Axiomatic Semantics for Java$^{\ell ight}$ in Isabelle/HOL

David von Oheimb[⋆]

Technische Universität München
http://www.in.tum.de/~oheimb/

Abstract. We introduce a Hoare-style calculus for a nearly full subset of sequential Java, which we call Java$^{\ell ight}$. In particular, we present solutions to challenging features like exception handling, static initialization of classes and dynamic binding of methods.
This axiomatic semantics has been proved sound and complete w.r.t. pour operational semantics of Java$^{\ell ight}$, described in earlier papers. To our knowledge, our Hoare logic is the first one for an object-oriented language that has been proved complete. The proofs also give new insights into the role of type-safety. All the formalization and proofs have been done with the theorem prover Isabelle/HOL.

1 Introduction

Since languages like Java are widely used in safety-critical applications, verification of object-oriented programs has grown more and more important. A first step towards verification seems to be developing a suitable axiomatic semantics (a.k.a. "Hoare logic") for such languages.

Recently several proposals for Hoare logics for object-oriented languages, e.g. [dB99,PHM99,HJ00], have been given. Typically they deal with some small core language and are partially proved sound on paper (except for [HJ00], which has been machine-checked). None of them has been proved complete. Our new logic, in part inspired by [PHM99], has the following special merits.

– Apart from static overloading and dynamic binding of methods as well as references to dynamically allocated objects, it also covers full exception handling, static fields and methods, and static initialization of classes. Thus our sequential sublanguage Java$^{\ell ight}$ is almost the same as Java Card[Sun99].
– Instead of modeling expressions with side-effects as assignments to intermediate variables, it handles them first-class. Thus programs to be verified do not need to undergo an artificial structural transformation.
– It is both sound – w.r.t. a mature formalization of the operational semantics of Java – and complete. This means that programs using even non-trivial features like mutual recursion and dynamic binding can be proved correct.
– It has been both defined and verified within the interactive theorem proving system Isabelle/HOL[Pau94]. This guarantees a rigorous and unambiguous formalization and reliable proofs.

⋆ Research funded by the DFG Project Bali, http://isabelle.in.tum.de/Bali/

2 Some basics of the Java$^{\ell ight}$ formalization

Our axiomatic semantics inherits all features concerning type declarations and the program state from our operational semantics of Java$^{\ell ight}$. See [ON99] for a more detailed description.

Here we just recall that a program Γ (which serves as the context for most judgments) consists of a list of class and interface declarations and that the execution state is defined as

datatype $\mathit{st} = \mathsf{st}\ (\mathit{globs})\ (\mathit{locals})$
types $\mathit{state} = \mathit{xcpt\ option} \times \mathit{st}$

where *globs* and *locals* map class references to objects (including class objects) and variable names to values, respectively, and *xcpt* references an exception object. Using the projection operators on tuples, we define e.g. $\mathsf{normal}\ \sigma \equiv \mathsf{fst}\ \sigma = \mathsf{None}$, which expresses that in state σ there is no pending exception, and write $\mathsf{snd}\ \sigma$ to refer to the state without the information on exceptions, typically denoted by s.

A term of Java$^{\ell ight}$ is either an expression, a statement, a variable, or an expression list, and has a corresponding result. For uniformity, even a statement has a (dummy) result, called Unit. The result of a variable is an *lval*, which is a value (for read access) and a state update function (for write access).

types $\mathit{terms} = (\mathit{expr} + \mathit{stmt}) + \mathit{var} + \mathit{expr\ list}$
types $\mathit{vals} \quad = \quad \mathit{val} \quad + \mathit{lval} + \mathit{val\ list}$
types $\mathit{lval} \quad = \mathit{val} \times (\mathit{val} \to \mathit{state} \to \mathit{state})$

There are many other auxiliary type and function definitions which we cannot define here for lack of space. The complete Isabelle sources, including an example, may be obtained from `http://isabelle.in.tum.de/Bali/src/Bali4/`.

3 The axiomatic semantics

3.1 Assertions

In our axiomatic semantics we shallow-embed assertions in the meta logic HOL, i.e. define them as predicates on (basically) the state, making the dependence on the state explicit and simplifying their handling within Isabelle. This general approach is extended in two ways.

– We let the assertions depend also on so-called *auxiliary variables* (denoted by the meta variable Z of any type α), which are required to relate variable contents between pre- and postconditions, as discussed in [Sch97].
– We extend the state by a stack (implemented as a list and denoted by Y) of result values of type res, which are used to transfer results between Hoare triples. In an operational semantics, these nameless values can be referred to via meta variables, but in an axiomatic semantics, such a simple technique is impossible since all values in a triple are logically bound to that scope (by universal quantification).

As a result, we define the type of assertions (with parameter α) as

types $\alpha\ \mathit{assn} = \mathit{res\ list} \times \mathit{state} \to \alpha \to \mathit{bool}$
datatype $\mathit{res} = \mathsf{Res}\ (\mathit{vals})\ |\ \mathsf{Xcpt}\ (\mathit{xcpt\ option})\ |\ \mathsf{Lcls}\ (\mathit{locals})\ |\ \mathsf{DynT}\ (\mathit{tname})$

We write e.g. $\mathsf{Val}\ v$ as an abbreviation for $\mathsf{Res}\ (\mathsf{In1}\ v)$, injecting a value v into *res*. Names like Val and DynT are used not only as constructors, but also as (destructor) patterns. For example, $\lambda\mathsf{Val}\ v{:}Y.\ f\ v\ Y$ is a function on the result stack that expects a value v as the top element and passes it to f together with the rest of the stack, referred to by Y.

In order to keep the Hoare rules short and thus more readable, we define several assertion (predicate) transformers.

- $\lambda s{:}\ P\ s \equiv \lambda(Y,\sigma).\ P\ (\mathsf{snd}\ \sigma)\ (Y,\sigma)$ allows P to peek at the state directly.
- $P \wedge.\ p\ \equiv \lambda(Y,\sigma)\ Z.\ P\ (Y,\sigma)\ Z \wedge p\ \sigma$ means that not only P holds but also p (applied to the program state only). The assertion $\mathsf{Normal}\ P \equiv P \wedge.\ \mathsf{normal}$ is a simple application stating that P holds and no exception has occurred.
- $P{\leftarrow}{:}f \equiv \lambda(Y,\sigma).\ P\ (Y,f\ \sigma)$ means that P holds for the state transformed by f.
- $P\ {;}.\ f \equiv \lambda(Y,\sigma')\ Z.\ \exists\sigma.\ P\ (Y,\sigma)\ Z \wedge \sigma' = f\ \sigma$ means that P holds for some state σ and the current state is then derived from σ by the state transformer f.

3.2 Hoare triples and validity

We define triples as judgments of the form $prog{\vdash}\{\alpha\ assn\}\ terms{\succ}\ \{\alpha\ assn\}$ with some obvious variants for the different sorts of terms, e.g.
$\Gamma{\vdash}\{P\}\ e{-}{\succ}\ \{Q\} \equiv \Gamma{\vdash}\{P\}\ \mathsf{In1}(\mathsf{Inl}\ e){\succ}\ \{Q\}$ and $\{P\}\ .c.\ \{Q\} \equiv \{P\}\ \mathsf{In1}(\mathsf{Inr}\ c){\succ}\ \{Q\}$.

Here we simplify the presentation by leaving out triples as assumptions within judgments, which are necessary to handle recursion; we have discussed this issue in detail in [Ohe99]. The validity of triples is defined as

$$\begin{aligned}\Gamma{\models}\{P\}\ t{\succ}\ \{Q\} \equiv \forall Y\ \sigma\ Z.\ \ & P\ (Y,\sigma)\ Z\ \longrightarrow\ \mathsf{type\_ok}\ \Gamma\ t\ \sigma\ \longrightarrow\\ & \forall v\ \sigma'.\ \ \Gamma{\vdash}\sigma\ {-}t{\succ}{\rightarrow}\ (v,\sigma')\ \longrightarrow\ Q\ (\mathsf{res}\ t\ v\ Y,\sigma')\ Z\end{aligned}$$

where Y stands for the result stack and Z denotes the auxiliary variables. The judgment $\mathsf{type\_ok}\ \Gamma\ t\ \sigma$ means that the term t is well-typed (if σ is a normal state) and that all values in σ conform to their static types. This additional precondition is required to ensure soundness, as discussed in §3.6. $\Gamma{\vdash}\sigma\ {-}t{\succ}{\rightarrow}\ (v,\sigma')$ is the evaluation judgment from the operational semantics meaning that from the initial state σ the term t evaluates to a value v and final state σ'. Note that we define partial correctness.

Unless t is statement, the result value v is pushed onto the result stack via $\mathsf{res}\ t\ v\ Y \equiv \mathsf{if}\ \mathsf{is\_stmt}\ t\ \mathsf{then}\ Y\ \mathsf{else}\ \mathsf{Res}\ v{:}Y$.

3.3 Result value passing

We define the following abbreviations for producing and consuming results:

- $P{\uparrow}{:}w\ \equiv \lambda(Y,\sigma).\ P\ (w{:}Y,\sigma)$ means that P holds where the result w is pushed.
- $\lambda w{:}.\ P\ w \equiv \lambda(w{:}Y,\sigma).\ P\ w\ (Y,\sigma)$ expects and pops a result w and asserts $P\ w$.

A typical application of the former is the rule for literal values v:

$$Lit\ \frac{}{\Gamma{\vdash}\{\mathsf{Normal}\ (P{\uparrow}{:}\mathsf{Val}\ v)\}\ \mathsf{Lit}\ v{-}{\succ}\ \{P\}}$$

Analogously to the well-known assignment rule, it states that for a literal expression (i.e., constant) v the postcondition P can be derived if P – with the value v inserted – holds as the precondition and the (pre-)state is normal.

The rule for array variables handles result values in a more advanced way:

$$AVar\ \frac{\Gamma\vdash\{\mathsf{Normal}\ P\}\ e_1\text{-}\!\succ\ \{Q\}\qquad \Gamma\vdash\{Q\}\ e_2\text{-}\!\succ\ \{\lambda\mathsf{Val}\ i{:}.\ \mathsf{RefVar}\ (\mathsf{avar}\ \Gamma\ i)\ R\}}{\Gamma\vdash\{\mathsf{Normal}\ P\}\ e_1[e_2]\text{=}\!\succ\ \{R\}}$$

where $\mathsf{RefVar}\ vf\ P \equiv \lambda(\mathsf{Val}\ a{:}Y,(x,s)).\ \mathsf{let}\ (v,x') = vf\ a\ x\ s\ \mathsf{in}\ (P{\uparrow}{:}\mathsf{Var}\ v)\ (Y,(x',s))$. Both subexpressions are evaluated in sequence, where Q as intermediate assertion typically involves the result of e_1. The final postcondition R is modified for the proof on e_2 as follows: from the result stack two values are expected and popped, namely i (the index) and a (an address) of e_2 and e_1, respectively. Out of these and the intermediate state (x,s), the auxiliary function avar computes the variable v, which is pushed as the final result, and (possibly) an exception x'.

For terms involving a condition, we define the assertion $P{\uparrow}{:}\mathsf{Bool}{=}b \equiv \lambda(Y,\sigma)\ Z.\ \exists v.\ (P{\uparrow}{:}\mathsf{Val}\ v)\ (Y,\sigma)\ Z \wedge (\mathsf{normal}\ \sigma \longrightarrow \mathsf{the\_Bool}\ v = b)$ expressing (basically) that the result of a preceding boolean expression is b. Together with the meta-level conditional expression $(\mathsf{if}\ b\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2)$ depending on b and $P'{\uparrow}{:}\mathsf{Bool}{=}b$ identifying b with the result of a boolean expression e_0, we can describe both branches of conditional terms with a single triple, like in

$$Cond\ \frac{\Gamma\vdash\{\mathsf{Normal}\ P\}\ e_0\text{-}\!\succ\ \{P'\}\quad \forall b.\ \Gamma\vdash\{P'{\uparrow}{:}\mathsf{Bool}{=}b\}\ (\mathsf{if}\ b\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2)\text{-}\!\succ\ \{Q\}}{\Gamma\vdash\{\mathsf{Normal}\ P\}\ e_0\ \ ?\ e_1\ :\ e_2\text{-}\!\succ\ \{Q\}}$$

The value b is universally quantified, such that when applying this rule, one has to prove its second antecedent for any possible value, i.e., both True and False. What is a notational convenience here (to avoid two triples, one for each case), will be essential for the *Call* rule, given below.

The rules for the standard statements appear almost as usual:

$$Skip\ \frac{}{\Gamma\vdash\{P\}\ .\mathsf{Skip}.\ \{P\}}\qquad Loop\ \frac{\Gamma\vdash\{P\}\ e\text{-}\!\succ\ \{P'\}\qquad \Gamma\vdash\{P'{\uparrow}{:}\mathsf{Bool}{=}\mathsf{True}\}\ .c.\ \{P\}}{\Gamma\vdash\{P\}\ .\mathtt{while}(e)\ c.\ \{P'{\uparrow}{:}\mathsf{Bool}{=}\mathsf{False}\}}$$

Note that in all[1] rules (except Loop for obvious reasons) the postconditions of the conclusion is a variable. Thus in the typical “backward-proof” style of Hoare logic the rules are applied easily.

3.4 Dynamic binding

The great challenge of an axiomatic semantics for an object-oriented language is dynamic binding in method calls, for two reasons.

First, the code selected depends on the class D dynamically computed from a reference expression e. The range of values for D depends on the whole program and thus cannot be fixed locally, in contrast to the two possible boolean values appearing in conditional terms described above. Standard Hoare triples cannot express such an unbound case distinction. We handle this problem with the strong technique given above, using universal quantification and the precondition $R{\uparrow}{:}\mathsf{DynT}\ D \wedge\ldots$ with the special result value $\mathsf{DynT}\ D$. An alternative solution is

[1] The rules not mentioned here may be found in the appendix.

given in [PHM99], where D is referred to via This and the possible variety of D is handled in a cascadic way using two special rules.

Second, the actual value D often can be inferred statically, but in general for invocation mode "virtual", one can only know that it is a subtype of some reference type rt computed by static analysis during type-checking. The intuitive – but absolutely non-trivial – reason why the subtype relation Class $D \preceq$RefT rt holds is of course type-safety. The problem here is how to establish this relation. The rules given in [PHM99], for example, put the burden of verifying the relation on the user, which is possible, but in general not practically feasible. In contrast, our solution make the relation available to the user as a helpful assumption (see the sub-formula $\Gamma \vdash mode {\to} D \preceq rt$ in the rule given below), which transfers the proof burden once and for all to the soundness proof on the meta-level.

The remaining parts of the rule for method calls deals with the unproblematic issues of argument evaluation, setting up the local variables (including parameters) of the called method and restoring the previous local variables on return, for which we use the special result value Lcls.

$$\mathit{Call}\ \frac{\begin{array}{c}\Gamma \vdash \{\mathsf{Normal}\ P\}\ e {-}\!\succ \{Q\} \\ \Gamma \vdash \{Q\}\ args \doteq\!\succ \{\lambda\mathsf{Vals}\ vs{:}\mathsf{Val}\ a{:}.\ \lambda s :\ \mathsf{let}\ D = \mathsf{dyn\_class}\ mode\ s\ a\ \tau\ \mathsf{in} \\ \quad R{\uparrow}{:}\mathsf{DynT}\ D{\uparrow}{:}\mathsf{Lcls}\ (\mathsf{locals}\ s){\longleftarrow}{:}\mathsf{init\_lvars}\ \Gamma\ D\ (mn,pTs)\ mode\ a\ vs\} \\ \forall D.\ \Gamma \vdash \{R{\uparrow}{:}\mathsf{DynT}\ D\ \wedge.\lambda\sigma.\ \mathsf{normal}\ \sigma \longrightarrow \Gamma \vdash mode {\to} D \preceq rt\} \\ \mathsf{Body}\ D\ (mn,pTs) {-}\!\succ \{\lambda\mathsf{Val}\ v{:}\mathsf{Lcls}\ l{:}.\ S{\uparrow}{:}\mathsf{Val}\ v{\longleftarrow}{:}\mathsf{set\_lvars}\ l\}\end{array}}{\Gamma \vdash \{\mathsf{Normal}\ P\}\ \{rt,\tau,mode\} e.mn(\{pTs\} args) {-}\!\succ \{S\}}$$

3.5 Class initialization

The static initialization of classes is an unpleasant feature to model as its structure depends on the class hierarchy and it is not syntax-driven but rather triggered on demand. Thus at several places, e.g. field access and method calls, one has to consider potential initialization of some referenced class C, which we denote by the special statement init C. If the class in question is already initialized, there is nothing do:

$$\mathit{Done}\ \frac{}{\Gamma \vdash \{\mathsf{Normal}\ (P\ \wedge.\ \mathsf{initd}\ C)\}\ .\mathsf{init}\ C.\ \{P\}}$$

Otherwise, initialization allocates a new static object, treats the superclass (if any), and finally invokes the static initializers of the class itself, whereby the current local variables have to be hidden and later restored:

$$\mathit{Init}\ \frac{\begin{array}{c}\mathsf{the}\ (\mathsf{class}\ \Gamma\ C) = (sc,\_,\_,\_,ini) \\ sup = \mathsf{if}\ C = \mathtt{Object}\ \mathsf{then}\ \mathsf{Skip}\ \mathsf{else}\ \mathsf{init}\ sc \\ \Gamma \vdash \{\mathsf{Normal}\ ((P\ \wedge.\ \mathsf{Not} \circ \mathsf{initd}\ C)\ ;.\ \mathsf{supd}\ (\mathsf{new\_stat\_obj}\ \Gamma\ C))\}\ .sup.\ \{Q{\uparrow}{:}.\lambda s.\ \mathsf{Lcls}\ (\mathsf{locals}\ s)\} \\ \Gamma \vdash \{Q\ ;.\ \mathsf{set\_lvars\ empty}\}\ .ini.\ \{\lambda\mathsf{Lcls}\ l{:}.\ R{\longleftarrow}{:}\mathsf{set\_lvars}\ l\}\end{array}}{\Gamma \vdash \{\mathsf{Normal}\ (P\ \wedge.\ \mathsf{Not} \circ \mathsf{initd}\ C)\}\ .\mathsf{init}\ C.\ \{R\}}$$

3.6 Soundness and completeness

With the help of Isabelle/HOL, we have proved soundness and completeness:

$\mathsf{wf\_prog}\ \Gamma \ \longrightarrow\ \Gamma{\models}\{P\}\ t{\succ}\ \{Q\} \ =\ \Gamma{\vdash}\{P\}\ t{\succ}\ \{Q\}$

where $\mathsf{wf\_prog}\ \Gamma$ means that the program Γ is well-formed. As usual, soundness is proved by rule induction on the derivation of triples. Surprisingly, type-safety plays a crucial role here. The important fact that for method calls the subtype relation $\mathsf{Class}\ D{\preceq}\mathsf{RefT}\ rt$ holds can be derived only if the state conforms to the environment. This was the reason for bringing the judgment $\mathsf{type\_ok}$ into our definition of validity, which also gives rise to the new rule (required for the completeness proof)

$$hazard\ \frac{}{\Gamma{\vdash}\{P\ \wedge.\ \mathsf{Not} \circ \mathsf{type\_ok}\ \Gamma\ t\}\ t{\succ}\ \{Q\}}$$

indicating that if at any time conformance was violated, anything could happen.

Completeness is proved (basically) by structural induction with the MGF approach discussed in [Ohe99]. This includes an outer auxiliary induction on the number of methods already verified, which requires well-typedness in order to ensure that for any program there is only a finite number of methods to consider. Due to class initialization, an extra induction on the number of classes already initialized is required.

4 Example

To illustrate our approach and for gaining experience how our Hoare logic behaves in practice, 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=new Ext();
try {e.foo(null); }
catch (NullPointerException z) {throw z; }
```

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. All important features of Java$^{\ell ight}$ are taken into account.

We prove that if there is enough memory to successfully allocate an instance of `Ext`, calling `foo` on this instance with a `null` argument will eventually throw a `NullPointer` exception. This is because, taking dynamic binding into account, `foo` of `Ext` is called, which attempts to assign to the field `vee` through the `null` reference. The resulting exception is caught, but immediately re-thrown.

$$\mathsf{enough\_mem} \longrightarrow \mathit{tprg}\vdash\{\mathsf{Normal}\ \mathit{Any}\}\ .\mathit{tblk}.\ \{\mathit{Any} \wedge.\ \lambda s.\ \mathit{tprg},s\vdash\mathsf{catch}\ \mathsf{SXcpt}\ \mathit{NP}\}$$

$\mathit{where}\ \mathit{Any} = \lambda(Y,\sigma)\ Z.\ \mathsf{True}$

$\mathit{tprg} = ([],[(\mathit{Base},\mathit{BaseCl}),(\mathit{Ext},\mathit{ExtCl})]@\mathsf{system\_classes})$

$\mathit{tblk} = \mathsf{Expr}(\mathsf{LVar}\ e\texttt{:=new}\ \mathit{Ext})\texttt{;}$

$\quad\texttt{try}\ \mathsf{Expr}(\{\mathsf{ClassT}\ \mathit{Base},\mathsf{ClassT}\ \mathit{Base},\mathsf{IntVir}\}$

$\quad\quad\mathsf{Acc}\ (\mathsf{LVar}\ e)\texttt{.}\mathit{foo}(\{[\mathsf{Class}\ \mathit{Base}]\}[\mathsf{Lit\ Null}]))$

$\quad\texttt{catch}((\mathsf{SXcpt}\ \texttt{NullPointer})\ z)\ (\texttt{throw}\ (\mathsf{Acc}\ (\mathsf{LVar}\ z)))$

The proof is done in the typical "backward"-style and takes some 80 steps of rule application, simplification, and classical reasoning. About 30% of this deals with class initialization. The rule of consequence is applied four times, and there are four explicit instantiations of schematic assertion variables.

5 Conclusion

We have sketched a Hoare logic for a (rather extensive) subset of Java. This logic seems to be the first one for an object-oriented language that has been proved not only sound, but even complete.

Unfortunately, many of the rules given are quite complex. This is in part due to the result value handling for (sub-)expressions, for which we could not find a simpler solution. But the main point is that Java is an inherently difficult language, taking into account e.g. mutual recursion, dynamic binding, exception handling, and static initialization. As experience with our small example confirms, verifying programs heavily dealing with exceptions and class initialization is tedious, though further machine support might be a relief.

Nevertheless, using an axiomatic semantics like ours for program verifications helps concentrating on the interesting properties of a program (rather than fiddling with details of the state as with an operational semantics), and this experience carries over from procedural to object-oriented languages like Java.

Both for formalizing the Hoare rules and conducting the meta-level proofs, the support of the theorem proving system was indispensable. With some 400 lines of theories and about 1500 lines of (already rather condensed) proof scripts on a highly complex subject, otherwise there would not only be plenty of opportunity for omissions and inaccuracies, but also the sheer amount of inferences to perform by hand would be overwhelming. This is particularly true since within such a non-trivial project many iterations are performed, leading to frequent replay of the large proofs with often subtle, but possibly crucial differences.

References

dB99. Frank de Boer. A WP-calculus for OO. In *Foundations of Software Science and Computation Structures*, volume 1578 of *LNCS*. Springer-Verlag, 1999.

HJ00. Marieke Huisman and Bart Jacobs. Java program verfication via a Hoare logic with abrupt termination. In *Fundamental Approaches to Software Engineering (FASE'00)*, LNCS. Springer-Verlag, 2000. to appear.

Ohe99. David von Oheimb. Hoare logic for mutual recursion and local variables. In C. Pandu Rangan, V. Raman, and R. Ramanujam, editors, *FST&TCS'99*, volume 1738 of *LNCS*, pages 168–180. Springer-Verlag, 1999.

ON99. David von Oheimb and Tobias Nipkow. Machine-checking the Java specification: Proving type-safety. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *LNCS*. Springer-Verlag, 1999. `http://isabelle.in.tum.de/Bali/papers/Springer98.html`.

Pau94. Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *LNCS*. Springer-Verlag, 1994. For an up-to-date description, see `http://isabelle.in.tum.de/`.

PHM99. Arnd Poetzsch-Heffter and Peter Müller. A programming logic for sequential Java. In S. D. Swierstra, editor, *Programming Languages and Systems (ESOP '99)*, volume 1576 of *LNCS*, pages 162–176. Springer-Verlag, 1999.

Sch97. Thomas Schreiber. Auxiliary variables and recursive procedures. In *TAPSOFT'97*, volume 1214 of *LNCS*, pages 697–711. Springer-Verlag, 1997.

Sun99. Sun Microsystems. *Java Card Specification*, 1999. `http://java.sun.com/products/javacard/`.

A The remaining rules

$$\mathit{conseq}\ \frac{\begin{array}{c}\forall Y\ \sigma\ Z\ .\ P\ \ (Y\ ,\sigma)\ Z\ \longrightarrow (\exists P'\ \ Q'.\ \Gamma\vdash\{P'\}\ t\succ\ \{Q'\} \land (\forall w\ \sigma'.\\ (\forall Y'\ \ Z'.\ P'\ (Y',\sigma)\ Z' \longrightarrow Q'\ (\mathsf{res}\ t\ w\ Y',\sigma')\ Z') \longrightarrow Q\ \ (\mathsf{res}\ t\ w\ Y\ ,\sigma')\ Z))\end{array}}{\Gamma\vdash\{P\}\ t\succ\ \{Q\}}$$

$$\mathit{Xcpt}\ \frac{}{\Gamma\vdash\{(\lambda(Y,\sigma).\ P\ (\mathsf{res}\ t\ (\mathsf{arbitrary3}\ t)\ \ Y,\sigma))\ \land.\ \mathsf{Not}\circ\mathsf{normal}\}\ t\succ\ \{P\}}$$

$$\mathit{Super}\ \frac{}{\Gamma\vdash\{\mathsf{Normal}\ (\lambda s:\ P\uparrow:\mathsf{Val}\ (\mathsf{val\_this}\ s))\}\ \mathtt{super}{-}\!\succ\ \{P\}}$$

$$\mathit{LVar}\ \frac{}{\Gamma\vdash\{\mathsf{Normal}\ (\lambda s:\ P\uparrow:\mathsf{Var}\ (\mathsf{lvar}\ vn\ s))\}\ \mathsf{LVar}\ vn{=}\!\succ\ \{P\}}$$

$$\mathit{FVar}\ \frac{\Gamma\vdash\{\mathsf{Normal}\ P\}\ .\mathsf{init}\ C.\ \{Q\}\qquad \Gamma\vdash\{Q\}\ e{-}\!\succ\ \{\mathsf{RefVar}\ (\mathsf{fvar}\ C\ \mathit{stat}\ \mathit{fn})\ R\}}{\Gamma\vdash\{\mathsf{Normal}\ P\}\ \{C,\mathit{stat}\}e.\mathit{fn}{=}\!\succ\ \{R\}}$$

$$\mathit{Acc}\ \frac{\Gamma\vdash\{\mathsf{Normal}\ P\}\ va{=}\!\succ\ \{\lambda\mathsf{Var}\ (v,f):.\ Q\uparrow:\mathsf{Val}\ v\}}{\Gamma\vdash\{\mathsf{Normal}\ P\}\ \mathsf{Acc}\ va{-}\!\succ\ \{Q\}}$$

$$\mathit{Ass}\ \frac{\begin{array}{c}\Gamma\vdash\{\mathsf{Normal}\ P\}\ va{=}\!\succ\ \{Q\}\\ \Gamma\vdash\{Q\}\ e{-}\!\succ\ \{\lambda\mathsf{Val}\ v{:}\mathsf{Var}\ (w,f):.\ R\uparrow:\mathsf{Val}\ v{\leftarrow}{:}\mathsf{assign}\ f\ v\}\end{array}}{\Gamma\vdash\{\mathsf{Normal}\ P\}\ va{:}{=}e{-}\!\succ\ \{R\}}$$

$$\textit{Nil}\ \frac{}{\{\textsf{Normal}\ P{\uparrow}\!:\!\textsf{Vals}\ []\}\ []\doteq\!\succ \{P\}}$$

$$\textit{Cons}\ \frac{\Gamma\vdash\{\textsf{Normal}\ P\}\ e\!-\!\succ \{Q\} \qquad \Gamma\vdash\{Q\}\ es\doteq\!\succ \{\lambda\textsf{Vals}\ vs\!:\!\textsf{Val}\ v\!:\!.\ R{\uparrow}\!:\!\textsf{Vals}\ (v\!:\!vs)\}}{\Gamma\vdash\{\textsf{Normal}\ P\}\ e\!:\!es\doteq\!\succ \{R\}}$$

$$\textit{NewC}\ \frac{\Gamma\vdash\{\textsf{Normal}\ P\}\ .\textsf{init}\ C.\ \{\textsf{Alloc}\ \Gamma\ (\textsf{CInst}\ C)\ id\ Q\}}{\Gamma\vdash\{\textsf{Normal}\ P\}\ \texttt{new}\ C\!-\!\succ \{Q\}}$$

where $\textsf{Alloc}\ \Gamma\ otag\ f\ P \equiv$
$\lambda(Y,(x,s))\ Z.\ \forall\sigma'\ a.\ \Gamma\vdash(f\ x,s)\ -\textsf{halloc}\ otag\!\succ\! a\!\rightarrow\ \sigma' \longrightarrow (P{\uparrow}\!:\!\textsf{Val}\ (\textsf{Addr}\ a))\ (Y,\sigma')\ Z$

$$\textit{NewA}\ \frac{\begin{array}{c}\Gamma\vdash\{\textsf{Normal}\ P\}\ .\textsf{init\_comp\_ty}\ T.\ \{Q\}\\ \Gamma\vdash\{Q\}\ e\!-\!\succ \{\lambda\textsf{Val}\ i\!:\!.\ \textsf{Alloc}\ \Gamma\ (\textsf{Arr}\ T\ (\textsf{the\_Intg}\ i))\ (\textsf{check\_neg}\ i)\ R\}\end{array}}{\Gamma\vdash\{\textsf{Normal}\ P\}\ \texttt{new}\ T[e]\!-\!\succ \{R\}}$$

$$\textit{Cast}\ \frac{\Gamma\vdash\{\textsf{Normal}\ P\}\ e\!-\!\succ \{\lambda\textsf{Val}\ v\!:\!.\ Q{\uparrow}\!:\!\textsf{Val}\ v\!\leftarrow\!\!:\!\lambda(x,s).\ (\textsf{raise\_if}\ (\neg\Gamma,s\vdash v\ \textsf{fits}\ T)\ \texttt{ClassCast}\ x,s)\}}{\Gamma\vdash\{\textsf{Normal}\ P\}\ \textsf{Cast}\ T\ e\!-\!\succ \{Q\}}$$

$$\textit{Inst}\ \frac{\Gamma\vdash\{\textsf{Normal}\ P\}\ e\!-\!\succ \{\lambda\textsf{Val}\ v\!:\!.\ \lambda s:\ (Q{\uparrow}\!:\!\textsf{Val}\ (\textsf{Bool}\ (v\neq\textsf{Null}\ \wedge\ \Gamma,s\vdash v\ \textsf{fits}\ \textsf{RefT}\ T)))\}}{\Gamma\vdash\{\textsf{Normal}\ P\}\ e\ \texttt{instanceof}\ T\!-\!\succ \{Q\}}$$

$$\textit{Body}\ \frac{\begin{array}{c}\textsf{the}\ (\textsf{cmethd}\ \Gamma\ C\ sig) = (md,\ \_,\ \_,\ blk,\ \textsf{res})\\ \Gamma\vdash\{\textsf{Normal}\ P\}\ .\textsf{init}\ md.\ \{Q\} \quad \Gamma\vdash\{Q\}\ .blk.\ \{R\} \quad \Gamma\vdash\{R\}\ \textsf{res}\!-\!\succ \{S\}\end{array}}{\Gamma\vdash\{\textsf{Normal}\ P\}\ \textsf{Body}\ C\ sig\!-\!\succ \{S\}}$$

$$\textit{Expr}\ \frac{\Gamma\vdash\{\textsf{Normal}\ P\}\ e\!-\!\succ \{\lambda w\!:\!.\ Q\}}{\Gamma\vdash\{\textsf{Normal}\ P\}\ .\textsf{Expr}\ e.\ \{Q\}} \qquad \textit{Comp}\ \frac{\Gamma\vdash\{\textsf{Normal}\ P\}\ .c_1.\ \{Q\} \quad \Gamma\vdash\{Q\}\ .c_2.\ \{R\}}{\Gamma\vdash\{\textsf{Normal}\ P\}\ .c_1;c_2.\ \{R\}}$$

$$\textit{If}\ \frac{\Gamma\vdash\{\textsf{Normal}\ P\}\ e\!-\!\succ \{P'\} \qquad \forall b.\ \Gamma\vdash\{P'{\uparrow}\!:\!\textsf{Bool}{=}b\}\ .(\textsf{if}\ b\ \textsf{then}\ c_1\ \textsf{else}\ c_2).\ \{Q\}}{\Gamma\vdash\{\textsf{Normal}\ P\}\ .\texttt{if}(e)\ c_1\ \texttt{else}\ c_2.\ \{Q\}}$$

$$\textit{Throw}\ \frac{\Gamma\vdash\{\textsf{Normal}\ P\}\ e\!-\!\succ \{\lambda\textsf{Val}\ a\!:\!.\ Q\!\leftarrow\!\!:\!\lambda(x,s).\ (\textsf{throw}\ a\ x,s)\}}{\Gamma\vdash\{\textsf{Normal}\ P\}\ .\texttt{throw}\ e.\ \{Q\}}$$

$$\textit{Try}\ \frac{\begin{array}{c}\Gamma\vdash\{\textsf{Normal}\ P\}\ .c_1.\ \{Q\}\\ \Gamma\vdash\{(Q\ \wedge.\lambda\sigma.\ \ \Gamma,\sigma\vdash\textsf{catch}\ C)\ ;.\ \textsf{new\_xcpt\_var}\ vn\}\ .c_2.\ \{R\}\\ \Gamma\vdash\{Q\ \wedge.\lambda\sigma.\ \neg\Gamma,\sigma\vdash\textsf{catch}\ C\}\ .\textsf{Skip}.\ \{R\}\end{array}}{\Gamma\vdash\{\textsf{Normal}\ P\}\ .\texttt{try}\ c_1\ \texttt{catch}(C\ vn)\ c_2.\ \{R\}}$$

$$\textit{Fin}\ \frac{\begin{array}{c}\Gamma\vdash\{\textsf{Normal}\ P\}.c_1.\{\lambda(Y,(x,s)).\ (Q{\uparrow}\!:\!\textsf{Xcpt}\ x)\ (Y,(\textsf{None},s))\}\\ \Gamma\vdash\{\textsf{Normal}\ Q\}.c_2.\{\lambda\textsf{Xcpt}\ x'\!:\!.\ R\!\leftarrow\!\!:\!\lambda(x,s).\ (\textsf{xcpt\_if}\ (x'\neq\textsf{None})\ x'\ x,s)\}\end{array}}{\Gamma\vdash\{\textsf{Normal}\ P\}\ .c_1\ \texttt{finally}\ c_2.\ \{R\}}$$