma,lT)\wedge(x'=\mathsf{None}\longrightarrow\Gamma,\mathsf{fst}\ \sigma'\vdash v::\preceq T))\quad\wedge \\ (\Gamma\vdash(x,\sigma)\ -es[\succ]vs\rightarrow(x',\sigma')\longrightarrow \\ \quad \forall lT.\ \sigma::\preceq(\Gamma,lT)\longrightarrow\forall Ts.\ (\Gamma,lT)\vdash es[::]Ts\longrightarrow \\ \qquad \sigma'::\preceq(\Gamma,lT)\wedge(x'=\mathsf{None}\longrightarrow\mathsf{list\_all2}\ (\lambda v\ T.\ G,\mathsf{fst}\ \sigma'\vdash v::\preceq T)\ vs\ Ts)) \end{array}$$

This theorem is proved again by simultaneous rule induction and requires a large amount of auxiliary lemmas. The part concerning statements may be re-phrased in a more readable form as

$$\mathsf{wf\_java\_prog}\ \Gamma\ \wedge\ E=(\Gamma,lT)\ \wedge\ E\vdash c\surd\ \wedge\ \Gamma\vdash(\_,\sigma)\ -c\rightarrow(\_,\sigma')\ \wedge\ \sigma::\preceq E\longrightarrow\sigma'::\preceq E$$

An almost direct consequence is that "method not understood" run-time errors cannot occur.

6.1 The μJava Virtual Machine

The source language Java comes with an abstract machine known as the *Java Virtual Machine* (*JVM*) [LY96]. Its instruction set is a high-level assembly language specifically tailored for Java. It is meant to be the standard target language for Java compilers. In this subsection we present the μJVM, an abstract version of the JVM geared towards μJava. The main simplification in our approach is the use of μJava program skeletons for holding μJVM instruction sequences. This is the reason why type *prog* in §5.4.1 above is defined in a generic way and is parameterized by the actual method code. As a result, we can avoid the specialized file formats ("class files") described in the JVM specification [LY96]. Their main purpose is to record the type information present in the source programs.

6.1.1 Related work

There are a number of operational definitions of variants of the JVM. Our model is based on the work of Pusch [Pus99] which in turn is based on the work of Qian [Qia99]. Hartel *et al.* [HBL99] independently arrive at a similar formalization. A description based on Abstract State Machines is given by Börger and Schulte [BS98].

6.1.2 μJVM instructions

The μJVM is a stack machine. For each method invocation there is an operand stack for expression evaluation and a list of local variables (which includes the parameters). The formal model is described in §6.1.3.

Following the JVM specification [LY96], we have structured the instructions into several groups of related instructions, describing each by its own execution function. This makes the operational semantics easier to understand, since every function only works on the parameters that are needed for the corresponding group of instructions:

datatype *instr* = LS *load_and_store*
| CO *create_object*
| MO *manipulate_object*
| CH *check_object*
| MI *meth_inv*
| MR *meth_ret*
| OS *op_stack*
| BR *branch*

We will now discuss these groups briefly. Instruction names mostly follow the JVM nomenclature.

- **datatype** *load_and_store* = Load *nat* | Store *nat* | Bipush *int* | Aconst_null
  Load n pushes the contents of local variable n on the stack. Store n pops the top from the stack and puts it into local variable n. Bipush i pushes integer i on the stack. Aconst_null pushes the null reference on the stack.

- **datatype** *create_object* = New *cname*
  New C creates a new object of class C and initializes its fields according with their default values.

- **datatype** *manipulate_object* = Getfield *vname cname* | Putfield *vname cname*
  Getfield x C uses the top of the stack as the address of an object of class C and replaces that address with the value of field x of the object; Putfield x C expects both a value and an address on the stack, puts the value into field x of the object and pops both value and address from the stack.

- **datatype** *check_object* = Checkcast *cname*
  Checkcast C checks if the top of the stack is a reference to an object of class C (or a subclass of C) and throws an exception if not.

- **datatype** *meth_inv* = Invoke *mname* (*ty list*)
  Invoke *mn ts* interprets the top of the stack as a reference to an object and calls the method determined by the class of that object and the signature (*mn*,*ts*).

- **datatype** *meth_ret* = Return
  Return returns from a method invocation. The result value is the top of the stack.

- **datatype** *op_stack* = Pop | Dup | Swap
  The instructions pop, duplicate and swap the top of the stack.

- **datatype** *branch* = Goto *int* | Ifcmpeq *int*
  Goto is unconditional, whereas Ifcmpeq compares and pops the two top elements of the stack and only performs the jump if they are equal. The integer argument is added to the current program counter.

This is just a representative sample of instructions which is sufficient to serve as a target language for μJava. Because some features are missing (e.g. arrays and interfaces) and because we have restricted the instruction set to the essentials, some groups have become singletons. Type conversion instructions have disappeared altogether because we have only one numeric type, *int*. Arithmetic is missing (as in μJava) but would be trivial to include. There are also some subtle differences to the corresponding JVM instructions:

- A number of instructions such as Load, Store and Return are overloaded, i.e. they work for all types. This streamlines the instruction set without, as we shall see, compromising type-safety.

- Invoke does not carry the name of the class defining the invoked method. This complicates the proof of type-safety, as discussed in §6.3.

6.1.3 Operational semantics of μJVM instructions

As discussed above, a μJVM program is simply a μJava program where the method bodies consist of instruction sequences. Thus we instantiate our program skeleton type *prog* once more:

types *bytecode* = *instr list*
jvm_prog = (*nat* × *bytecode*) *prog*

The natural number in the method body simply says how many local variable this method has (which could be computed by scanning the bytecode). It is used when allocating stack space upon method invocation. The JVM also records the maximum stack size, which we have chosen to leave open.

The μJVM state is formalized as a triple consisting of an optional exception, the heap, and the frame stack. This is just like in μJava's *xstate*, except that the local variables are replaced by the frame stack. For each active method invocation, there exists a frame containing its own operand stack, a list of local variables, the name of the current class, the signature of the current method, and the program counter:

types *jvm_state* = *xcpt option* × *heap* × *frame list*
frame = *opstack* × *locvars* × *cname* × *sig* × *nat*
opstack = *val list*
locvars = *val list*

Note that *xcpt*, *val* and *heap* are inherited from μJava. The local variables are laid out in the format [*this,p1,..,pm,l1,..,ln*], where *this* is the value of `this`, *p1* through *pm* are the parameters of the method, and *l1* through *ln* are the remaining local variables.

Execution of a μJVM instruction transforms the machine state. A raised exception or an empty frame stack means the μJVM is in a final state—remember that there is no exception handling. If the machine has not yet reached a final state, function exec performs a single execution step: it calls an appropriate execution function (e.g. exec_mo) and incorporates the result in the new machine state. To model the distinction between final and non-final states, the result type of exec is *jvm_state option*: None is returned if there is no successor state.

exec :: *jvm_prog* × *jvm_state* → *jvm_state option*

```
exec(Γ, (None, hp, (stk,loc,C,sig,pc)#frs)) =
  Some (case snd(snd(snd(the(method (Γ,C) sig)))) ! pc of
          MO ins →
           let (xp',hp',stk',pc') = exec_mo ins hp stk pc
           in (xp',hp',(stk',loc,C,sig,pc')#frs)
        | ... → ...)
exec(Γ, (xp, hp, [])) = None
exec(Γ, (Some xp, hp, frs)) = None
```

For example, the operational semantics of Getfield looks like this:

```
exec_mo :: manipulate_object → heap → opstack → nat →
             (xcpt option × heap × opstack × nat)

exec_mo (Getfield F C) hp stk pc =
  let oref   = hd stk;
      xp'    = if oref=Null then Some NullPointer else None;
      (oc,fs) = the(hp(the_Addr oref));
      stk'   = if xp'=None then the(fs(F,C))#(tl stk) else tl stk
  in (xp', hp, stk', pc+1)
```

F is a field name and *C* the defining class of the field. The top of the operand stack *stk* should contain a reference to a class instance stored on the heap *hp*. In case of a null reference an exception is thrown. Otherwise, the fields *fs* are extracted from the referenced object. The content of the field determined by (*F,C*) is pushed on the operand stack. Finally, the program counter *pc* is incremented.

But what if the stack is empty upon execution of Getfield? The JVM specification describes the operational semantics for each instruction in the context of a JVM state where several constraints hold, e.g. there must be an appropriate number of arguments on the operand stack, or the operands must be of a certain type. If the constraints are not satisfied, the behavior of the JVM is undefined. In our approach, we formalize the behavior of μJVM instructions with total functions. If a state does not satisfy the constraints of the current instruction, the result is defined but we don't know what it is. For example, the top of an empty operand stack is hd [], but the definition of hd (which we have not shown) only says that hd [] is some arbitrary but fixed value.

Finally, execution of the entire code consists of repeated application of exec as long as the result is not None. The relation $\Gamma \vdash s \xrightarrow{jvm} t$ should be read as "In the context of a μJVM program Γ, the execution starting with state s leads to t in finitely many steps". Its definition uses the reflexive transitive closure of successful execution steps:

$$\Gamma \vdash s \xrightarrow{jvm} t \;\equiv\; (s,t) \in \{(s,s').\ \mathsf{exec}(\Gamma,s) = \mathsf{Some}\ s'\}^*$$

6.2 The Bytecode Verifier

An essential part of the JVM is the *bytecode verifier* that statically checks several safety-relevant constraints before execution of the code. One main aspect of the bytecode verifier is to statically derive the types of all runtime data and check that all instructions will receive arguments of the correct type. Hence the bytecode verifier can be seen as a type checker (more precisely: a type reconstructor) for the JVM. Therefore it has become customary to separate the bytecode verifier into a specification in terms of a type system and an implementation as a data flow analyzer. Thus the correctness argument for bytecode verification is split in two parts: a type safety proof relating the type system and the operational semantics, and an implementation proof relating the type system and the data flow analyzer.

6.2.1 Related work

Our approach for the proof of type-safety builds on the work of Qian [Qia99] who covers a considerably larger subset of the JVM. Closely related is the work by Stata and Abadi [SA98], who treat subroutines, and the work by Freund and Mitchell [FM98], who treat object initialization. The correctness of the data flow analyzer is analyzed by Goldberg [Gol98] and, more abstractly, Nipkow [Nip99b]. An unorthodox approach to bytecode verification via model checking is reported by Basin *et al.* [BFPV99].

6.2.2 Types and type relations

The static types for the μJVM are the same as for μJava, i.e. of type *ty*. Thus we can reuse a number of concepts from the μJava level. The main distinction is that the JVM allows the type of local variables to change during execution. If at a particular instruction a local variable may hold values from either of two incompatible types (because two execution paths lead to this instruction), this local variable will have type "unusable" at this point. In our formalization we work with the HOL type *ty option*, which is either None, representing the unusable type, or Some T, representing type T. We call these types *static types* because they are the result of a static analysis of the program.

The subtype relation is lifted from types to static types as follows: any static type is a subtype of None (because None represents the set of all types), and otherwise the subtype relation on *ty* is simply lifted:

$$\begin{array}{l}
_ \vdash _ \preceq_o _ :: \mathit{jvm_prog} \to \mathit{ty\ option} \to \mathit{ty\ option} \to \mathit{bool} \\
(\Gamma \vdash _ \quad\ \preceq_o \mathsf{None}) \quad\ = \mathsf{True} \\
(\Gamma \vdash \mathsf{None} \ \preceq_o \mathsf{Some}\ T) \ = \mathsf{False} \\
(\Gamma \vdash \mathsf{Some}\ T \preceq_o \mathsf{Some}\ T') \ = (\Gamma \vdash T \preceq T')
\end{array}$$

A *state type* contains type information for all local variables and the operand stack at a certain program point. The local variables may contain unusable values, whereas on the operand stack only usable values may be stored.

$$\begin{aligned} &\mathit{locvars\_type} &&= \mathit{ty\ option\ list} \\ &\mathit{opstack\_type} &&= \mathit{ty\ list} \\ &\mathit{state\_type} &&= \mathit{opstack\_type} \times \mathit{locvars\_type} \end{aligned}$$

We extend the predicate $\preceq_o$ in two steps to state types:

$$\begin{aligned} &\_ \vdash \_ \preceq_l \_ :: \mathit{jvm\_prog} \to \mathit{locvars\_type} \to \mathit{locvars\_type} \to \mathit{bool} \\ &\Gamma \vdash LT \preceq_l LT' \equiv \mathsf{length}\ LT = \mathsf{length}\ LT' \wedge (\forall (t,t') \in \mathsf{set}\ (\mathsf{zip}\ LT\ LT').\ \Gamma \vdash t \preceq_o t') \end{aligned}$$

$$\begin{aligned} &\_ \vdash \_ \preceq_s \_ :: \mathit{jvm\_prog} \to \mathit{state\_type} \to \mathit{state\_type} \to \mathit{bool} \\ &\Gamma \vdash s \preceq_s s' \equiv \Gamma \vdash \mathsf{map\ Some}\ (\mathsf{fst}\ s) \preceq_l \mathsf{map\ Some}\ (\mathsf{fst}\ s') \wedge \Gamma \vdash \mathsf{snd}\ s \preceq_l \mathsf{snd}\ s' \end{aligned}$$

Type information for the entire code of a method is collected in a value of *method type*. A value of *class type* maps a method signature to a value of method type, and a value of *program type* maps a class name to a value of class type:

$$\begin{aligned} &\mathit{method\_type} &&= \mathit{state\_type\ list} \\ &\mathit{class\_type} &&= \mathit{sig} \to \mathit{method\_type} \\ &\mathit{prog\_type} &&= \mathit{cname} \to \mathit{class\_type} \end{aligned}$$

6.2.3 Static well-typedness

Given some instruction sequence, the bytecode verifier has to infer type information for each instruction, i.e. a method type, such that the whole sequence is well-typed. We concentrate on what well-typedness means and ignore the computation of the method type. Let us start by looking at an example of a well-typed instruction sequence.

instruction	stack	local variables
Load 0	[]	[Some(Class B), Some(int)]
Dup	[Class A]	[Some(Class B), None]
Store 1	[Class A, Class A]	[Some(Class B), None]
Invoke m []	[Class A]	[Some(Class B), Some(Class A)]
Goto -3	[Class A]	[Some(Class B), Some(Class A)]

On the left the instructions are shown and on the right the type of the stack elements and the local variables (*opstack_type* and *locvars_type* above). The type information attached to an instruction characterizes the state *before* execution of that instruction. We assume that class B is a subclass of A and that A defines a method m which takes no arguments and returns a value of type A.

Execution starts with an empty stack and the two local variables hold a reference to an object of class B and an integer. The first instruction loads local variable 0, a reference to a B object, on the stack. The type information associated with following instruction may puzzle at first sight: it says that a reference to an A object is on the stack, and that local variable 1 has become unusable. This means the type information has become less precise but is still correct: a B object is also an A object, and an integer is now classified as unusable. Formally: Class $B \preceq$ Class A and Some(int) $\preceq_o$ None. The reason for these more general types is that the predecessor of the Dup instruction may have either been Load 0 or Goto -3. Since there exist different execution paths to reach Dup, the type information of the two paths has to be "merged". The type of the second local variable is either *int* or Class A, which are incompatible, i.e. the only common supertype is None.

Apart from this complication, the type of the stack elements and the local variables changes as expected. It now remains to specify formally when an instruction sequence is well-typed w.r.t. a method type.

We start by defining a predicate that checks whether an instruction at a certain program point is well-typed with respect to a given method type. Additionally, it checks several other constraints, e.g. an index to a local variable must not be greater than the number of local variables and the program counter must remain within the current method. These constraints are indispensable to carry out the soundness proof for the bytecode verifier. The type-checking predicate makes a case distinction over the instruction to be executed at the current program point. We only show the Load case:

```
wt_instr :: instr → jvm_prog → ty → method_type → nat → nat → bool
wt_instr (LS(Load i)) Γ _ sts max_pc pc =
  let (ST,LT) = sts!pc
  in    pc+1 < max_pc  ∧ i < length LT  ∧
        (∃T. LT!i = Some T  ∧ Γ ⊢ (T#ST , LT) ≼_s sts!(pc+1))
```

The above predicate enforces that there is a next instruction ($pc{+}1 < max\_pc$), that the local variable to be loaded exists ($i <$ length LT) and is usable ($LT!i =$ Some T), and that the new state type ($T\#ST$, LT) is a subtype of the state type associated with the successor instruction. We cannot ask for equality here because that next instruction can have other predecessors as well.

The well-typedness checks for all instructions except Return follow the above schema: there are some instruction-specific conditions and the general requirement that for each successor instruction pc' and corresponding successor state st' both $pc' < max\_pc$ and $\Gamma \vdash st' \preceq_s sts!pc'$ hold.

Step by step, we now extend the notion of well-typedness to methods, classes, and programs. At the start of the execution of a method, the operand stack must be empty, and the local variables must contain values according to the type of the current class C (local variable 0 holds `this`), the parameter types pTs of the current method, and None for all other local variables, because the μJVM (as the JVM) does not initialize the other local variables automatically (replicate i x produces a list of i copies of x):

```
wt_start :: jvm_prog → cname → ty list → nat → method_type → bool
wt_start Γ C pTs mxl Δ ≡
   Γ ⊢ ([], Some(Class C)#(map Some pTs)@(replicate mxl None)) ≼_s Δ!0
```

The body of a method must be non-empty. A method is well-typed with respect to a method type Δ, if it is well-typed at the beginning of the method body, and if for every program point in the method body the instruction is well-typed:

```
wt_method :: jvm_prog → cname → ty list → ty → nat → instr list → method_type → bool
wt_method Γ C pTs rT mxl ins Δ ≡
      let max_pc = length ins
      in 0 < max_pc ∧ wt_start Γ C pTs mxl Δ ∧
         (∀pc<max_pc. wt_instr (ins!pc) Γ rT Δ max_pc pc)
```

As *jvm_prog* is an instance of the parameterized type *prog* of program skeletons, well-typedness of μJVM programs is defined with the help of the parameterized well-formedness predicate wf_prog for program skeletons (see §5.6.1). We simply feed a suitably instantiated version of wt_method to wf_prog:

```
wt_jvm_prog :: jvm_prog → prog_type → bool
wt_jvm_prog Γ Φ ≡
      wf_prog (λΓ C (sig,rT,maxl,b). wt_method Γ C (snd sig) rT maxl b (Φ C sig)) Γ
```

This is in complete analogy with the definition of wf_java_prog in §5.6.3.

A bytecode verifier (or more abstractly: a type system) statically determines the types of all runtime data. A type system is sound if the statically predicted type gives a correct approximation of the runtime value produced during execution. Thus we first need to define this approximation relationship for the various runtime data and types in the μJVM. This means lifting the relation $::\preceq$ already defined at the μJava level (see §5.8.2) to complex data structures containing values and types.

The unusable type None approximates any value; a static type Some T approximates value v if v conforms to T:

```
approx_val :: jvm_prog → heap → val → ty option → bool
approx_val Γ h v  None      = True
approx_val Γ h v (Some T)  = (Γ,h⊢v::≼T)
```

This relation is lifted pointwise to local variables and the operand stack:

```
approx_loc :: jvm_prog → heap→ val list → locvars_type → bool
approx_loc Γ hp loc LT ≡ length loc = length LT ∧
                         (∀(val,t)∈set(zip loc LT). approx_val Γ hp val t)

approx_stk :: jvm_prog → heap → opstack → opstack_type → bool
approx_stk Γ hp stk ST ≡ approx_loc Γ hp stk (map Some ST)
```

The proof of type-safety follows the same pattern as the one for μJava: we show that if we start in a correct state and execute an instruction of a well-typed program, we again end up in a correct state. A correct state is one where the program type approximates all the runtime data in the state. Unfortunately, this notion of a correct state is too weak to make the above implication true—a typical example of an invariant that needs strengthening. One has to take into account that the frames on the frame stack are not unrelated but look roughly like this: the top element is the currently executing frame, which can be at any program point, but the remaining frames are all snapshots taken directly after an Invoke instruction. If we do not take this property into account, it becomes impossible to show that the Return instruction preserves correctness of states. Because of this difference between the top frame and the rest, correctness of a state treats the top frame separately:

```
correct_state :: jvm_prog → prog_type → jvm_state → bool
correct_state Γ Φ (None,hp,f#fs) = Γ ⊢h hp√ ∧ (let (stk,loc,C,sig,pc) = f
      in ∃rT maxl ins. method (Γ,C) sig = Some(C,rT,(maxl,ins)) ∧
         correct_frame Γ hp ((Φ C sig)!pc) maxl ins f ∧ correct_frames Γ hp Φ rT sig fs)
correct_state Γ Φ (None   ,hp,[]) = True
correct_state Γ Φ (Some x,hp,fs) = True
```

Correctness of a single frame requires the stack and heap data to be approximated by the corresponding state type. Additionally, the program counter should point at a valid instruction and the actual number of local variables should be as described by the method definition:

```
correct_frame :: jvm_prog → heap → state_type → nat → bytecode → frame → bool
correct_frame Γ hp (ST,LT) maxl ins (stk,loc,C,sig,pc) ≡
      approx_stk Γ hp stk ST  ∧ approx_loc Γ hp loc LT ∧
      pc < length ins ∧ length loc = length (snd sig) + maxl + 1
```

Predicate correct_frames, which checks the correctness of a stack of pending frames, takes as an argument the return type and the signature of the method executing in the frame directly above, in order to check that it matches the information in the pending frame below. This check is performed recursively through the frame stack:

```
correct_frames :: jvm_prog → heap → prog_type → ty → sig → frame list → bool
correct_frames Γ hp Φ rT0 sig0 [] = True
correct_frames Γ hp Φ rT0 sig0 (f#frs) =
  let (stk,loc,C,sig,pc) = f;
      (ST,LT) = (Φ C sig) ! pc
  in
  (∃rT maxl ins. method (Γ,C) sig = Some(C,rT,(maxl,ins)) ∧
  (∃mn pTs k. pc = k+1 ∧ ins!k = MI(Invoke mn pTs) ∧ (mn,pTs) = sig0 ∧
  (∃apTs D ST'. fst((Φ C sig)!k) = (rev apTs) @ (Class D) # ST' ∧
  length apTs = length pTs ∧
  (∃D' rT' maxl' ins'. method (Γ,D) sig0 = Some(D',rT',(maxl',ins')) ∧ Γ ⊢ rT0 ≼ rT') ∧
  correct_frame Γ hp (tl ST, LT) maxl ins f ∧
  correct_frames Γ hp Φ rT sig frs)))
```

This predicate needs some explanation. We discuss its body step by step. It requires that:

- a method with signature *sig* is defined in class C;
- the instruction to be executed is preceded by an Invoke instruction which invokes a method with signature *sig0* (from the frame above);
- the *opstack_type*-part of the state type for the Invoke instruction consists of a list of actual argument types $apTs$ in reverse order, followed by a class D and some remaining stack type;
- the length of the actual and the formal parameter lists agree;
- a method with signature *sig0* (executing in the frame above) is defined in some class D' above D with a return type that is a supertype of the return type of the actual method executing in the frame above;
- the frame itself is correct, as are the frames below. The current frame is checked against tl ST because its operand stack does not yet contain the return value of the method it invoked.

This is more complicated than the corresponding predicate in [Pus99] because the Invoke instruction does not carry around the defining class of the method any more. This loss of static information (wt_instr) requires the invariant (correct_frames) to be strengthened.

Now we can prove the following main type-safety theorem, the preservation of the invariant correct_state:

```
wt_jvm_prog Γ Φ ∧ correct_state Γ Φ s ∧ exec(Γ,s) = Some s' ⟶ correct_state Γ Φ s'
```

It is shown by a case distinction over the instructions. Most instructions are routine and many are proved automatically (once the necessary lemmas have been identified!). Invoke, however, requires fairly subtle reasoning involving key properties of the type system. The final corollary follows easily:

```
wt_jvm_prog Γ Φ ∧ correct_state Γ Φ s ∧ Γ ⊢ s -jvm→ t ⟶ correct_state Γ Φ t
```

7 Conclusion

We have given the reader a guided tour of a formal definition of μJava and the μJVM, their type safety proofs and the supporting theorem proving technology. Although μJava is a very impoverished version of Java, it should have convinced the reader of the main claim, that theorem provers are suitable tools for the analysis of programming language semantics. Further aspects of Java (a compiler and a Hoare logic—see [Ohe99] for a preliminary report) have been formalized as well, thus making μJava a non-trivial example of a formalized programming language. We believe that any serious programming language deserves such formal definition and analysis.

References

[ABI+96] Peter B. Andrews, Matthew Bishop, Sunil Issar, Dan Nesmith, Frank Pfenning, and Hongwei Xi. TPS: A theorem proving system for classical type theory. *J. Automated Reasoning*, 16:321–353, 1996.

[And86] Peter Andrews. *An Introduction to Mathematical Logic and Type Theory: to Truth through Proof.* Computer Science and Applied Mathematics. Academic Press, 1986.

[APP94] F. Andersen, K.D. Petersen, and J.S. Pettersson. Program verification using HOL-UNITY. In J. Joyce and C. Seger, editors, *Higher Order Logic Theorem Proving and Its Applications*, volume 780 of *Lect. Notes in Comp. Sci.*, pages 1–15. Springer-Verlag, 1994.

[BBC+97] B. Barras, S. Boutin, C. Cornes, J. Courant, J.C. Filliatre, E. Giménez, H. Herbelin, G. Huet, C. Muñoz, C. Murthy, C. Parent, C. Paulin, A. Saïbi, and B. Werner. The Coq Proof Assistant Reference Manual — Version V6.1. Technical Report 0203, INRIA, August 1997.

[BCD+88] Patrick Borras, Dominique Clément, Thierry Despeyroux, Janet Incerpi, Gilles Kahn, Bernard Lang, and Valérie Pascual. Centaur: the system. In *Third Symposium on Software Development Environments*, 1988. (Also appears as INRIA Report no. 777).

[BF95] Yves Bertot and Ranan Fraer. Reasoning with executable specifications. In *TAPSOFT '95: Theory and Practice of Software Development*, volume 915 of *Lect. Notes in Comp. Sci.*, pages 531–545. Springer-Verlag, 1995.

[BFPV99] David Basin, Stefan Friedrich, Joachim Posegga, and Harald Vogt. Java bytecode verification by model checking. System abstract. In N. Halbwachs and D. Peled, editors, *Computer Aided Verification (CAV'99)*, volume 1633 of *Lect. Notes in Comp. Sci.*, pages 491–494. Springer-Verlag, 1999.

[BGG+92] Richard Boulton, Andrew Gordon, Mike Gordon, John Harrison, John Herbert, and John Van Tassel. Experience with embedding hardware description languages in HOL. In V. Stavridou, T.F. Melham, and R.T. Boute, editors, *Theorem Provers in Circuit Design*, pages 129–156. North-Holland, 1992.

[BS98] Egon Börger and Wolfram Schulte. Defining the Java Virtual Machine as platform for provably correct Java compilation. In L. Brim, J. Gruska, and J. Zlatuska, editors, *Mathematical Foundations of Computer Science (MFCS'98)*, volume 1450 of *Lect. Notes in Comp. Sci.*, pages 17–35. Springer-Verlag, 1998.

[BS99] Egon Börger and Wolfram Schulte. A programmer friendly modular definition of the semantics of Java. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lect. Notes in Comp. Sci.*, pages 353–404. Springer-Verlag, 1999.

[C+86] Robert L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, 1986.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *J. Symbolic Logic*, 5:56–68, 1940.

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

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

[DE99] Sophia Drossopoulou and Susan Eisenbach. Describing the semantics of Java and proving type soundness. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lect. Notes in Comp. Sci.*, pages 41–82. Springer-Verlag, 1999.

[FHR99] Amy Felty, Douglas Howe, and Abhik Roychoudhury. Formal metatheory using implicit syntax, and an application to data abstraction for asynchronous systems. In H. Ganzinger, editor, *Automated Deduction — CADE-16*, volume 1632 of *Lect. Notes in Comp. Sci.*, pages 237–251. Springer-Verlag, 1999.

[FM98] Stephen N. Freund and John C. Mitchell. A type system for object initialization in the Java bytecode language. In *ACM Conf. Object-Oriented Programming: Systems, Languages and Applications*, 1998.

[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.

[Gol98] Allen Goldberg. A specification of Java loading and bytecode verification. In *Proc. 5th ACM Conf. Computer and Communications Security*, 1998.

[Gor85] M.C.J. Gordon. HOL — a machine oriented formulation of higher-order logic. Technical Report 68, University of Cambridge, Computer Laboratory, 1985.

[Gor89] M.C.J. Gordon. Mechanizing programming logics in higher order logic. In G. Birtwistle and P.A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving.* Springer-Verlag, 1989.

[HBL99] Pieter Hartel, Michael Butler, and Moshe Levy. The operational semantics of a Java secure processor. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lect. Notes in Comp. Sci.*, pages 313–352. Springer-Verlag, 1999.

[HC96] Barbara Heyd and Pierre Crégut. A modular coding of Unity in Coq. 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 251–266. Springer-Verlag, 1996.

[JHvB+98] B. Jacobs, M. Huisman, M. van Berkum, U. Hensel, and H. Tews. Reasoning about Java classes (preliminary report). In *ACM Conf. Object-Oriented Programming: Systems, Languages and Applications*, pages 329–340, 1998.

[JS94] J. Joyce and C. Seger, editors. *Higher Order Logic Theorem Proving and Its Applications*, volume 780 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 1994.

[Kle98] Thomas Kleymann. *Hoare Logic and VDM: Machine-Checked Soundness and Completeness Proofs.* PhD thesis, Department of Computer Science, University of Edinburgh, 1998. Report ECS-LFCS-98-392.

[Kli93] Paul Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2(2):176–201, 1993.

[LY96] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, 1996.

[MG94] Savitri Maharaj and Elsa Gunter. Studying the ml module system in hol. In T.F. Melham and J. Camilleri, editors, *Higher Order Logic Theorem Proving and its Applications*, volume 859 of *Lect. Notes in Comp. Sci.*, pages 269–284. Springer-Verlag, 1994.

[Nes94] Monica Nesi. Value-passing CCS in HOL. In J. Joyce and C. Seger, editors, *Higher Order Logic Theorem Proving and Its Applications*, volume 780 of *Lect. Notes in Comp. Sci.*, pages 352–365. Springer-Verlag, 1994.

[Nip98] Tobias Nipkow. Winskel is (almost) right: Towards a mechanized semantics textbook. *Formal Aspects of Computing*, 10:171–186, 1998.

[Nip99a] Tobias Nipkow. *Isabelle/HOL. The Tutorial*, 1999. Unpublished Manuscript. Available at `http://isabelle.in.tum.de/doc/tutorial.pdf`.

[Nip99b] Tobias Nipkow. Towards verified bytecode verifiers. Submitted for publication, 1999.

[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.

[Ohe99] David von Oheimb. Hoare logic for mutual recursion and local variables. In C. Pandu Rangan, V. Raman, and R. Ramanujam, editors, *Foundations of Software Technology and Theoretical Computer Science (FST&TCS)*, volume 1738 of *Lect. Notes in Comp. Sci.*, pages 168–180. Springer-Verlag, 1999.

[ON99] David von Oheimb and Tobias Nipkow. Machine-checking the Java specification: Proving type-safety. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lect. Notes in Comp. Sci.*, pages 119–156. Springer-Verlag, 1999.

[Pau87] Lawrence C. Paulson. *Logic and Computation*. Cambridge University Press, 1987.

[Pau94] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 1994. Isabelle home page: `http://isabelle.in.tum.de/`.

[Pau99] Lawrence C. Paulson. Mechanizing UNITY in Isabelle. Technical Report 467, University of Cambridge, Computer Laboratory, May 1999. To appear in ACM Trans. Computational Logic.

[Pfe91] Frank Pfenning. Logic programming in the LF Logical Framework. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 66–78. Cambridge University Press, 1991.

[Pol94] Robert Pollack. *The Theory of LEGO: A Proof Checker for the Extended Calculus of Constructions*. PhD thesis, University of Edinburgh, 1994.

[Pra95] Wishnu Prasetya. *Mechanically Supported Design of Self-stabilising Algorithms*. PhD thesis, University of Utrecht, 1995.

[PS99] Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Automated Deduction — CADE-16*, volume 1632 of *Lect. Notes in Comp. Sci.*, pages 202–206. Springer-Verlag, 1999.

[Pus99] Cornelia Pusch. Proving the soundness of a Java bytecode verifier specification in Isabelle/HOL. In W.R. Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99)*, volume 1579 of *Lect. Notes in Comp. Sci.*, pages 89–103. Springer-Verlag, 1999.

[Qia99] Zhenyu Qian. A formal specification of Java Virtual Machine instructions for objects, methods and subroutines. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lect. Notes in Comp. Sci.*, pages 271–311. Springer-Verlag, 1999.

[SA98] Raymie Stata and Martín Abadi. A type system for Java bytecode subroutines. In *Proc. 25th ACM Symp. Principles of Programming Languages*, pages 149–161. ACM Press, 1998.

[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.

[Sli97] Konrad Slind. Derivation and use of induction schemes in higher-order logic. In E. Gunter and A. Felty, editors, *Theorem Proving in Higher Order Logics*, volume 1275 of *Lect. Notes in Comp. Sci.*, pages 275–290. Springer-Verlag, 1997.

[Sli99] Konrad Slind. *Reasoning about Terminating Functional Programs*. PhD thesis, Institut für Informatik, TU München, 1999.

[Sne85] Gregor Snelting. Experiences with the PSG — Programming System Generator. In H. Ehrig, C. Floyd, M. Nivat, and J. Thatcher, editors, *Formal Methods and Software Development (TAPSOFT '85, Vol. 2)*, volume 186 of *Lect. Notes in Comp. Sci.*, pages 148–162. Springer-Verlag, 1985.

[Sym94] D. Syme. Reasoning with the formal definition of Standard ML in HOL. In Joyce and Seger [JS94], pages 43–58.

[Sym99] Donald Syme. Proving Java type soundness. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lect. Notes in Comp. Sci.*, pages 83–118. Springer-Verlag, 1999.

[TW97] Haykal Tej and Burkhart Wolff. A corrected failure-divergence model for CSP in Isabelle/HOL. In J. Fitzgerald, C.B. Jones, and P. Lucas, editors, *FME '97: Industrial Applications and Strengthened Foundations of Formal Methods*, volume 1313 of *Lect. Notes in Comp. Sci.*, pages 318–337. Springer-Verlag, 1997.

[VG94] M. VanInwegen and E. Gunter. HOL-ML. In Joyce and Seger [JS94], pages 59–72.