Hoare Logic for Java in Isabelle/HOL

David von Oheimb

Institut für Informatik
Technische Universität München
`http://www4.in.tum.de/~oheimb/`

SUMMARY

This article presents a Hoare-style calculus for a substantial subset of Java Card, which we call Java$^{\ell ight}$. In particular, the language includes side-effecting expressions, mutual recursion, dynamic method binding, full exception handling, and static class initialization.

The Hoare logic of partial correctness is proved not only sound (w.r.t. our operational semantics of Java$^{\ell ight}$, described in detail elsewhere) but even complete. It is the first logic for an object-oriented language that is provably complete. The completeness proof uses a refinement of the Most General Formula approach. The proof of soundness gives new insights into the role of type safety. Further by-products of this work are a new general methodology for handling side-effecting expressions and their results, the discovery of the strongest possible rule of consequence, and a flexible Call rule for mutual recursion. We also give a small but non-trivial application example.

All definitions and proofs have been done formally with the interactive theorem prover Isabelle/HOL. This guarantees not only rigorous definitions, but also gives maximal confidence in the results obtained.

KEY WORDS: Hoare logic, axiomatic semantics, Java, Isabelle/HOL, verification, soundness, completeness, auxiliary variables, side effects, mutual recursion, dynamic binding, exception handling, type safety.

1. Introduction

1.1. Motivation and Aims

Since languages like Java [17] are widely used even in safety-critical applications, verification of object-oriented programs is becoming more and more important. A crucial first step towards verification is developing a suitable axiomatic semantics (a.k.a. "Hoare logic") for such languages. The resulting proof system should be provably sound, easy to use, and complete. Due to the complexity of practical languages and the large amount of detail involved, support by mechanical theorem provers seems indispensable.

This article focuses on the meta-theory of program verification and its mechanization in a theorem prover. The work presented here is aimed to be a solid basis for further research and developments regarding methodological issues and tool support for actual program verification.

Received 16 September 2000
Revised 28 February 2001
Accepted 2 March 2001

1.2. Related Work

Various sublanguages of Java have been formalized in theorem proving systems. Syme [55] proves type soundness by directly mechanizing and improving the work of Drossopoulou and Eisenbach [13]. Within the LOOP project [28], Jacobs *et al.* [27, 26, 25] translate Java classes to PVS or Isabelle/HOL theories and perform program verification with a Hoare-style logic proved sound within their system. Within project Oasis [6], Attali *et al.* [7] give an executable transition semantics and develop a visualization tool for a large subset of Java within the Centaur system. Leino *et al.* [37] translate (almost all of) Java 1.2 to a guarded-command-like language in order to generate verification conditions for extended static checking.

Many Hoare logics for object-oriented languages have been proposed meanwhile. Abadi and Leino [1] describe a type-system-like core calculus that later has been formalized by Hofmann and Tang [21] in LEGO. The OO logic work by Leavens focuses on behavioral subtyping, *e.g.* [35], and specification languages, in particular JML [34]. The LOOP project mentioned above provides a formal semantics for JML [30]. Poetzsch-Heffter *et al.* give a Hoare logic for a core language of Java [51], where Müller focuses on modular verification [40]. All these logics have been proved sound (typically on paper). None of them has been proved complete. The Hoare logic for the language SPOOL by America and de Boer [3, 12] has been proved complete (on paper), yet SPOOL implements only a quite weak form of object-orientation: it offers records and methods on them, but lacks subtyping and dynamic binding.

Other more specific references to related work are given where appropriate. Our logic has been developed within Project Bali [42]. The proofs of soundness and completeness have been inspired by [51] and [32], respectively. This article is essentially a slightly revised standalone version of [47, §5].

1.3. Characteristics

Our Hoare logic for Java has the following characteristics.

Language Coverage Apart from static overloading and dynamic binding of methods as well as references to dynamically allocated objects, our language, called Java$^{\ell ight}$, also covers full exception handling, static fields and methods, and static initialization of classes. The only important features still missing, name spaces and visibility control, will be added soon. Thus Java$^{\ell ight}$ is quite close to Java Card [54] which is essentially the sequential part of Java.

Semantical Assertions Our semantic view of assertions identifies the assertion language with the underlying meta logic of our proof system. This simplifies the logics and makes the otherwise problematic issues of expressiveness of the assertion language and the completeness of the meta logic trivial. The dependency of assertions on the program state and auxiliary variables is made explicit, which is helpful for conducting meta-theory though a bit awkward for actual program verification.

First-class Expressions 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 artificial structural transformations. Jacobs *et al.* [26] claim to deal with side-effecting expressions, too, though their paper does not reveal their approach and their WHILE rule does not allow side-effects in the loop condition.

Proven Soundness and Completeness Our logic is both sound — w.r.t. a mature formalization of the operational semantics of Java$^{\ell ight}$ — and complete. This means that programs using even non-trivial features like mutual recursion, class initialization, and dynamic binding can be proved correct.

Reliability Both Java$^{\ell ight}$ and its Hoare logic have been defined and verified within Isabelle/HOL. This guarantees a rigorous and unambiguous formalization and reliable proofs.

1.4. Overview

This article is an extended version of [45] and [46]. We describe how to extend classical Hoare logic [18, 4] to handle side-effecting expressions, exceptions, mutual recursion, dynamic binding, static initialization and other difficult features.

- §2 and §3 give a brief introduction to the underlying theorem proving system Isabelle/HOL and our formalization of the static and operational semantics of Java$^{\ell ight}$.
- §4 to §9 present the axiomatic semantics of partial correctness, motivating and discussing all required concepts like assertions, triples and their validity, as well as the structural and Java-specific rules.
- §10 and §11 sketch the proofs of soundness and (relative) completeness, both w.r.t. the operational semantics.
- §12 reports on a small application example and §13 concludes giving a summary and some evaluation.

2. Isabelle/HOL

For our formalization and proofs we use the *Isabelle/HOL* system. This is the generic interactive theorem prover *Isabelle* [50] instantiated with Church's classical, simply-typed Higher-Order Logic (HOL) [10].

Isabelle/HOL provides an expressive specification language with type inference and advanced syntax capabilities resulting in well-readable formulas. It offers a variety of interactive and semi-automatic proof tools (not further described here) whose trustworthiness is due to the LCF [15] approach. The system is well-documented and equipped with the convenient Proof General interface [5].

In the next two subsections, we will briefly introduce the basics of Isabelle required for understanding the declarations and formulas given in the subsequent sections. We adopt the following typographic conventions: The names of logical constants like 'True' or 'cfield' appear in sans serif, while type names like '*bool*' or '*state*' and variables like '*v*' appear in italics, and Java keywords like '`catch`' appear in courier font.

2.1. Isabelle Definitions

Types follow the syntax of ML. Type abbreviations are introduced simply as equations. Logical constants are declared by giving their name and type, separated by '$::$'. Both types and constants may receive additional general mixfix syntax (possibly with graphical symbols and user-defined precedences). We write non-recursive definitions and abbreviations with '$\equiv$'.

The appearance of formulas is standard, *e.g.* '$\longrightarrow$' is the (right-associative) infix implication symbol. Quantifications have very low syntactical precedence, *i.e.* their scope extends to the end of the given formula or until the next closing parenthesis. Terms are expressions of an extended λ-calculus similar to ML. Function application is written in curried style. Predicates are functions with Boolean result.

There are primitive and general recursive function definitions as well as inductive definitions of relations, which we use heavily. A free datatype is defined by listing its constructors together with their argument types, separated by '$|$'.

2.2. HOL Library

By *HOL* we mean Isabelle/HOL, which is not to be confused with its nearest relative, Gordon's HOL system [14]. The fundamentals of HOL are introduced in [43], whereas the latest version of HOL can be found in [22]. A recent gentle introduction is [41].

In HOL there are simple types like *bool* and *nat*, as well as the polymorphic type $(\alpha)set$ of homogeneous sets for any element type α. Occasionally we apply the infix 'image' operator lifting a function over a set, defined as $f \text{ ``} S \equiv \{f\,x\ |x.\ x{\in}S\}$.

The product type $\alpha \times \beta$ comes with the projection functions fst and snd. Tuples are pairs nested to the right, *e.g.* $(a,b,c) = (a,(b,c))$. They may be used also as patterns like in $\lambda(x,y).\ f\,x\,y$.

The sum type $\alpha + \beta$ comes with the injections Inl and Inr. For the ternary sum $\alpha + \beta + \gamma$ we assume the injections $\mathsf{In1}$, $\mathsf{In2}$ and $\mathsf{In3}$.

We frequently use the datatype

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

It has an unpacking function $\mathsf{the} :: (\alpha)option \to \alpha$ such that $\mathsf{the}\,(\mathsf{Some}\,x) = x$. ($\mathsf{the}\ \mathsf{None}$ is left unspecified.)

3. Java$^{\ell ight}$ Formalization

Our axiomatic semantics inherits all features, in particular declarations and the program state, from our operational semantics of Java$^{\ell ight}$. Here we mention just the most important definitions used in the following sections. There are many other auxiliary type and function definitions which we cannot define here for lack of space. See [48] and [47, §2 and §3] for a complete and more detailed description. The actual Isabelle/HOL theories and proof scripts may be obtained from `http://isabelle.in.tum.de/Bali/src/Bali5/`.

3.1. Declarations

A *program*, denoted by Γ, consists of a list of interface and class declarations:

$$\mathit{prog} = (\mathit{idecl})\mathit{list} \times (\mathit{cdecl})\mathit{list}$$

It serves as the context for most judgments. The well-formedness predicate $\mathsf{wf\_prog}$ states that a program fulfills all the sanity constraints typically checked by the compiler. The (much weaker) well-structuredness predicate $\mathsf{ws\_prog}$ just states that the class and interface hierarchies are finite and acyclic.

3.2. Terms and Results

A Java$^{\ell ight}$ *term* is either an expression, a statement, a variable, or an expression list, and has a corresponding result. It helps to handle all four classes of terms as uniformly as possible. Thus even statements are modeled to have a (dummy) result. The result of a variable has type $\mathit{vvar} = \mathit{val} \times (\mathit{val} \to \mathit{state} \to \mathit{state})$, which is a value (for read access) and a state update function (for write access). This is reminiscent of *L-values* introduced by Strachey [53].

$$\begin{array}{lcl}
\mathit{terms} & = & (\mathit{expr} + \mathit{stmt}) + \mathit{var} + (\mathit{expr})\mathit{list} \\
\mathit{vals} & = & \quad \mathit{val} \qquad\ \ + \mathit{vvar} + (\mathit{val}\ )\mathit{list}
\end{array}$$

We will use the functions

$$\begin{array}{l}
\mathsf{Val}\ \ v \equiv \mathsf{In1}\ v \\
\mathsf{Var}\ \ v \equiv \mathsf{In2}\ v \\
\mathsf{Vals}\ v \equiv \mathsf{In3}\ v
\end{array}$$

for injecting single values, variables and value lists into *vals*. The names Val, Var, and Vals will be used not only for the injections, but also as (destructor) patterns. For example, $\lambda\mathsf{Val}\ v.\ f\ v$ is a function on the result entry that expects a single value v and passes it to f.

Statements in Java$^{\ell ight}$ are reduced to their essentials. The datatype definition gives the abstract syntax as follows.

$$\begin{array}{rl}
\mathit{stmt} = & \mathsf{Skip} \\
| & \mathsf{Expr}\ \mathit{expr} \\
| & \mathit{stmt}\texttt{;}\ \mathit{stmt} \\
| & \texttt{if}(\mathit{expr})\ \mathit{stmt}\ \texttt{else}\ \mathit{stmt} \\
| & \texttt{while}(\mathit{expr})\ \mathit{stmt} \\
| & \texttt{throw}\ \mathit{expr} \\
| & \texttt{try}\ \mathit{stmt}\ \texttt{catch}\ (\mathit{tname}\ \mathit{ename})\ \mathit{stmt} \\
| & \mathit{stmt}\ \texttt{finally}\ \mathit{stmt} \\
| & \mathsf{init}\ \mathit{tname}
\end{array}$$

The artificial statement $\mathsf{init}\ C$ is used to model static initialization of a class C at the points of (potential) first active use.

Java *expressions*, in particular method calls and object creation, do not only yield results but typically also cause side effects on the program state. A common modeling technique is to get rid of side effects by transforming the problematic expressions into a series of assignments (which are then considered as statements) to temporary variables. We believe that such a transformation is inadequate since it severely alters the structure of programs and has non-trivial semantical connections, *e.g.* to exception propagation. Therefore we handle expressions first-class, even if this causes inconveniences, above all for the axiomatic semantics (cf. §4.3).

The abstract syntax of Java$^{\ell ight}$ expressions (and *variables* contained in them) is

$$\begin{array}{rl}
expr = & \texttt{new}\ tname \\
& |\ \texttt{new}\ ty[expr] \\
& |\ \mathsf{Cast}\ ty\ expr \\
& |\ expr\ \texttt{instanceof}\ ref\_ty \\
& |\ \mathsf{Lit}\ val \\
& |\ \texttt{super} \\
& |\ \mathsf{Acc}\ var \\
& |\ var\ \texttt{:=}\ expr \\
& |\ expr\ \texttt{?}\ expr\ \texttt{:}\ expr \\
& |\ \{ref\_ty,\ ref\_ty,\ inv\_mode\}expr\texttt{..}mname(\{(ty)list\}(expr)list) \\
& |\ \mathsf{Methd}\ tname\ sig \\
& |\ \mathsf{Body}\ tname\ stmt\ expr
\end{array}$$

$$\begin{array}{rl}
var = & \mathsf{LVar}\ lname \\
& |\ \{tname,bool\}expr\texttt{..}ename \\
& |\ expr[expr]
\end{array}$$

The subterms in braces are so-called *type annotations*: extra information typically added by the compiler.

The auxiliary expression $\mathsf{Methd}\,C\,m$ is employed within method calls (cf. §9.7). It denotes the implementation of method m of class C, a concept crucial for the axiomatic semantics, as will be motivated in §9.7.2. The unfolded version of a method implementation is its actual body, for which we introduce $\mathsf{Body}\,D\,c\,e$, a further auxiliary term is a useful abstraction used for simplifying the $Methd$ rule of the axiomatic semantics. Here D is the defining class, c is the (block of) statements in the body, and e is the result expression, used to emulate `return` statements.

See §12 for example program terms.

3.3. Type System

We model the most important *primitive types* and all *reference types*, as well all *type relations* between them. Here we just mention the *widening* relation, where $\Gamma \vdash S \preceq T$ means that in the context Γ a value of type S may be used in a place where a value of type T is expected.

The fact that a term t is well-typed and has type T is expressed via (inductively defined) *typing judgments* $(\Gamma,\Lambda) \vdash t{::}T$ where (Γ,Λ) is the type environment consisting of the program and a local part mapping variable names to types.

3.4. State

The program state is defined as

$$\begin{array}{ll}
st & = \mathsf{st}\ globs\ locals \\
state & = (xcpt)option \times st
\end{array}$$

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 projections 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.

We further define functionals mapping an update of the exception or store part of the state to an update of the full state,

$$\begin{array}{l}
\mathsf{xupd} :: ((\mathit{xcpt})\mathit{option} \to (\mathit{xcpt})\mathit{option}) \to \mathit{state} \to \mathit{state}\\
\mathsf{supd} :: (\mathit{st} \to \mathit{st}) \to \mathit{state} \to \mathit{state}\\
\mathsf{xupd}\ f \equiv \lambda(x,s).\ (f\,x,s)\\
\mathsf{supd}\ f \equiv \lambda(x,s).\ (x,f\,s)
\end{array}$$

applied for instance when setting local variables:

$$\begin{array}{l}
\mathsf{set\_lvars} \quad :: \mathit{locals} \to \mathit{state} \to \mathit{state}\\
\mathsf{set\_lvars}\ l' \equiv \mathsf{supd}\ (\lambda s.\ \mathsf{case}\ s\ \mathsf{of}\ \mathsf{st}\ g\ l \to \mathsf{st}\ g\ l')
\end{array}$$

In our model many situations arise where under a certain condition an exception should be raised, yet only if no exception is already present which has to take precedence. This behavior is captured by the function

$$\begin{array}{l}
\mathsf{raise\_if}\ :: \mathit{bool} \to \mathit{xname} \to (\mathit{xcpt})\mathit{option} \to (\mathit{xcpt})\mathit{option}\\
\mathsf{raise\_if}\ c\ xn \equiv \lambda x'.\ \mathsf{if}\ c \wedge x' = \mathsf{None}\ \mathsf{then}\ \mathsf{Some}\ (\mathsf{StdXcpt}\ xn)\ \mathsf{else}\ x'
\end{array}$$

3.5. Evaluation

The general evaluation judgment has the form

$$\mathit{prog} \vdash \mathit{state}\ \xrightarrow{\mathit{terms}}\ (\mathit{vals} \times \mathit{state})$$

where $\Gamma \vdash \sigma\ \xrightarrow{t}\ (w,\sigma')$ means that in the context of program Γ evaluation of term t from the initial state σ terminates in state σ' and yields the result w. There are specific variants of the evaluation judgment for all classes of terms, *e.g.* for expressions e with value v: $\Gamma \vdash \sigma\ \xrightarrow{e \succ v}\ \sigma' \equiv \Gamma \vdash \sigma\ \xrightarrow{\mathsf{In1}\ (\mathsf{Inl}\ e)}\ (\mathsf{Val}\ v,\sigma')$

When developing the axiomatic semantics we had to adapt our model for method calls as given *e.g.* in [48]. As will be motivated in §9.7.2, we distinguish the callee's side of method calls, the method implementation, from the caller's side and further handle the actual method body separately [49]. This also helps to keep the complexity of the method call rule bearable.

Thus our adapted method call rule does not any more evaluate the statements and result expression of the method body directly but calls evaluation of the *method implementation* $\mathsf{Methd}\ C\ sig$ with signature sig of some class C. The method implementation rule then looks up the method according to this information and determines the information needed for the body:

$$\frac{\Gamma \vdash \mathsf{Norm}\ s_0\ \xrightarrow{\mathsf{body}\ \Gamma\ C\ sig \succ v}\ \sigma_1}{\Gamma \vdash \mathsf{Norm}\ s_0\ \xrightarrow{\mathsf{Methd}\ \ C\ sig \succ v}\ \sigma_1}$$

where $\mathsf{body}\ \Gamma\ C\ sig = \mathsf{let}\ (D,\_,\_,c,e) = \mathsf{the}\ (\mathsf{cmethd}\ \Gamma\ C\ sig)\ \mathsf{in}\ \mathsf{Body}\ D\ c\ e$ is an auxiliary function that looks up the the method with the given signature in the class hierarchy (starting from the class C) and inserts the method body where D is the defining class, the statement c represents the actual block of commands, and e is the result expression.

The evaluation of the body is just a sequential composition of initializing the current class (if required), executing the block of statements, and evaluating the result expression:

$$\frac{\Gamma \vdash \mathsf{Norm}\ s_0\ \xrightarrow{\mathsf{init}\ D}\ \sigma_1 \quad \Gamma \vdash \sigma_1\ \xrightarrow{c}\ \sigma_2 \quad \Gamma \vdash \sigma_2\ \xrightarrow{e \succ v}\ \sigma_3}{\Gamma \vdash \mathsf{Norm}\ s_0\ \xrightarrow{\mathsf{Body}\ D\ c\ e \succ v}\ \sigma_3}$$

4. Assertions

In designing an axiomatic semantics the most critical notion is that of *assertions*, *i.e.* propositions describing the pre- and postconditions of term execution. The language of assertions and the underlying logic strongly determine the expressiveness and completeness of the resulting verification logic.

4.1. Logical Language

As the assertion language and logic, we could use *Peano Arithmetic*, *i.e.* first-order predicate logic with equality, natural numbers, $+$ and $*$. This is because we will quantify essentially just over (lists and finite mappings of) values. The program state can be encoded using lists of (lists of) values, the potentially problematic variable update functions (contained in type *vvar*) can be coded as a simple choice between the three possible cases (local, field, and array variables), and for lists standard encodings exist. Thus a rather minimal language would be sufficient, but at the expense of technically awkward encodings. Based on these observations, we could define a notion of expressiveness suiting our needs.

We could embed the assertion language, Peano Arithmetic or something less minimal, deeply in our meta logic HOL. This means defining as a first step the (abstract) syntax of the language and then assign semantics to it. Such deep embeddings [9] are useful when doing meta-theory on the language in question since one can express properties of the syntactic structure and prove generic properties of the language. In particular, this would enable us to explicitly treat expressiveness and the whole issue of completeness (basically) along the lines of Cook [11], who did a good job separating concerns within the completeness affair. Yet as observed by Kleymann [32, §2.12], some of the known incompleteness results crucially depend on certain expressiveness properties and have been misinterpreted in the sense that they were attributed relevance for practical purposes, which they in fact do not have. Actually, we are not aware of any actual verification system based on Hoare logic where incompleteness of the underlying logic is an issue. Thus Kleymann and others follow Aczel's suggestion [2] not to consider expressiveness when investigating the completeness of a Hoare-style logic. Moreover, compared to a shallow embedding, a deep embedding would complicate in particular the proofs of soundness and completeness as it requires talking explicitly about the syntactic level (*i.e.* terms and substitutions) and their semantic interpretation, which would just add clutter without giving us any benefit.

For the reasons given above we shallow-embed the assertion language in HOL, *i.e.* short-circuit both logical languages. This not only relieves us from defining (and using) the syntax, semantics and proof rules of the assertion language — we do not have to bother with expressiveness. Concerning derivability, we automatically only have to deal with *relative completeness* in the sense of Cook [11], *i.e.* completeness of the Hoare logic rule system itself modulo (in-)completeness of the underlying meta logic. In particular, within the rule of consequence, the derivability of an implication between assertions is replaced simply by its validity, *i.e.* mere implication in the meta logic HOL.

Since assertions depend on the program variables (including the heap), from the HOL perspective, they are essentially just predicates on the state. We use the state as an explicit parameter of the assertions, which is most appropriate when conducting meta-theory. Thus an example Hoare triple that is traditionally given as $\{\mathsf{True}\}\ c\ \{X{=}\mathtt{1}\}$ now reads as $\{\lambda\sigma.\ \mathsf{True}\}\ c\ \{\lambda\sigma.\ \mathsf{locals}\ (\mathsf{snd}\ \sigma)\ X = \mathsf{Some}\ (\mathsf{Intg}\ \mathtt{1})\}$, where the rather cumbersome expression in the postcondition could of course be suitably abbreviated and the assertions pretty-printed. For actual program verification, a mechanism for hiding the state and directly referring to the program variable names, as given by Wenzel [56], would enhance readability.

4.2. Auxiliary Variables

Program verification typically involves relating pre- and postconditions of program terms, in particular when stating that a certain portion of the state does not change or when giving input-/output specifications of methods (or general procedures). Such relations are easily expressed in VDM [31] using "hooked" expressions within the postcondition to refer to the initial state. With plain Hoare logic, one cannot make such references, but one can extend the logic by so-called *auxiliary variables*, or *logical variables*, which are universally bound at a higher level.

For example, the proposition that a procedure P does not change the contents of a program variable X may be formulated as the triple $\{X{=}Z\}\ \mathsf{Call}\ P\ \{X{=}Z\}$, which should mean that whenever X has some value denoted by Z before calling P, after return it still has the same value, as given by Z. A potential interpretation for Z is to be a program variable not occurring in P, but this essential side condition cannot be expressed within the logic. A better and rather intuitive alternative is that Z be viewed as a free variable, which is thus implicitly universally quantified at the outermost logical level. Yet this gives the desired interpretation only if the triple occurs (implication-)positively, and thus is unsuitable when triples are used also in antecedents as required for verifying recursive methods (cf. §9.7.2).

Viewing Z as an arbitrary (yet fixed) constant preserves correctness, but this approach suffers from incompleteness: take a procedure triple like $\{X{=}Z\}\ \mathsf{Call}\ SQ\ \{Y{=}Z{*}Z\}$ as an example. Both for handling recursive calls during its proof and for different applications after it has been proved, different instantiations of Z may be required, which is impossible if Z is a constant. The classical but cumbersome and often incomplete way out, as described *e.g.* by Apt [4], is inventing some (more or less ad-hoc) set of substitution and adaptation rules involving sometimes intricate side-conditions on variable occurrences. A semantically satisfactory solution that we could adopt would involve (implicit or explicit) universal quantification at the level of triples like $\forall Z.\ \{P\ Z\}\ c\ \{Q\ Z\}$, but this changes the outer structure of Hoare triples and makes them more difficult to handle, in particular if they occur in the antecedents.

We follow the approach promoted and applied by Kleymann[*] [52]: implicit quantification of auxiliary variables at the level of triple validity. In order to abstract from the number and the names of the auxiliary variables used in different triples, assertions receive an extra parameter representing the collection of all required auxiliary variables. The type of the parameter is not specified and thus can be instantiated as appropriate, typically to a tuple of values (for actual program verification) or the whole state (for meta theory). With this extended notion of assertions, the motivating example $\{X{=}Z\}\ \mathsf{Call}\ P\ \{X{=}Z\}$ now reads as $\{\lambda\sigma\ Z.\ \mathsf{locals}\ (\mathsf{snd}\ \sigma)\ X = Z\}\ \mathsf{Call}\ P\ \{\lambda\sigma\ Z.\ \mathsf{locals}\ (\mathsf{snd}\ \sigma)\ X = Z\}$ where in this case the type of Z is of course the type of the variable X.

[*] formerly Schreiber

4.3. Result Values

We still need a further — orthogonal — extension of assertions, namely for handling the *result values* of side-effecting expressions. In contrast to most other axiomatic semantics given in the literature, and as already motivated in §3.2, we take such expressions seriously.

Homeier and Martin [23] appear to have been the first ones to embed side-effecting expressions in a machine-checked axiomatic semantics. They transform expressions syntactically into the assertion language while using simultaneous substitutions to account for side-effects. This solution does not require special triples and result value entries. On the other hand, it is not general enough because it can handle only variable assignments (including *e.g.* incrementation operators) but not method calls within expressions which appear frequently in object-oriented programs.

We use triples not only to describe the behavior of statements, but also all other classes of terms, *i.e.* expressions, expression lists, and variables. This requires a mechanism for not only recording, but also for passing on the values produced by these terms. In an operational semantics, the (nameless) result values can be referred to and passed on via meta variables bound at the outermost logical level, but in an axiomatic semantics, such a simple technique is impossible: the behavior of a term has to be described solely by a suitable triple without any reference to its surroundings. Thus all variables occurring in the pre- and postconditions of the triple have to be logically bound to that triple. Violating this principle easily leads to unsound or incomplete rule systems.

Kowaltowski [33] rather early pointed to the right direction giving a surprisingly simple (syntactic) solution: Within assertions there is a default reference, call it ρ, to the result of the current expression. The rule for constant expressions, for example, then reads as

$$\frac{}{\{P[c/\rho]\}\ c\ \{P\}}$$

which is reminiscent of the well-known assignment rule applied to $\rho := c$. The rule for an arbitrary binary operator $\oplus$ reads as

$$\frac{\{P\}\ d\ \{Q[\rho/\tau]\} \quad \{Q\}\ e\ \{R[\tau \oplus \rho/\rho]\}}{\{P\}\ d \oplus e\ \{R\}} \quad \tau \text{ does not occur elsewhere}$$

and can be justified by simulating the the special result variables with intermediate program variables. Unfortunately these rules cannot be used directly in our rigorous semantical setting as they rely on syntactic substitutions and the problematic syntactic side condition of variable freshness.

Another early treatment of side-effects is due to Boehm [8] who uses a kind of dynamic logic dividing between effects and values of expressions. This approach seems conceptually nice, but apparently leads to complex proofs.

After some experimentation we found and implemented a first solution of the result representation problem: let the assertions refer to a stack of result values. This not only gives a default reference, *viz.* the stack top, but also an arbitrary number of unique references for further intermediate results where the explicit syntactic shifting performed when having two or more intermediate results is handled by pushing and popping elements.

Later we noticed that there is a simpler solution: assertions receive the (single) current result value as a parameter. Thus substitution can be modeled simply by a combined abstraction and application, and the rule for constant expressions reads as

$$\frac{}{\{\lambda\rho.\ P\ c\}\ c\ \{P\}}$$

Note that the precondition effectively ignores the result parameter ρ, *i.e.* $P\ c$ does not depend on it, since it makes sense only in the postcondition. We will later (cf. §4.5) abbreviate the precondition to $P{\leftarrow}c$.

Multiple result values within a Hoare logic rule can be handled by suitable explicit universal quantification and substitution of all but the last value in the following way. First, observe that we may rewrite the rule for binary operators to

$$\frac{\{P\}\ d\ \{Q\} \quad \{Q[\tau/\rho]\}\ e\ \{R[\tau \oplus \rho/\rho]\}}{\{P\}\ d \oplus e\ \{R\}} \quad \text{for some } \tau \text{ not occurring elsewhere}$$

This form has two advantages: we only need substitution to ρ and the side condition on τ can be made local to the second triple in the assumptions. Thus we can model the side condition semantically by universal quantification of τ around the second triple and end up with the rule

$$\frac{\{P\}\ d\ \{Q\} \quad \forall\tau.\ \{\lambda\rho.\ Q\ \tau\}\ e\ \{\lambda\rho.\ R\ (\tau \oplus \rho)\}}{\{P\}\ d \oplus e\ \{R\}}$$

Both subexpressions are evaluated in sequence, where Q as intermediate assertion typically involves the result of d. The final postcondition R is modified for the proof on e as follows: we take the second intermediate result ρ, combine it with the first intermediate result τ as obtained from the precondition, and use the combined value as the overall result.

Huisman [25] also deals with side-effecting expressions, though she is not explicit about the approach employed. Yet the rules given reveal that she uses a similar (though more complicated) semantical technique, namely existentially quantified result functions that depend on an universally quantified state parameter.

4.4. Assertion Type

Having decided on the logical language, the use of auxiliary variables, and the result entry, we can finally give the type of assertions. It has a type parameter α for the auxiliary variables and is defined as a relation between a result value, the state and the auxiliary variables:

$$\alpha\ \mathit{assn} = \mathit{vals} \to \mathit{state} \to \alpha \to \mathit{bool}$$

See §12 for application examples.

For implications between assertions we use the abbreviation

$$P \Rightarrow Q \equiv \forall Y\,\sigma\,Z.\ P\ Y\,\sigma\,Z \longrightarrow Q\ Y\,\sigma\,Z$$

As done here, we typically refer to the result parameter of an assertion as Y, the state as σ, and the auxiliary variables as Z.

4.5. Combinators

In order to keep the axiomatic rules short and thus more readable, we define several assertion (predicate) combinators hiding the state, result, and auxiliary variables in applications as far as possible.

- $\lambda s\,.\,.\ P\ s \equiv \lambda Y\,\sigma.\ P\ (\mathsf{snd}\ \sigma)\ Y\,\sigma$ allows P to peek at the state (without the exception status) 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.
- $f\ .\,;\ P \equiv \lambda Y\,\sigma.\ P\,Y\ (f\,\sigma)$ 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 held for some state σ and the current state is the image of σ under the state transformer f.

Note that the latter two combinators have (almost) inverse effect, in the following sense: $((f\ .\,;\ P)\ ;\,.\ f) \Rightarrow P$ and $P \Rightarrow (f\ .\,;\ (P\ ;\,.\ f))$.

Another group of combinators provides abbreviations for producing and consuming results:

- $P{\leftarrow}w \equiv \lambda Y.\ P\ w$ means that P holds where w is substituted as the result. In successive substitutions the leftmost one prevails: $P{\leftarrow}w{\leftarrow}v = P{\leftarrow}w$.
- $\lambda w\ :\,.\ P\ w \equiv \lambda Y.\ P\ Y\ Y$ peeks at the current result and passes it to P.
- $P{\downarrow} \equiv \lambda Y\,\sigma\,Z.\ \exists Y.\ P\ Y\,\sigma\,Z$ simply ignores the result.
- $P{\downarrow}{=}w \ \equiv \lambda Y\ :\,.\ P{\downarrow} \wedge.\ (\lambda\sigma.\ Y{=}w)$ asserts that the current result is w and then ignores it.

5. Triples

We define Hoare-style *triples* (as usual consisting of a term and two assertions as pre- and postconditions) via a datatype

$(\alpha)\mathit{triple}$

with a single constructor with the mixfix syntax:

$\{\alpha\ \mathit{assn}\}\ \mathit{terms}{\succ}\ \{\alpha\ \mathit{assn}\}$

We give a variant for each class of terms, *viz.* expressions, variables, expression lists, and statements:

$$\begin{array}{l}
\{P\}\ e{-}{\succ}\ \{Q\} \equiv \{P\}\ \mathsf{In1}\ (\mathsf{Inl}\ e){\succ}\ \{Q\}\\
\{P\}\ e{=}{\succ}\ \{Q\} \equiv \{P\}\ \mathsf{In2}\quad\ \ e \succ \{Q\}\\
\{P\}\ e{\doteq}{\succ}\{Q\} \equiv \{P\}\ \mathsf{In3}\quad\ \ e \succ \{Q\}\\
\{P\}\ \ .c.\ \ \{Q\} \equiv \{P\}\ \mathsf{In1}\ (\mathsf{Inr}\ c){\succ}\ \{Q\}
\end{array}$$

In some triples of our Hoare logic rules given below the term parameter will be a (quantifier-free) meta-level expression such as $\mathsf{if}\ b\ \mathsf{then}\ \mathsf{Skip}\ \mathsf{else}\ c$ rather than a pure term of (the abstract syntax of) Java$^{\ell ight}$. This does not hinder the use of such rules since during application these expressions are reduced to pure Java$^{\ell ight}$ terms, in this case to either Skip or c. Another typical example is the expression $\mathsf{body}\ \Gamma\ C\ \mathit{sig}$ which is ultimately replaced by the actual body of the given method.

Concerning the handling of recursive of methods, we take (a variant of) the standard approach: triples appear not only in the consequent, but sets of triples are used as antecedents within the validity and derivability judgments.

$(\alpha)\mathit{triples} = (\alpha)\mathit{triple\ set}$

Furthermore, in order to handle mutual recursion, it is convenient to use sets of triples as (multiple) consequents as well, as further explained in §9.7.2. Semantically speaking, forming sets of triples always means conjunction of its members. Note that for simplicity we allow infinite sets here though only finite sets of triples are derivable.

Actually, the triple type should not have a type parameter and triples should be universally quantified over the type of the auxiliary variables instead: $\forall\alpha.\ \{\alpha\ \mathit{assn}\}\ \mathit{terms}{\succ}\ \{\alpha\ \mathit{assn}\}$. This is not possible in HOL due to the weak (parametric) polymorphism. Thus all members of a set of triples and also the antecedents and consequents as a whole within a Hoare judgment, and thus all triples within a single derivation, are bound to have the same type of auxiliary variables.

6. Validity

6.1. Single Triples

The *validity* of a single triple is a judgment of the form

$\mathit{prog}{\models}\mathit{nat} : (\alpha)\mathit{triple}$

We define partial correctness as

$$\begin{array}{l}\Gamma\models n\!:\{P\}\ t\succ\ \{Q\} \equiv \forall Y\,\sigma\,Z.\ \ P\ Y\,\sigma\,Z\ \longrightarrow\ \mathsf{type\_ok}\ \Gamma\ t\ \sigma \longrightarrow \\ \qquad\qquad\qquad\qquad (\forall Y'\ \sigma'.\ \ \Gamma\vdash\sigma\ \xrightarrow{t:n}\ (Y',\sigma')\ \longrightarrow\ Q\ Y'\ \sigma'\ Z)\end{array}$$

Validity of $\{P\}\ t\succ\ \{Q\}$ (w.r.t. the recursive depth n explained in the next subsection) intuitively means that if P holds for some type-conforming starting state σ and the evaluation of the term t terminates, then Q holds for the result and the final state σ'. Note the universal quantification on the auxiliary variable Z motivated in §4.2.

The predicate

$$\begin{array}{l}\mathsf{type\_ok} :: \mathit{prog} \to \mathit{term} \to \mathit{state} \to \mathit{bool} \\ \mathsf{type\_ok}\ \Gamma\ t\ \sigma \equiv \exists\Lambda.\ (\mathsf{normal}\ \sigma \longrightarrow \exists T.\ (\Gamma,\Lambda)\vdash t{::}T)\wedge \sigma{::}{\preceq}(\Gamma,\Lambda)\end{array}$$

expresses that the term t is well-typed (at least if σ is a normal state) and that all values in σ conform to their static types, both w.r.t. the global environment Γ and some local environment Λ. This additional precondition is required to ensure soundness, as will be discussed in §10.3.

6.2. Recursive Depth

The judgment $\Gamma\vdash\sigma\ \xrightarrow{t:n}\ (Y',\sigma')$ above is a slight refinement of the evaluation judgment given in §3.5. The refinement does not alter the semantics of evaluation, *i.e.* the new parameter n is a mere annotation. It states that evaluation is done with a *recursive depth* bound by n. This notion is required for the proof of soundness and thus will be motivated in §10.2.

The inductive rules defining the extended judgment are exactly the same as in our basic operational semantics, except that the annotation $:n$ is added above the long arrow and the rule for unfolding the method body is replaced by

$$\frac{\Gamma\vdash \mathsf{Norm}\ s_0\ \xrightarrow{\mathsf{body}\ \Gamma\ C\ sig\text{-}{\succ}v\,:\,n}\ \sigma_1}{\Gamma\vdash \mathsf{Norm}\ s_0\ \xrightarrow{\mathsf{Methd}\ C\ sig\text{-}{\succ}v\,:\,n+1}\ \sigma_1}$$

reflecting the increase of the recursive depth due to method calls.

The original and refined versions are equivalent in the following sense:

$$\Gamma\vdash\sigma\ \xrightarrow{t}\ w\sigma'\ \ \mathit{iff}\ \ \exists n.\ \Gamma\vdash\sigma\ \xrightarrow{t:n}\ w\sigma'$$

This can be shown by rule induction for each direction. The 'only if' direction relies on the fact that the recursive depth is monotone:

$$n{\leq}m \longrightarrow \Gamma\vdash\sigma\ \xrightarrow{t:n}\ w\sigma' \longrightarrow \Gamma\vdash\sigma\ \xrightarrow{t:m}\ w\sigma'$$

Note that validity is monotone in the opposite direction:

$$m{\leq}n \longrightarrow \Gamma\models n\!:t \longrightarrow \Gamma\models m\!:t$$

6.3. Liftings

The validity of a single triple canonically carries over to sets of triples:

$$\begin{array}{l}\mathit{prog}||\!\!\models\!\mathit{nat}\!:\!(\alpha)\mathit{triples}\\ \Gamma||\!\!\models\! n\!:\!\mathit{ts} \equiv \forall t{\in}\mathit{ts}.\ \Gamma|\!\!\models\! n\!:\!t\end{array}$$

More interesting is the extension of validity to antecedents, defined as

$$\begin{array}{l}\mathit{prog},(\beta)\mathit{triples}||\!\!\models\!(\alpha)\mathit{triples}\\ \Gamma,A||\!\!\models\!\mathit{ts} \equiv \forall n.\ \Gamma||\!\!\models\! n\!:\!A \longrightarrow \Gamma||\!\!\models\! n\!:\!\mathit{ts}\end{array}$$

meaning that a set of triples ts is valid up to any given recursive depth under the assumption that the set A is valid up to the same depth.

Note the different type parameters for the sets of triples in the antecedent and in the consequent. This emphasizes that they may have different types of auxiliary variables. Unfortunately, due to the restriction mentioned in §5 and the rules asm and $Methd$ (given below) which short-circuit the type variables, antecedents and consequents in the rules are restricted to identical types.

We abbreviate the validity of a single triple under a set of assumptions as

$$\Gamma,A|\!\!\models\! t \equiv \Gamma,A||\!\!\models\!\{t\}$$

Note that our definition for $\Gamma,A||\!\!\models\!\mathit{ts}$ is weaker than the version one might expect, *viz.*

$$(\forall n.\ \Gamma||\!\!\models\! n\!:\!A) \longrightarrow (\forall n.\ \Gamma||\!\!\models\! n\!:\!\mathit{ts})$$

Yet for an empty set of assumptions, both variants are equivalent, and $\Gamma,\emptyset|\!\!\models\! t$ gives the standard notion of validity in the sense that it effectively forgets about the recursive depth:

$$\begin{array}{l}\Gamma,\emptyset|\!\!\models\!\{P\}t\succ\{Q\}\ \ \mathit{iff}\\ \forall n.\ \Gamma||\!\!\models\! n\!:\!\emptyset \longrightarrow \Gamma||\!\!\models\! n\!:\!\{\{P\}t\succ\{Q\}\}\ \ \mathit{iff}\\ \forall n.\ \forall t{\in}\{\{P\}t\succ\{Q\}\}.\ \Gamma|\!\!\models\! n\!:\!t\ \ \mathit{iff}\\ \forall Y\,\sigma\,Z\,Y'\,\sigma'.\ P\,Y\,\sigma\,Z \longrightarrow \mathsf{type\_ok}\ \Gamma\ t\ \sigma \longrightarrow \Gamma\vdash\sigma \xrightarrow{t} (Y',\sigma') \longrightarrow Q\,Y'\,\sigma'\,Z\end{array}$$

The *derivability judgments* have the general form

$$\mathit{prog},(\beta)\mathit{triples}||\!-(\alpha)\mathit{triples}$$

but for the standard case of a single triple in the consequent we use the abbreviation

$$\Gamma,A\vdash t \equiv \Gamma,A||\!-\{t\}$$

7. Structural Rules

As for any Hoare-style logic, there are a number of structural rules applicable for any kind of term: rules for handling antecedents and multiple consequents, exceptions, and the rule of consequence. Many logics involve further rules handling variables and logical connectives within the assertions, but with a strong rule of consequence at hand they are not actually necessary.

Prepared using cpeauth.cls

7.1. Handling Consequents

The first two rules deal with deriving finite sets of triples, which is done one by one, until finally the empty set is reached:

$$\mathit{insert}\ \frac{\Gamma,A\vdash t \quad \Gamma,A|\!\vdash ts}{\Gamma,A|\!\vdash \{t\}\cup ts} \qquad \mathit{empty}\ \frac{}{\Gamma,A|\!\vdash \emptyset}$$

As opposed to introducing sets of triples in the consequent, one may throw away triples using

$$\mathit{weaken}\ \frac{\Gamma,A|\!\vdash ts' \quad ts \subseteq ts'}{\Gamma,A|\!\vdash ts}$$

7.2. Handling Assumptions

Assumptions are introduced using the Methd rule (cf. §9.7.2) and exploited using the following rule:

$$\mathit{asm}\ \frac{ts\subseteq A}{\Gamma,A|\!\vdash ts}$$

There is also a rule for throwing away assumptions, by shrinking the assumption set A to A':

$$(\mathit{thin})\ \frac{\Gamma,A'|\!\vdash ts \quad A' \subseteq A}{\Gamma,A|\!\vdash ts}$$

It does not need to be asserted, but can be derived (with rule induction) from the others [†]. If we had given the simpler but less convenient rule

$$(\mathit{asm}')\ \frac{}{\Gamma,A|\!\vdash A}$$

for exploiting assumptions, the thin rule could no longer be derived.

The cut rule is admissible (*i.e.* valid) but not derivable:

$$(\mathit{cut})\ \frac{\Gamma,A'|\!\vdash ts \quad \Gamma,A|\!\vdash A'}{\Gamma,A|\!\vdash ts}$$

It could be added for convenience, yet we leave it out since it is not strictly necessary for completeness.

7.3. Abrupt Termination

In Java, the concept of *abrupt termination* is used to describe the effects of `break`, `continue` and `return` statements as well as exceptions. Currently we treat only the most important one, exceptions. We provide result expressions to be evaluated at the end of each method body, and the three kinds of statements just mentioned have to be emulated by suitable conditional statements. Yet we plan to support them directly in the near future, as it should be straightforward to handle them using our

[†] Therefore we write its name in parentheses, as we will do for all derived rules.

exception mechanism, which would be an optimization of the approach given in [26]: the type of exceptions *xcpt* just has to be generalized to include the three additional sources of abrupt termination, and suitable catching constructs have to be added to the rules for loops and method calls.

The remainder of this subsection motivates and briefly describes our approach for dealing with exceptions.

Many axiomatic semantics in the literature leave out exceptions and thus cannot infer anything in case of exceptional states. Poetzsch-Heffter and Müller are extending their work [51] to include them. If modeled, exceptions are often given a more exceptional state than they deserve. For instance, in the transition semantics of Drossopoulou and Eisenbach [13] exceptions are regarded as a special form of terms. Thus, a syntactic trick called "expression contexts" has to be used to describe exception propagation in a uniform way. Huisman and Jacobs [26] model the result state of expressions and statements with an outer distinction between hangup, normal completion, and abnormal completion, while giving the store as a parameter where appropriate. This violates the principle of uniformity and thus adds clutter through the omnipresence of case distinctions on the state in their model. Moreover, the axiomatic semantics based on this model uses special kinds of Hoare triples (each with its own version of validity) for reasoning about exceptions, which at least doubles the number of rule variants needed. The different versions of validity have recently been unified [30], but the redundancy within each rule remains. Leino [36] distinguishes between normal and exceptional postconditions. This leads to repetition of (parts of) the assertions needed during program verification.

We deal with exceptions in a very simple and straightforward way: like in our operational semantics, the exception status is part of the program state. Thus it is available within the assertions basically the same way as any variable and heap contents are, and we may re-use predicates on the state like normal for the axiomatic semantics.

In order to describe the propagation of exceptions we use the generic rule

$$Xcpt\ \frac{}{\Gamma{,}A{\vdash}\,\{P{\leftarrow}(\mathsf{arbitrary3}\ t)\ \wedge\!.\ \mathsf{Not} \circ \mathsf{normal}\}\ t{\succ}\ \{P\}}$$

stating that if an exception has occurred, the current command term *t* is ignored (as well as the result entry) and thus the state is not changed. Just a suitable dummy result is generated. All other rules — except for the *Loop* rule — may assume that the initial state is normal (using the predicate transformer Normal), *i.e.* they have the general form

$$\frac{\{\mathsf{Normal}\ P'\}\ t_1{\succ}\ \{Q\} \quad \{Q\}\ t_2{\succ}\ \ldots}{\{\mathsf{Normal}\ P\}\ t{\succ}\ \{R\}}$$

7.4. Rule of Consequence

Kleymann suggests [52, §4.1 and 4.3] a rule of consequence that is stronger than the usual one because it allows to adapt the values of auxiliary variables as required. In the context of recursion this helps to avoid incompleteness and introduction of ad-hoc rules of adaptation. Hofmann [19] gives a rule that is even a bit stronger. After transforming his rule to our setting, simplifying it a bit and adding result value handling, it reads as

$$(conseq12)\ \frac{\begin{array}{c}\Gamma{,}A{\vdash}\,\{P'\}\ t{\succ}\ \{Q'\}\\ \forall Y\,\sigma\,Z\,Y'\,\sigma'.\ P\ Y\,\sigma\,Z \longrightarrow (\forall Y\,Z'.\ P'\ Y\,\sigma\,Z' \longrightarrow Q'\ Y'\ \sigma'\ Z') \longrightarrow Q\ Y'\ \sigma'\ Z\end{array}}{\Gamma{,}A{\vdash}\,\{P\}\ t{\succ}\ \{Q\}}$$

For symmetry, its second premise can be formulated equivalently as

$$\forall \sigma\ Y'\ \sigma'.\ (\forall Y\, Z.\ P'\ Y\ \sigma\ Z \longrightarrow Q'\ Y'\ \sigma'\ Z) \longrightarrow (\forall Y\, Z.\ P\ Y\ \sigma\ Z \longrightarrow Q\ Y'\ \sigma'\ Z)$$

Intuitively, it states that the validity of the triple involving P' and Q' implies the validity of the triple involving P and Q. To this end, the usual conditions $P \Rightarrow P'$ and $Q' \Rightarrow Q$ are sufficient but not necessary.

It will turn out that for completeness we further need (derivatives of) the rule

$$(\mathit{escape})\ \frac{\forall Y\ \sigma\ Z.\ P\ Y\ \sigma\ Z \longrightarrow \Gamma{,}A{\vdash}\{\lambda Y'\ \sigma'\ Z'.\ (Y'{,}\sigma') = (Y{,}\sigma)\}\ t{\succ}\ \{\lambda Y\ \sigma\ Z'.\ Q\ Y\ \sigma\ Z\}}{\Gamma{,}A{\vdash}\{P\}\ t{\succ}\ \{Q\}}$$

We call it *escape* rule since it enables extrusion of the result entry, state, and auxiliary variables from the triple's precondition such that P can be used as an assumption on the meta-logical level governing the rest of the triple. This is essential in particular when dealing with dynamic binding where code depends on the state. Simple consequences of the *escape* rule are the following two:

$$(\mathit{constant})\ \frac{C \longrightarrow \Gamma{,}A{\vdash}\{P\}\ t{\succ}\ \{Q\}}{\Gamma{,}A{\vdash}\{P \wedge .\ (\lambda\sigma.\ C)\}\ t{\succ}\ \{Q\}}$$

$$(\mathit{impossible})\ \frac{}{\Gamma{,}A{\vdash}\{\lambda Y\ \sigma\ Z.\ \mathsf{False}\}\ t{\succ}\ \{Q\}}$$

As already described in [44], we noticed that rather than asserting the rules *conseq12* and *escape*, it is possible to give an even stronger rule of consequence from which these two may be derived:

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

This version is the strongest possible one since it directly reflects the semantics of the pre- and postconditions involved, as can be seen when conducting its soundness proof. Note that it allows choosing the pre- and postconditions of the inner triple, P' and Q', under the assumption P and depending on its parameters Y, σ, and Z.

Common structural rules such as

$$(\mathit{trivial})\ \frac{}{\Gamma{,}A{\vdash}\{P\}\ \ t{\succ}\ \{\lambda Y\ \sigma\ Z.\ \mathsf{True}\}}$$

$$(\mathit{disj})\ \frac{\Gamma{,}A{\vdash}\{P_1\}\ t{\succ}\ \{Q_1\}\quad \Gamma{,}A{\vdash}\{P_2\}\ t{\succ}\ \{Q_2\}}{\Gamma{,}A{\vdash}\{\lambda Y\ \sigma\ Z.\ P_1\ Y\ \sigma\ Z \vee P_2\ Y\ \sigma\ Z\}\ t{\succ}\ \{\lambda Y\ \sigma\ Z.\ Q_1\ Y\ \sigma\ Z \vee Q_2\ Y\ \sigma\ Z\}}$$

are no longer required at all and may be derived easily if desired. In contrast, two other derived rules are quite handy, namely the restriction of the rule of consequence to either the pre- or postcondition:

$$(\mathit{conseq1})\ \frac{\Gamma{,}A{\vdash}\{P'\}\ t{\succ}\ \{Q\}\quad P \Rightarrow P'}{\Gamma{,}A{\vdash}\{P\}\ t{\succ}\ \{Q\}}\qquad (\mathit{conseq2})\ \frac{\Gamma{,}A{\vdash}\{P\}\ t{\succ}\ \{Q'\}\quad Q' \Rightarrow Q}{\Gamma{,}A{\vdash}\{P\}\ t{\succ}\ \{Q\}}$$

8. Universal Quantification

In the rules given in the next section we will use several techniques introducing and exploiting universal quantification around triples in the rules' premises. Their common purpose is to extend the scope in which values are visible in order to reflect dependencies between different pre- and postconditions. The techniques may be nested and combined with each other: for instance, the rule for method calls (cf. §9.7) applies all of them on a single triple. We classify the techniques by the source of the values, as follows.

State Extrusion is of the form $\forall z.\ \Gamma{,}A \vdash \{P\ z \wedge\!.\ (\lambda\sigma.\ z = f\ \sigma)\ \}\ t\ z \succ\ \{Q\ z\}$, which means picking some part of the state (using some function *f*) in the precondition, binding it to a variable z, and using it anywhere within the triple. This technique is useful *e.g.* to save, modify locally to t, and restore local variables. It is very similar to a technique within the MGF approach that will be explained in §11.1, the difference being that there implicit quantification of auxiliary variables is used.

Result Extrusion, already introduced in §4.3, has the general form $\forall z.\ \Gamma{,}A \vdash \{P\ z {\leftarrow} \mathit{Inj}\ z\}\ t\ z \succ\ \{Q\ z\}$ where Inj is one of the injections Val, Var or Vals. It means matching (and binding) the result of the previous triple and using it anywhere within the triple, typically in order to calculate a new result or let the term t depend on it.

Value Passing involves two triples:
$\forall z.\ \Gamma{,}A \vdash \{P\ z\}\ t_1\ z \succ\ \{Q\ z\}$ and $\forall z.\ \Gamma{,}A \vdash \{f\ (Q\ z)\ z\}\ t_2\ z \succ\ \{R\ z\}$ where f is an assertion transformer modifying $Q\ z$. It is logically equivalent to $\forall z.\ \Gamma{,}A \vdash \{P\ z\}\ t_1\ z \succ\ \{Q\ z\} \wedge \Gamma{,}A \vdash \{f\ (Q\ z)\ z\}\ t_2\ z \succ\ \{R\ z\}$ and thus simply extends the scope of z to the second triple. This is useful for passing on previously obtained values to other triples.

An interesting issue is how the bound variables are actually used because this affects the usability and completeness of the resulting Hoare logic. There are again three cases:

State Transformation uses the bound variables to change the state within a pre- or postcondition, typically by applying the assertion transformer $\_\ .;\ \_$.

Result Generation uses the values to compute results to be entered in the result component of a postcondition, typically by applying the assertion transformer $\_ \leftarrow \_$.

Term Dependence means, syntactically speaking, that the term between the pre- and postcondition is an expression actually referring to the bound variables.

The first two uses are harmless: in applications one has to derive triples of the form $\forall z.\ \Gamma{,}A \vdash \{P\ z\}\ t \succ\ \{Q\ z\}$ (where t does not depend on z) for all potential values of z. Concerning the Hoare logic rules, this can be done in a uniform way for some fixed but arbitrary value z because the shape of the triple is the same for all z. Only the proofs of side conditions that possibly emerge when using the rule of consequence might require enhanced proof principles like a (local) induction over z. Yet for these proofs we can assume the full power of predicate logic anyway since we consider relative completeness, as motivated in §4.1.

The third use, dependent terms, is problematic because the triples involved have the form $\forall z.\ \Gamma{,}A \vdash \{P\ z\}\ t\ z \succ\ \{Q\ z\}$ where the term $t\ z$ does depend on z such that uniform rule application is not possible. The only other option for a finitary proof within Hoare logic is to explicitly enumerate all possible cases for z, where of course the variety has to be finite. Often this is easy because the type of z is *bool* (or some other finite type). The other option is to derive from the precondition $P\ z$, typically by applying the $escape$ rule, that only finitely many values for z — commonly even just one — are actually possible. Then the proof proceeds by constructing a finite set (or superset) of the possible values and derive the triple for each of its members where the term expression $t\ z$, as well as the assertions $P\ z$ and $Q\ z$, can be reduced to something not mentioning the variable z anymore.

9. Java-specific Rules

This section contains the main part of our axiomatic semantics, namely the Hoare-style rules for each kind of Java term.

We have designed each rule (except for $Loop$) such that its final postcondition is given by a predicate variable only. Thus application of the rules in the typical “backward-proof” style of Hoare logic is simplified because we avoid the need for applying the rule of consequence in order to adapt the syntactical form of the postcondition, which normally requires awkward explicit instantiations. In other words, the weakest precondition of a given postcondition is typically generated automatically.

9.1. Standard Statements

Thanks to our implicit exception propagation mechanism, the rules for the standard statements appear almost as usual.

$$Skip\ \frac{}{\Gamma{,}A\vdash\{\mathsf{Normal}\ (P{\leftarrow}\bullet)\}\ .\mathsf{Skip}.\ \{P\}}$$

In order to obtain soundness w.r.t. our notion of validity, here (and in a few other rules) we have to mention explicitly the dummy result of statements, $\bullet$. In applications this is no harm since pre- and postconditions of statements do not refer to the result entry anyway.

$$Comp\ \frac{\Gamma{,}A\vdash\{\mathsf{Normal}\ P\}\ .c_1.\ \{Q\}\quad \Gamma{,}A\vdash\{Q\}\ .c_2.\ \{R\}}{\Gamma{,}A\vdash\{\mathsf{Normal}\ P\}\ .c_1\,;c_2.\ \{R\}}$$

For expression statements, the result of the expression is discarded using the $\leftarrow$ operator:

$$Expr\ \frac{\Gamma{,}A\vdash\{\mathsf{Normal}\ P\}\ e{-}{\succ}\ \{Q{\leftarrow}\bullet\}}{\Gamma{,}A\vdash\{\mathsf{Normal}\ P\}\ .\mathsf{Expr}\ e.\ \{Q\}}$$

For terms involving a condition we define the Boolean result substitution operator

$$P{\leftarrow}{=}b \equiv \lambda Y\ \sigma\ Z.\ \exists v.\ (P{\leftarrow}\mathsf{Val}\ v)\ \sigma\ Z \wedge (\mathsf{normal}\ \sigma \longrightarrow \mathsf{the\_Bool}\ v = b)$$

which is a variant of the operator $\leftarrow$ expressing that, unless an exception has been thrown, the result of the preceding Boolean expression is b. Using it in conjunction with the result extrusion technique introduced in §8 and the meta-level conditional expression $\mathsf{if}\ b\ \mathsf{then}\ \_\ \mathsf{else}\ \_$, we can describe both branches of conditional terms with a single triple, like in

$$If\ \frac{\Gamma{,}A\vdash\{\mathsf{Normal}\ P\}\ e{-}{\succ}\ \{P'\}\quad \forall b.\ \Gamma{,}A\vdash\{P'{\leftarrow}{=}b\}\ .(\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2).\ \{Q\}}{\Gamma{,}A\vdash\{\mathsf{Normal}\ P\}\ .\mathtt{if}(e)\ c_1\ \mathtt{else}\ c_2.\ \{Q\}}$$

What is a notational convenience here (to avoid two triples, one for each branch), will be essential for the *Call* rule, given below (cf. §9.7).

Another application of the Boolean substitution for case distinctions is

$$\mathit{Loop}\ \frac{\Gamma,A\vdash\{P\}\ e{-}\!\!\succ\ \{P'\}\quad \Gamma,A\vdash\{\mathsf{Normal}\ (P'\leftarrow{=}\mathsf{True})\}\ .c.\ \{P\}}{\Gamma,A\vdash\{P\}\ .\mathtt{while}(e)\ c.\ \{(P'\leftarrow{=}\mathsf{False}){\downarrow}{=}\bullet\}}$$

The loop body needs to be verified only if no exception has been thrown meanwhile and the Boolean expression e yields True. Upon termination e yields False (unless an exception has occurred). Here both P and P' play the role of the loop invariant, where P' is typically equivalent to P, at least if e does not have side-effects.

Here, in order to achieve completeness, we have to give P rather than Normal P as the precondition of the triple in the rule's conclusion because it should be the same as the invariant P and in general the invariant cannot maintain the absence of exceptions. Furthermore, also for ensuring completeness, here we have to deviate from the principle of having as the final postcondition a predicate variable only. Thus applying the rule of consequence will be necessary here, but this does not cause extra nuisance since finding and inserting suitable invariants P (and P') requires manual engagement anyway.

9.2. Exception Handling

The rule for the `throw` statement modifies the postcondition Q by updating the exception component of the state with the reference just evaluated.

$$\mathit{Throw}\ \frac{\Gamma,A\vdash\{\mathsf{Normal}\ P\}\ e{-}\!\!\succ\ \{\lambda\mathsf{Val}\ a\ \mathtt{:.}\ \mathsf{xupd}\ (\mathsf{throw}\ a)\ \mathtt{.;}\ Q\leftarrow\bullet\}}{\Gamma,A\vdash\{\mathsf{Normal}\ P\}\ .\mathtt{throw}\ e.\ \{Q\}}$$

If no exception has already occurred meanwhile and the evaluated reference a is not Null (in which case a `NullPointer` exception is raised), the auxiliary function throw a assigns the reference to the exception component of the state.

When describing the effect of the statement `try` c_1 `catch`$(C\ vn)\ c_2$ we have to distinguish whether in the state after executing c_1 an exception of appropriate (dynamic) type, *viz.* a subclass of C, is present, as denoted by $\Gamma,\sigma\vdash\mathsf{catch}\ C$. Only if this is the case, the statement c_2 is considered with its exception parameter vn set (using the function new_xcpt_var) to the caught exception. Otherwise, the final postcondition R has to be implied immediately.

$$\mathit{Try}\ \frac{\begin{array}{c}\Gamma,A\vdash\{\mathsf{Normal}\ P\}\ .c_1.\ \{\mathsf{SXAlloc}\ \Gamma\ Q\}\\ \Gamma,A\vdash\{Q\wedge.\ (\lambda\sigma.\ \Gamma,\sigma\vdash\mathsf{catch}\ C)\ \mathtt{;.}\ \mathsf{new\_xcpt\_var}\ vn\}\ .c_2.\ \{R\}\\ (Q\wedge.\ (\lambda\sigma.\ \neg\Gamma,\sigma\vdash\mathsf{catch}\ C))\Rightarrow R\end{array}}{\Gamma,A\vdash\{\mathsf{Normal}\ P\}\ .\mathtt{try}\ c_1\ \mathtt{catch}(C\ vn)\ c_2.\ \{R\}}$$

where

$$\mathsf{SXAlloc}\ \Gamma\ P\equiv\lambda Y\ \sigma\ Z.\ \forall\sigma'.\ \Gamma\vdash\sigma\ \xrightarrow{\mathsf{sxalloc}}\ \sigma'\longrightarrow P\ Y\ \sigma'\ Z$$

The auxiliary relation $\xrightarrow{\mathsf{sxalloc}}$ is a peculiarity of our model used to describe the allocation of exception objects for standard exceptions.

The rule for the `finally` statement needs to transfer the exception status before executing the second substatement to the postcondition where it is combined with the current exception status, producing the final status: if one exception occurs in either clause, it is (re-)raised after the statement, and if both parts throw an exception, the first one takes precedence. In earlier versions of our axiomatic semantics, we employed a special result stack entry to transfer the exception status (denoted by $\mathsf{fst}\,\sigma$ here), but this is not necessary: it suffices to apply the state extrusion technique, as explained in §8.

$$Fin\ \frac{\begin{array}{c}\Gamma,A\vdash\{\mathsf{Normal}\,P\}\ .c_1.\ \{Q\}\\ \forall x.\ \Gamma,A\vdash\{Q\wedge.\ (\lambda\sigma.\ x{=}\mathsf{fst}\,\sigma)\ \texttt{;.}\ \mathsf{xupd}\ (\lambda x.\ \mathsf{None})\}\ .c_2.\\ \{\mathsf{xupd}\ (\lambda x'.\ \mathsf{if}\ x{\neq}\mathsf{None}\wedge x'{=}\mathsf{None}\ \mathsf{then}\ x\ \mathsf{else}\ x')\ \texttt{.;}\ R\}\end{array}}{\Gamma,A\vdash\{\mathsf{Normal}\,P\}\ .c_1\ \texttt{finally}\ c_2.\ \{R\}}$$

9.3. 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 $\mathsf{init}\,C$.

If the class in question is already initialized, there is nothing to do:

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

Otherwise, initialization allocates a new static object using $\mathsf{init\_class\_obj}$, treats the superclass (if any), and finally invokes the static initializers of the class itself, whereby the current local variables are remembered in the bound variable l, hidden during the call to ini (using $\mathsf{set\_lvars\ empty}$), and later restored:

$$Init\ \frac{\begin{array}{c}\mathsf{the}\ (\mathsf{class}\ \Gamma\ C)=(sc,\_,\_,\_,ini)\qquad super=\mathsf{if}\ C=\texttt{Object}\ \mathsf{then\ Skip\ else\ init}\ sc\\ \Gamma,A\vdash\{\mathsf{Normal}\ ((P\wedge.\ \mathsf{Not}\circ\mathsf{initd}\ C)\ \texttt{;.}\ \mathsf{supd}\ (\mathsf{init\_class\_obj}\ \Gamma\ C))\}\ .super.\ \{Q\}\\ \forall l.\ \Gamma,A\vdash\{Q\wedge.\ (\lambda\sigma.\ l=\mathsf{locals}\ (\mathsf{snd}\ \sigma))\ \texttt{;.}\ \mathsf{set\_lvars\ empty}\}\ .ini.\ \{\mathsf{set\_lvars}\ l\ \texttt{.;}\ R\}\end{array}}{\Gamma,A\vdash\{\mathsf{Normal}\ (P\wedge.\ \mathsf{Not}\circ\mathsf{initd}\ C)\}\ .\mathsf{init}\,C.\ \{R\}}$$

Note that the values of sc, ini and $super$ depending on C are known statically and thus application of this rule simplifies immediately.

9.4. Simple Expressions

As already motivated in §4.3, the rule for literal values is

$$Lit\ \frac{}{\Gamma,A\vdash\{\mathsf{Normal}\ (P{\leftarrow}\mathsf{Val}\ v)\}\ \mathsf{Lit}\ v{-}\!\succ\ \{P\}}$$

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. An equivalent but typically less convenient alternative form would be

$$(Lit2)\ \frac{}{\Gamma,A\vdash\{\mathsf{Normal}\,P\}\ \mathsf{Lit}\ v{-}\!\succ\ \{\mathsf{Normal}\ (P{\downarrow}{=}\mathsf{Val}\ v)\}}$$

The rule for `super` is similar, except that one has to peek at the state in order to get the value of This:

$$Super\ \frac{}{\Gamma,A\vdash\{\mathsf{Normal}\ (\lambda s\texttt{..}\ P{\leftarrow}\mathsf{Val}\ (\mathsf{val\_this}\ s))\}\ \texttt{super}{-}\!\succ\ \{P\}}$$

Variable access is the first example where the result entry is a variable. Here we just need its first component, which is the current value of the variable.

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

The rule for variable assignment uses the result extrusion technique to refer to the resulting of *va*.

$$Ass\ \frac{\begin{array}{c}\Gamma{,}A{\vdash}\{\mathsf{Normal}\ P\}\ va{=}\!\succ\ \{Q\}\\ \forall vf.\ \Gamma{,}A{\vdash}\{Q{\leftarrow}\mathsf{Var}\ vf\}\ e{-}\!\succ\ \{\lambda\mathsf{Val}\ v\ \texttt{:.}\ \mathsf{assign}\ (\mathsf{snd}\ vf)\ v\ \texttt{.;}\ R\}\end{array}}{\Gamma{,}A{\vdash}\{\mathsf{Normal}\ P\}\ va\texttt{:=}e{-}\!\succ\ \{R\}}$$

Note that since the value of the assignment is the value of the right-hand side, it is not necessary to give the result substitution $R{\leftarrow}\mathsf{Val}\ v$ in the postcondition of e explicitly. The assignment takes place only if no exception is already present and the update function itself does not throw a new one, as implemented by

$$\begin{array}{l}\mathsf{assign}\ f\ v \equiv \lambda(x{,}s).\ \mathsf{let}\ (x'{,}s') = \mathsf{if}\ x = \mathsf{None}\ \mathsf{then}\ f\ v\ (x{,}s)\ \mathsf{else}\ (x{,}s)\\ \qquad\qquad\quad \mathsf{in}\ \ (x'{,}\mathsf{if}\ x' = \mathsf{None}\ \mathsf{then}\ s'\ \mathsf{else}\ s)\end{array}$$

The rule for conditional expressions parallels the one for conditional statements:

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

A type cast merely evaluates its argument and raises an exception if the dynamic type of the result happens to be unsuitable:

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

Similarly, the type comparison operator flags whether the type of its argument is assignable to the given reference type:

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

9.5. Object Creation

Allocating an object on the heap requires lifting the $\xrightarrow{\mathsf{halloc}}$ relation (allocating a fresh object with the given tag on the heap and initializing it) to the assertion level in analogy to the SXAlloc transformer given above.

$$NewC\ \frac{\Gamma{,}A{\vdash}\{\mathsf{Normal}\ P\}\ .\mathsf{init}\ C.\ \{\mathsf{Alloc}\ \Gamma\ (\mathsf{CInst}\ C)\ Q\}}{\Gamma{,}A{\vdash}\{\mathsf{Normal}\ P\}\ \mathsf{new}\ C{-}\!\succ\ \{Q\}}$$

where

$$\mathsf{Alloc}\ \Gamma\ otag\ P \equiv \lambda Y\ \sigma\ Z.\ \forall\sigma'\ a.\ \Gamma{,}A{\vdash}\sigma\ \xrightarrow{\mathsf{halloc}\ otag\succ a}\ \sigma' \longrightarrow P\ (\mathsf{Val}\ (\mathsf{Addr}\ a))\ \sigma'\ Z$$

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

where

$$\mathsf{check\_neg}\ i\ \equiv \mathsf{raise\_if}\ (\mathsf{the\_Intg}\ i{<}\texttt{0})\ \texttt{NegArrSize}$$

9.6. Variables

The rule for local variables is analogous to the rule for the `super` expression:

$$LVar\ \frac{}{\Gamma,A\vdash\{\mathsf{Normal}\ (\lambda s\texttt{..}\ P\leftarrow\mathsf{Var}\ (\mathsf{lvar}\ vn\ s))\}\ \mathsf{LVar}\ vn{=}\!\succ\ \{P\}}$$

where lvar vn s calculates the semantical description of the local variable vn in the state s.

The rules for field and array variables have a common pattern: calling a variable-generating function vf on the state and applying a given assertion to the resulting variable and state. We capture this in the predicate transformer

$$\begin{array}{l} \_\ \texttt{..;}\ \_ :: (state \to vvar \times state) \to (\alpha)assn \to (\alpha)assn \\ vf\ \texttt{..;}\ P \equiv \lambda Y\,\sigma.\ \mathsf{let}\ (v,\sigma') = vf\ \sigma\ \mathsf{in}\ P\ (\mathsf{Var}\ v)\ \sigma' \end{array}$$

Applying it, we obtain the two rules

$$FVar\ \frac{\begin{array}{c}\Gamma,A\vdash\{\mathsf{Normal}\ P\}\ \mathsf{.init}\ C.\ \{Q\} \\ \Gamma,A\vdash\{Q\}\ e{-}\!\succ\ \{\lambda\mathsf{Val}\ a\texttt{:.}\ \mathsf{fvar}\ C\ stat\ fn\ a\ \texttt{..;}\ R\}\end{array}}{\Gamma,A\vdash\{\mathsf{Normal}\ P\}\ \{C,stat\}e\texttt{..}fn{=}\!\succ\ \{R\}}$$

$$AVar\ \frac{\begin{array}{c}\Gamma,A\vdash\{\mathsf{Normal}\ P\}\ e_1{-}\!\succ\ \{Q\} \\ \forall a.\ \Gamma,A\vdash\{Q\leftarrow\mathsf{Val}\ a\}\ e_2{-}\!\succ\ \{\lambda\mathsf{Val}\ i\texttt{:.}\ \mathsf{avar}\ \Gamma\ i\ a\ \texttt{..;}\ R\}\end{array}}{\Gamma,A\vdash\{\mathsf{Normal}\ P\}\ e_1[e_2]{=}\!\succ\ \{R\}}$$

The auxiliary function fvar C $stat$ fn a calculates the $vvar$ for field fn (which is static iff $stat$ is True) of class C contained in the object at location a. It throws a `NullPointer` exception if necessary. Analogously, avar Γ i a calculates the $vvar$ (cf. §3.2) for the element i of the array at location a for a given state in the context of the program Γ. It may not only throw a `NullPointer`, but also `ArrStore` and `IndOutBound` exception.

9.7. Method Call

A rather complex issue within an axiomatic semantics in general is mutual recursion. For an object-oriented language, dynamic binding in method calls gives a further challenge. This subsection motivates and describes the axiomatic semantics for method calls $\{t,md,mode\}e\texttt{..}mn(\{pTs\}args)$ where e is the receiver expression, mn the method name, $args$ the argument list, $mode$ the invocation mode, and the remaining parts in braces are type annotations: t is the type of e, md the defining class of the method found at compile time, and pTs the list of its parameter types, which is used to resolve static overloading.

9.7.1. Dynamic Binding

Handling dynamic binding for method calls is difficult for two reasons.

First, the actual method to be called depends on the class D dynamically computed from the receiver e and thus in general cannot be inferred statically. The usual technique for dealing with term dependence, as done *e.g.* for the standard Hoare rule for conditional statements, is to statically enumerate all possible values. We cannot use it for D because the variety of possible values is large — but finite because it is bound by the total number of methods in the given program — and not fixed locally since it depends on the class hierarchy. We handle this problem with the state extrusion and term dependence technique introduced in §8, introducing universal quantification for D and the precondition $(\lambda(x,s).\ D = \mathsf{target}\ mode\ s\ a\ md \wedge\ \ldots\ )$ binding D. An alternative solution is given by Poetzsch-Heffter and Müller in [51], where D is referred to via This and the possibly large range of values for D is handled by an abstraction which they call *virtual methods*. Verifying such methods amounts to show (in a cascadic way using two special rules) that all possible implementations fulfil the property required for the method call.

Second, one should be able to assume that for invocation mode `interface` or `virtual` the actual value D is a subtype of t, the (static) type of e. The intuitive — but absolutely non-trivial — reason why the relation $\mathsf{Class}\ D {\preceq} \mathsf{RefT}\ t$ holds is of course type safety. The problem here is how to establish this relation. The rules given in [51], for example, put the burden of verifying the relation on the user, which is a legal option, but in general not practically feasible. In contrast, our solution makes the relation available to the user as a helpful assumption, which transfers the proof burden once and for all to the soundness proof on the meta-level.

We write the subtype relation in the form

$$prog \vdash inv\_mode {\rightarrow} tname {\preceq} ref\_ty$$

and give it the definition

$$\begin{array}{l}\Gamma \vdash mode {\rightarrow} D {\preceq} t \equiv mode = \mathsf{IntVir} \longrightarrow \\ \quad \mathsf{is\_class}\ \Gamma\ D \wedge (\mathsf{if}\ (\exists T.\ t {=} \mathsf{ArrayT}\ T)\ \mathsf{then}\ D {=} \mathtt{Object}\ \mathsf{else}\ \Gamma \vdash \mathsf{Class}\ D {\preceq} \mathsf{RefT}\ t)\end{array}$$

reflecting the knowledge on D required for program verification in case of virtual method invocation.

A minor further complication is that we have to transfer the result a of the expression e not to the triple directly following but to the one after it. We can cope with this by combining the result extrusion and value passing techniques.

The remaining parts of the method call rule 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 universally quantified variable l.

$$Call\ \frac{\begin{array}{c}\Gamma,A \vdash \{\mathsf{Normal}\ P\}\ e {-}{\succ}\ \{Q\} \quad \forall a.\ \Gamma,A \vdash \{Q {\leftarrow} \mathsf{Val}\ a\}\ args {\doteq}{\succ}\ \{R\ a\} \\ \forall a\ vs\ D\ l.\ \Gamma,A \vdash \{(R\ a {\leftarrow} \mathsf{Vals}\ vs\ \ \wedge.\ (\lambda(x,s).\ D = \mathsf{target}\ mode\ s\ a\ md \wedge l = \mathsf{locals}\ s)\ \mathtt{;.} \\ \qquad \mathsf{init\_lvars}\ \Gamma\ D\ (mn,pTs)\ mode\ a\ vs) \wedge.\ (\lambda\sigma.\ \mathsf{normal}\ \sigma \longrightarrow \Gamma \vdash mode {\rightarrow} D {\preceq} t)\} \\ \mathsf{Methd}\ D\ (mn,pTs) {-}{\succ}\ \{\mathsf{set\_lvars}\ l\ \mathtt{.;}\ S\}\end{array}}{\Gamma,A \vdash \{\mathsf{Normal}\ P\}\ \{t,md,mode\}e\mathtt{..}mn(\{pTs\}args) {-}{\succ}\ \{S\}}$$

Note that $\Gamma \vdash mode {\rightarrow} D {\preceq} t$ is asserted only if the current state is normal. Otherwise, an exception occurred evaluating e (or $args$) such that we cannot assume anything about D. For the same reason the well-typedness judgment in the definition of $\mathsf{type\_ok}$ (cf. §6.1) is guarded by $\mathsf{normal}\ \sigma$ since otherwise $\mathsf{Methd}\ D\ (mn,pTs)$ is not necessarily well-typed.

The above rule is general enough to handle all sorts of method calls, also static ones and those relative to interfaces or `super`. One may derive simpler specialized versions of the Call rule, for example for static method calls:

$$(\mathit{CallS})\ \frac{\begin{array}{c}\Gamma,A\vdash\{\mathsf{Normal}\ P\}\ e\!-\!\!\succ\{Q\}\quad \Gamma,A\vdash\{Q\!\downarrow\}\ \mathit{args}\doteq\!\!\succ\{R\}\\ \forall vs\ l.\ \Gamma,A\vdash\{R\leftarrow\mathsf{Vals}\ vs\ \wedge.\ (\lambda\sigma.\ l=\mathsf{locals}\ (\mathsf{snd}\ \sigma))\ \texttt{;.}\ \mathsf{init\_lvars}\ \Gamma\ C\ (mn,pTs)\ \mathsf{Static}\ a\ vs\}\\ \mathsf{Methd}\ C\ (mn,pTs)\!-\!\!\succ\{\mathsf{set\_lvars}\ l\ \texttt{.;}\ S\}\end{array}}{\Gamma,A\vdash\{\mathsf{Normal}\ P\}\ \{t,\mathsf{ClassT}\ C,\mathsf{Static}\}e\texttt{..}mn(\{pTs\}args)\!-\!\!\succ\{S\}}$$

Note that the rule can ignore (applying ↓) the value of e, and the free variable a acts as a dummy parameter of the function $\mathsf{init\_lvars}$.

9.7.2. *Mutual Recursion*

We cope with recursive calls adopting the standard solution of introducing Hoare triples as antecedents within the derivation judgments. Thanks to the well-known recursion rule (see *e.g.* [4, §3.2] and [51]), within an unfolded method body one may appeal to a suitable assumption catering for all further recursive calls.

As discussed in [44], the recursion rule is sufficient for completeness, but its nested application required for mutual recursion in general gives rise to replication of proofs for part of the methods involved. This nuisance can be overcome with rules that allow *simultaneous* rather than nested verification. One such rule is given by Homeier and Martin in [24]. Since they aim at verification condition generation, they designed a rule for verifying all procedures contained in a program simultaneously, which requires the user to identify in advance a single specification for each method suitable to cover all invocation contexts. Our rule can also be used to verify all methods at once, but is more flexible for interactive verification: each time a call to a cluster of mutually recursive procedures is encountered, it allows verifying simultaneously as many (and no more) procedures as desired and to identify the necessary specifications locally.

The Methd rule allows verifying the specifications of a set of methods ms (or to be exact, method implementations) simultaneously by verifying their expansions, that is, the corresponding method bodies [49]. It is vital that when encountering recursive calls during the verification of the bodies, one can assume that the method implementations already fulfil their specifications. This explains why method implementations are separated from method bodies, something that would not be necessary for the underlying operational semantics itself.

$$\mathit{Methd}\ \frac{\Gamma,A\cup\{\{P\}\ \mathsf{Methd}\!-\!\!\succ\{Q\}\mid ms\}\ |\!\!\vdash\ \{\{P\}\ \mathsf{body}\ \Gamma\!-\!\!\succ\{Q\}\mid ms\}}{\Gamma,A|\!\!\vdash\{\{P\}\ \mathsf{Methd}\!-\!\!\succ\{Q\}\mid ms\}}$$

where $\{\{P\}\ \mathit{tf}\!-\!\!\succ\{Q\}\mid ms\}\equiv(\lambda(C,sig).\ \{\mathsf{Normal}\ (P\ C\ sig)\}\ \mathit{tf}\ C\ sig\!-\!\!\succ\{Q\ C\ sig\})\ \text{``}\,ms$
yields a family of method triples indexed by the set ms (consisting of pairs of a class and signature). Both the assertions P and Q and the term function tf depend on the index values given by ms, such that members like
$\{\mathsf{Normal}\ (P\ C\ sig)\}\ \mathsf{Methd}\ C\ sig\!-\!\!\succ\{Q\ C\ sig\}$ are generated.

The structural rules for handling antecedents and sets of triples in the consequent have been described in §7.

The function body (cf. §3.5) and the constructor Body for intermediate body terms have been introduced because the $Methd$ rule above is already complex enough. After the function body Γ has obtained a class name and a signature during applications of the $Methd$ rule, handling the method body is the same as in the operational semantics: the expression body Γ C sig calculates the intermediate term Body D c e with the entries for the defining class of the method, the actual method body, and the result expression. These are then processed sequentially using the rule

$$Body\ \frac{\Gamma{,}A{\vdash}\{\mathsf{Normal}\ P\}\ .\mathsf{init}\ D.\ \{Q\} \quad \Gamma{,}A{\vdash}\{Q\}\ .c.\ \{R\} \quad \Gamma{,}A{\vdash}\{R\}\ e{-}{\succ}\ \{S\}}{\Gamma{,}A{\vdash}\{\mathsf{Normal}\ P\}\ \mathsf{Body}\ D\ c\ e{-}{\succ}\ \{S\}}$$

9.7.3. Virtual Methods, Specifications and Behavioral Subtyping

In our axiomatic semantics, interfaces play only an indirect role. This is due to the fact that in Java they have no semantics other than contributing to the type hierarchy on a purely syntactical basis, *viz.* the availability of methods. Furthermore, Java does not support method specifications nor behavioral subtyping, apparently because this is considered to be too difficult and to limit expressiveness.

From the methodological perspective it would be interesting to take into account behavioral subtyping. Doing so, one introduces specifications for all method declarations. The specification of a method declared within some class or interfaces type t has to be fulfilled by all method bodies implementing or overriding that method within t and any subtype of it. Then a derived rule for method calls can exploit the semantic subtyping as follows. For verifying calls to methods bound by the static type t of the receiver expression (as given in the type annotation of the call) one only has to show that the corresponding specification fulfils the requirements in question, rather than directly showing them for all method implementations in subclasses of t. This approach would be particularly helpful when addressing modularity for proofs: when adding new classes to an already verified program, none of the finished proofs has to be re-done; only the new implementation relations have to be verified.

Note that the “virtual methods” approach mentioned above (cf. §9.7.1) is similar in the sense that method calls and dynamic binding are decoupled, but does not impose the restrictions of behavioral subtyping. Thus this approach in itself cannot support modular verification in the way just given. However, when proof obligations on program extensions are added (in the form of assumptions to be eliminated after the program is freezed), as explained by Müller [39], modular verification is supported in a rather flexible way. A further source of flexibility is that the method specification to be proved for a given call may be generated from the call itself or may be derived from an independent general specification of that virtual method.

Our approach using the $Call$ rule leads to essentially the same verification steps as the “virtual methods” approach and also allows the use of assumptions. Thus at least in principle the methodology developed by [39] can be applied as well, though the “virtual methods” approach supports this more explicitly and conveniently.

9.8. Expression Lists

Lists of expressions are dealt with canonically. Just note that in Isabelle/HOL the empty list is written [] and the ‘cons’ operator is the infix ‘#’.

$$\mathit{Nil}\ \frac{}{\{\textsf{Normal}\ P{\leftarrow}\textsf{Vals}\ []\}\ []\dot{=}\!\succ\ \{P\}}$$

$$\mathit{Cons}\ \frac{\begin{array}{c}\Gamma{,}A{\vdash}\{\textsf{Normal}\ P\}\ e{-}\!\succ\ \{Q\}\\ \forall v.\ \Gamma{,}A{\vdash}\{Q{\leftarrow}\textsf{Val}\ v\}\ es\dot{=}\!\succ\ \{\lambda\textsf{Vals}\ vs:.\ R{\leftarrow}\textsf{Vals}\ (v\#vs)\}\end{array}}{\Gamma{,}A{\vdash}\{\textsf{Normal}\ P\}\ e\#es\dot{=}\!\succ\ \{R\}}$$

9.9. Critical Review

The inductively defined Hoare logic rules given in this section precisely cover the axiomatic semantics of Java$^{\ell ight}$. Unfortunately, they are not easy to read and to apply by hand. The reason for this is the inherent complexity of the language, requiring in particular non-trivial transformations of the state. On the other hand, the format of our rules should facilitate the construction of an automatic verification condition generator, and our experience with example proofs (§12) shows that even for rather complex assertions the theorem proving system deals with proof obligations mostly automatically.

One might also get the impression that our axiomatic semantics is rather close to the operational semantics. We believe that this similarity is not intrinsic but due to our use of the same state transformers as for the operational semantics. We do this for simplicity in conjunction with our semantical notion of assertions, but it should be possible to give a more syntactic notion of assertions and corresponding rules that do not refer to any concept of the operational semantics. The general advantages of a Hoare logic over the operational semantics for program verification, namely concentration on the actually relevant properties of the state and powerful tools for dealing with loops and recursion, are fully realized by our axiomatic semantics.

10. Soundness

A Hoare logic that is unsound would be useless since its very purpose is to verify correctness of programs. Thus after giving a Hoare logic the proof of its soundness is obligatory, in particular when — like in our case — the rules are rather involved and thus their correctness is by far not obvious.

10.1. General Approach

The ultimate goal for proving soundness of our axiomatic semantics w.r.t. the operational semantics is

$$\textsf{wf\_prog}\ \Gamma\ \longrightarrow\ \Gamma{,}\emptyset{\vdash}t\ \longrightarrow\ \Gamma{,}\emptyset{\models}t$$

i.e. any triple t that is derivable from the empty set of assumptions is valid. The additional premise that the program is well-formed is required to show soundness of the method call rule requiring type safety, as explained in §9.7.1 and 10.3.

The soundness goal is a direct instance of

$$\mathsf{wf\_prog}\ \Gamma \longrightarrow \Gamma{,}A|{\vdash}\mathit{ts} \longrightarrow \Gamma{,}A|{\models}\mathit{ts}$$

which can be shown as usual by rule induction on the derivation of $|\vdash$.

The different cases emerging in the induction are basically straightforward, with a few notable exceptions. Since the $Loop$ rule involves a loop invariant rather than unfolding the loop as done for the operational semantics, it requires an auxiliary rule induction on the derivation of the evaluation judgment as contained in the definition of validity.

10.2. Method Implementation Rule

The $Methd$ rule demands special treatment because it adds assumptions about recursive calls, such that an (inductive) argument on the depth of these calls is needed in order to avoid circularities. This could be achieved by syntactic manipulations that unfold procedure calls up to a given depth n, as done *e.g.* in [19]. Instead, we prefer a semantic approach inspired by the proofs given in [51] and [32]: employing the notion of recursive depth already introduced in §6.2.

Induction on the recursive depth boils down to showing in the base case

$$\Gamma{\models}\mathtt{0}{:}\{\mathsf{Normal}\ P\}\ \mathsf{Methd}\ C\ \mathit{sig}{-}{\succ}\ \{Q\}$$

and in the inductive step

$$\Gamma{\models}n{:}\{\mathsf{Normal}\ P\}\ \mathsf{body}\ \Gamma\ C\ \mathit{sig}{-}{\succ}\{Q\} \longrightarrow \Gamma{\models}n{+}\mathtt{1}{:}\{\mathsf{Normal}\ P\}\ \mathsf{Methd}\ C\ \mathit{sig}{-}{\succ}\ \{Q\}$$

10.3. Method Call Rule and Type safety

The $Call$ rule is not only bulky (and thus it helps to treat this case separately from the main rule induction) but also raises major semantical complications. Interestingly, type safety plays a crucial role here: The important fact that for virtual method calls the relation $\Gamma \vdash \mathit{mode}{\to}D{\preceq}\mathit{rt}$ holds, can be derived in general only if the method call is well-typed and the state in which the class D has been dynamically looked up conforms to its environment.

In order to obtain the desired conformance property, one essentially has to keep it as an invariant. But rather than requiring the user to prove this property over and over for each program to be verified, we built it — together with well-typedness — into our notion of validity (cf. §6.1) in the form of the judgment $\mathsf{type\_ok}$. This also gives rise to a new rule:

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

The rule, which will be required for the completeness proof, indicates that if at any time conformance was violated, anything could happen — something that is in line with our intuition on erroneous program execution.

Including conformance (and well-typedness) into validity complicates the proof of soundness, because now we have to show that it is an invariant property of any valid triple, affecting each case of the main rule induction, not only the one for method calls. Fortunately, we have already proved type soundness for the operational semantics, as described *e.g.* in [48]:

$$\begin{array}{l}\mathsf{wf\_prog}\,\Gamma \wedge \Gamma\vdash\sigma \xrightarrow{t} (w,\sigma') \longrightarrow \\ \forall\Lambda\ T.\ \sigma{::}\preceq(\Gamma,\Lambda) \wedge (\Gamma,\Lambda)\vdash t{::}T \longrightarrow \\ \quad \sigma'{::}\preceq(\Gamma,\Lambda) \wedge (\mathsf{normal}\,\sigma' \longrightarrow \Gamma,\mathsf{snd}\,\sigma'\vdash w{::}\preceq T)\end{array}$$

For a well-formed program, if evaluation of a well-formed term terminates and the initial state conforms to its environment, so does the final state, and the result conforms to the type of the term if no exception has been thrown. Making use of this theorem can be simplified — yet not hidden completely — by using for the main induction actually a variant of our validity notion, namely

$$\begin{array}{l}\Gamma\models n{:}\{P\}\ t\succ\ \{Q\} \equiv \forall Y\,\sigma\,Z.\ \ P\ Y\,\sigma\,Z \longrightarrow \forall\Lambda\ T.\ \sigma{::}\preceq(\Gamma,\Lambda) \longrightarrow \\ \qquad\qquad (\mathsf{normal}\,\sigma \longrightarrow (\Gamma,\Lambda)\vdash t{::}T) \longrightarrow \forall Y'\ \sigma'.\ \ \Gamma\vdash\sigma \xrightarrow{\underline{t{:}n}} (Y',\sigma') \longrightarrow \\ \qquad\qquad Q\ Y'\ \sigma'\ Z \wedge \sigma'{::}\preceq(\Gamma,\Lambda)\end{array}$$

One can show easily, exploiting type soundness, that for well-formed programs this variant is equivalent to the original one. Even with this variant, parts of the type soundness proof have to be repeated, *e.g.* to derive well-typedness of the static initializer invoked for class initialization.

11. Completeness

The proof of completeness, stating that the given Hoare logic is useful (at least from the theoretical perspective), is much more challenging than the proof of soundness. We give the outline of the proof in a bit more detail since it is the first such proof for an object-oriented language.

We benefit heavily from the MGF approach which is described below. We extend this approach, which was given for only a single recursive procedure, to mutually recursive methods and static initialization using auxiliary inductions. As discussed in [44], when dealing with mutual recursion some complications arise, which could be overcome in three different ways, each with specific advantages and drawbacks. Here we implement the first two variants involving structural induction that either is nested as deep as the number of methods involved or handles all these methods simultaneously. The third variant, not used here, employs rule induction on the operational semantics, which is more powerful and would save a lot of effort avoiding the auxiliary inductions. On the other hand, it requires an unpleasant unfolding variant of the *Loop* rule and an additional divergence rule. Thus it is probably too tightly connected to the operational semantics, and the employment of rule inductions makes its usability at least doubtful in the light of the proof-theoretical remarks given in §11.5.

Our ultimate goal for proving (relative) completeness is to show that for a well-structured program, any valid triple is derivable from the empty set of assumptions:

$$\mathsf{ws\_prog}\,\Gamma \longrightarrow \Gamma,\emptyset\models t \longrightarrow \Gamma,\emptyset\vdash t$$

The well-known approach involving weakest preconditions $\mathsf{wp}\,t\,Q$ for a given term t and postcondition Q cannot be pursued here, because when verifying recursive method calls the postcondition changes such that structural induction on t does not go through.

11.1. MGF Approach

The *Most General Formula (MGF)* approach was introduced by Gorelick [16] and promoted by Apt [4], Kleymann [32] and others.

For partial correctness, the MGF of a term t gives for the most general precondition (which just remembers the initial state) the strongest postcondition, which is the operational semantics of t. More precisely, the MGF of t in the context of a program Γ is defined as

$$\{\doteq\}\ t\succ\ \{\Gamma\!\rightarrow\}$$

where

$$\begin{array}{lcl} \{P\}\ t\succ\ \{\Gamma\!\rightarrow\} & \equiv & \{P\}\ t\succ\ \{\lambda Y\,\sigma'\,\sigma.\ \Gamma\vdash\sigma \xrightarrow{t} (Y,\sigma')\} \\ \doteq & \equiv & \lambda Y\,\sigma\,Z.\ \sigma = Z \end{array}$$

Note that here the auxiliary variables have the type *state* since they refer to the initial program state. In the precondition the state is stored in Z and retrieved in the postcondition in the form of the bound variable σ. Using it, the postcondition asserts that the result Y and state σ' are exactly those obtained from the initial state by evaluating t. Thus the MGF is trivially valid, and the main task is to show that it is also derivable.

A property of the MGF used often is that $\Gamma,A\vdash\{\mathsf{Normal}\ \doteq\}\ t\succ\ \{\Gamma\!\rightarrow\}$ can be interchanged freely with $\Gamma,A\vdash\{\doteq\}\ t\succ\ \{\Gamma\!\rightarrow\}$, *i.e.* restricting the precondition to normal states is sufficient, because the case of an exceptional state can be always dealt with the Xcpt rule.

The main lemma of the MGF approach is

$$\mathit{MGF\_deriv}\quad \mathsf{ws\_prog}\ \Gamma \longrightarrow \Gamma,\emptyset\vdash\{\doteq\}\ t\succ\ \{\Gamma\!\rightarrow\}$$

Once the derivability of the MGF has been proved, completeness is a rather simple consequence, as follows. We show

$$\Gamma,\emptyset\models\{P\}\ t\succ\ \{Q\} \longrightarrow\ \Gamma,\emptyset\vdash\{\doteq\}\ t\succ\ \{\Gamma\!\rightarrow\}\ \longrightarrow\ \Gamma,\emptyset\vdash\{P\}\ t\succ\ \{Q\}$$

First, we apply the rule

$$(\mathit{no\_hazard})\ \frac{\Gamma,A\vdash\{P\ \wedge.\ \mathsf{type\_ok}\ \Gamma\ t\}\ t\succ\ \{Q\}}{\Gamma,A\vdash\{P\}\ t\succ\ \{Q\}}$$

derived from hazard (cf. §10.3), in order to obtain the extra precondition $\mathsf{type\_ok}\ \Gamma\ t$ which is needed because this judgment is part of our notion of validity. Next, we apply the $\mathit{conseq12}$ rule, and after unfolding the definitions of validity we are left with the proof obligation

$$\begin{array}{l} (\forall n\ Y\,\sigma\,Z.\ (\forall t{\in}\emptyset.\ \Gamma\models n\!:\!t)\ \longrightarrow\ P\ Y\,\sigma\,Z\ \longrightarrow\ \mathsf{type\_ok}\ \Gamma\ t\,\sigma\ \longrightarrow \\ \quad (\forall Y'\,\sigma'.\ \Gamma\vdash\sigma \xrightarrow{t:n} (Y',\sigma')\ \longrightarrow\ Q\ Y'\,\sigma'\,Z))\ \longrightarrow \\ (\forall Y\,\sigma\,Z.\ P\ Y\,\sigma\,Z \wedge \mathsf{type\_ok}\ \Gamma\ t\,\sigma\ \longrightarrow\ (\forall Y'\,\sigma'.\ \Gamma\vdash\sigma \xrightarrow{t} (Y',\sigma') \longrightarrow Q\ Y'\,\sigma'\,Z)) \end{array}$$

which is a rather trivial predicate-logical theorem, exploiting the fact

$$\Gamma\vdash\sigma \xrightarrow{t} (w,\sigma')\ \ \longrightarrow\ (\exists n.\ \Gamma\vdash\sigma \xrightarrow{t:n} (w,\sigma'))$$

The main lemma is proved basically by structural induction. Complications arise because for method calls as well as class initialization the terms involved do not become structurally smaller. To solve the problem we employ auxiliary inductions on the number of methods not yet verified and on the number of classes not yet initialized, as explained in the two following subsections.

11.2. Mutual Recursion

The idea for handling mutual recursion is as follows. First prove derivability of the MGF under the assumption that it has already been proved for all methods:

$$\mathit{MGF\_asm}\quad \begin{array}{l}(\forall C\ sig.\ \textsf{is\_methd}\ \Gamma\ C\ sig \longrightarrow \Gamma{,}A\vdash\{\doteq\}\ \textsf{Methd}\ C\ sig{-}\!\succ\ \{\Gamma\!\rightarrow\})\longrightarrow\\ \Gamma{,}A\vdash\{\doteq\}\ t\!\succ\ \{\Gamma\!\rightarrow\}\end{array}$$

Then prove $\mathit{MGF\_deriv}$ applying the lemma, the Methd rule, which supplies the required assumptions for the methods, and the asm rule for exploiting them. There are two alternatives for collecting the assumptions. Both alternatives rely on the fact that for a well-structured program the number of methods to consider is finite:

$$\mathit{finite\_is\_methd}\quad \textsf{ws\_prog}\ \Gamma \longrightarrow \textsf{finite}\ \{(C{,}sig).\ \textsf{is\_methd}\ \Gamma\ C\ sig\}$$

It is interesting to note that for the whole proof of completeness — in contrast to soundness — well-formedness is not required at all, and the only occasion where we need well-structuredness is to ensure finiteness.

11.2.1. Nested Version

One alternative is to use the classical recursion rule that adds just one assumption per application:

$$\frac{\Gamma{,}A \cup \{\{\textsf{Normal}\ P\}\ \textsf{Methd}\ \ C\ sig{-}\!\succ\ \{Q\}\}\vdash\ \{\textsf{Normal}\ P\}\ \textsf{body}\ \Gamma\ C\ sig{-}\!\succ\ \{Q\}}{\Gamma{,}A\vdash\{\textsf{Normal}\ P\}\ \textsf{Methd}\ C\ sig{-}\!\succ\ \{Q\}}$$

A minor advantage of this version is that it does not require the rules empty and insert for handling sets of triples in the consequents. The main disadvantage is that it requires a complicated scheme for induction on the number of methods not yet considered:

$$\frac{\textsf{finite}\ U \quad uA = (\lambda(C,sig).\ \{\doteq\}\ \textsf{Methd}\ C\ sig{-}\!\succ\ \{\Gamma\!\rightarrow\})\ ``\ U}{\forall A.\ A \subseteq uA \longrightarrow n \leq |uA| \longrightarrow |A| = |uA| - n \longrightarrow (\forall t.\ \Gamma{,}A\vdash\{\doteq\}\ t\!\succ\ \{\Gamma\!\rightarrow\})}$$

which is proved by induction on n. It is applied instantiating U to $\{(C{,}sig).\ \textsf{is\_methd}\ \Gamma\ C\ sig\}$, n to $|U|$, and consequently A to $\emptyset$, yielding the desired result. Note that without finiteness, calculations on cardinality like $|A| = |uA| - n$ would be meaningless.

The induction scheme has been inspired by Hofmann [20]. His lecture notes [19] further contain a proof of completeness using the MGF approach for the language IMP augmented by a single procedure, which we have simplified and extended for our purposes.

11.2.2. Simultaneous Version

We invented the $Methd$ rule that allows handling procedures simultaneously not only in order to simplify applications, but also to make the complicated nesting scheme of the first version dispensable for the meta-theoretic completeness proof. Using its power, the second version becomes rather straightforward. In this case, the cut rule would be convenient, but it can be circumvented by employing the derived rule

$$\frac{F \subseteq U \quad \mathsf{finite}\ U \quad ((\forall (C,sig){\in}F.\ \Gamma,A{\vdash} f\ C\ sig) \longrightarrow (\forall (C,sig){\in}U.\ \Gamma,A{\vdash} g\ C\ sig))}{\Gamma,A|{\vdash}(\lambda(C,sig).\ f\ C\ sig)\ ``F \longrightarrow \Gamma,A|{\vdash}(\lambda(C,sig).\ g\ C\ sig)\ ``F}$$

which is proved by induction on the size of U. On application, both F and U get instantiated to $\{(C,sig).\ \mathsf{is\_methd}\ \Gamma\ C\ sig\}$, so finiteness of the number of methods is vital also here.

11.3. Static Initialization

Now it remains to show $\Gamma,A{\vdash}\{\doteq\}\ t{\succ}\ \{\Gamma{\rightarrow}\}$ under the assumption that the MGFs for all proper methods are derivable:

$$\forall C\ sig.\ \mathsf{is\_methd}\ \Gamma\ C\ sig \longrightarrow \Gamma,A{\vdash}\{\doteq\}\ \mathsf{Methd}\ C\ sig{-}{\succ}\ \{\Gamma{\rightarrow}\}$$

The precondition of the assumption can be discharged because due to the relation $\mathsf{type\_ok}$ in our notion of validity, we get the fact that $\mathsf{Methd}\ C\ sig$ is well-typed for free. Furthermore, all well-typed methods are proper, as follows easily from the corresponding definitions:

$$wt\_Methd\_is\_methd \quad (\Gamma,\Lambda){\vdash}\mathsf{Methd}\ C\ sig{::}{-}T \longrightarrow \mathsf{is\_methd}\ \Gamma\ C\ sig$$

Static initialization requires an induction on the number of classes not yet initialized. To this end we define the auxiliary concepts

$$\begin{array}{l}\mathsf{nyinitcls} :: prog \rightarrow state \rightarrow (tname)set\\ \mathsf{nyinitcls}\ \Gamma\ \sigma \equiv \{C.\ \mathsf{is\_class}\ \Gamma\ C \wedge \neg\ \mathsf{initd}\ C\ \sigma\}\end{array}$$

$$\begin{array}{l}\_{\vdash}\mathsf{init}{\le}\_ :: prog \rightarrow nat \rightarrow state \rightarrow bool\\ \Gamma{\vdash}\mathsf{init}{\le}n \equiv \lambda\sigma.\ |\mathsf{nyinitcls}\ \Gamma\ \sigma| \le n\end{array}$$

$$\begin{array}{l}\{{=}{:}\_\}\ \_{\succ}\ \{\_{\rightarrow}\} :: nat \rightarrow term \rightarrow prog \rightarrow (state)triple\\ \{{=}{:}n\}\ t{\succ}\ \{\Gamma{\rightarrow}\} \equiv \{\doteq \wedge.\ \Gamma{\vdash}\mathsf{init}{\le}n\}\ t{\succ}\ \{\Gamma{\rightarrow}\}\end{array}$$

such that $\mathsf{nyinitcls}\ \Gamma\ \sigma$ is the set of classes not yet initialized, or to be more precise, whose initialization has not yet begun, in state σ. The triple $\{{=}{:}n\}\ t{\succ}\ \{\Gamma{\rightarrow}\}$ is a variant of the MGF with the extra precondition that the number of classes not yet initialized is not greater than n.

$\mathsf{nyinitcls}\ \Gamma\ \sigma$ is finite because it is a subset of the finite set of proper classes. It cannot grow (and thus increase its cardinality) during program execution, and it actually shrinks by one when a class is newly initialized.

Since $\Gamma,A{\vdash}\{\doteq\}\ t{\succ}\ \{\Gamma{\rightarrow}\}$ is equivalent to $\forall n.\ \Gamma,A{\vdash}\{{=}{:}n\}\ t{\succ}\ \{\Gamma{\rightarrow}\}$, it remains to show

$$\forall n\ C\ sig.\ \Gamma,A{\vdash}\{{=}{:}n\}\ \mathsf{Methd}\ C\ sig{-}{\succ}\ \{\Gamma{\rightarrow}\} \longrightarrow (\forall t.\ \Gamma,A{\vdash}\{{=}{:}n\}\ t{\succ}\ \{\Gamma{\rightarrow}\})$$

We can do this by full induction on n, *i.e.* we may assume that $\forall t.\ \Gamma,A{\vdash}\{{=}{:}m\}\ t{\succ}\ \{\Gamma{\rightarrow}\}$ already holds for all smaller m.

11.4. Main Induction

Finally, we have collected enough assumptions such that the main induction will go through. We show

$$\begin{array}{l}(\forall n\; C\; sig.\; \Gamma{,}A{\vdash}\{{=}{:}\,n\}\; \mathsf{Methd}\; C\; sig{-}{\succ}\; \{\Gamma{\rightarrow}\})\;\longrightarrow\\ (\forall m.\; m{<}n \longrightarrow (\forall t.\; \Gamma{,}A{\vdash}\{{=}{:}\,m\}\; t{\succ}\; \{\Gamma{\rightarrow}\}))\;\longrightarrow\\ \quad\Gamma{,}A{\vdash}\{{=}{:}\,n\}\; t{\succ}\; \{\Gamma{\rightarrow}\}\end{array}$$

by structural induction on t. We comment on the most interesting cases.

The first premise is exploited for handling the Methd expression itself and its application in the *Call* case simply by assumption.

We prove the case of the initialization statement as the separate lemma

$$(\forall m.\; m{<}n \longrightarrow \forall t.\; \Gamma{,}A{\vdash}\{{=}{:}\,m\}\; t{\succ}\; \{\Gamma{\rightarrow}\}) \longrightarrow \Gamma{,}A{\vdash}\{{=}{:}\,n\}\; .\mathsf{init}\; C.\; \{\Gamma{\rightarrow}\}$$

because it is needed several times, namely for all terms that involve potential class initialization. When applying the lemma, we can of course make use of the second premise. The actual use of the premise $(\forall m.\; m{<}n \longrightarrow \dots)$ is in the proof of the lemma: for handling the initialization of the superclass and the static initializer of the current class where we know that the current class is actually being initialized and thus the number of classes not yet initialized decreases. This proof is one of the rare cases where we take advantage of the implicit precondition that the current term (init C here) is well-typed and thus C is a proper class.

For the *Loop* case, we employ an alternative formulation of the MGF, augmented by a predicate p in the precondition:

$$\begin{array}{l}\Gamma{,}A{\vdash}\{\mathsf{Normal}\; (\doteq \wedge.\; p)\}\; t{\succ}\; \{\Gamma{\rightarrow}\}\; =\\ \Gamma{,}A{\vdash}\{\mathsf{Normal}\; ((\lambda Y\, \sigma\, Z.\; \forall w\sigma'.\; \Gamma{\vdash}\sigma \xrightarrow{t} w\sigma' \longrightarrow w\sigma' = Z) \wedge.\; p)\}\; t{\succ}\; \{\lambda Y\, s\, Z.\; (Y{,}s) = Z\}\end{array}$$

In the second line the auxiliary variable Z stores the result and final — rather than initial — state, which allows formulating a suitable loop invariant. We make use only of the "if" direction of the above equivalence, which is also the more interesting one: its proof relies on the facts that evaluation is deterministic and that there are at least two different program states.

Note that the alternative version of the MGF has a different type of auxiliary variable, namely $vals \times state$. Therefore, we have to apply the *conseq* and *Loop* rules with rather general types. Yet due to the type restrictions of HOL mentioned in §5, the original rules from the inductive definition are not general enough, and thus we have to state variants of them with identical propositions but generalized types as axioms.

11.5. Proof-theoretical Remarks

One might wonder if the implication proved,

$$\mathsf{ws\_prog}\; \Gamma \;\longrightarrow\; \Gamma{,}\emptyset{\models}t \;\longrightarrow\; \Gamma{,}\emptyset{\vdash}t$$

really means relative completeness, in the sense that for any given concrete program Γ and valid triple t there is a derivation for t, requiring only finitely many applications of the Hoare logic rules plus a complete predicate logic for dealing with side conditions like in the rule of consequence.

One potential problem is the fact that the rules *Methd* and *Init* yield structural expansion of terms rather than reduction of subterms, as already noted and dealt with in the previous subsections.

Furthermore, it is a general property of inductively defined sets S that any proof tree for a theorem $x \in S$ has a finite height but possibly infinite width, for example if the definition of S contains a rule

$$\frac{\forall y.\ f\,y{\in}S}{z{\in}S}$$

where y is of an infinite type. Since in particular for our inductively defined relation $\_,\_|\vdash\_$ there are rules involving universal quantifications on possibly infinite domains, for example parts of the program state, we cannot be sure a priori that any proof of $\Gamma,A|\vdash ts$ is finitary.

Thus we have to ensure ourselves of finiteness by other, more specific means, which we do in two different ways.

- The formalistic answer is that in our proof of completeness we only use the Hoare rules mentioned in §7 and §9, structural induction on terms, and induction on the number of methods and the number of uninitialized classes (both of which are finite). All remaining steps are term rewriting and finitary predicate-logical derivations (involving *e.g.* the rules of the operational semantics in order to derive the postcondition of the MGF), and in particular we do not employ rule induction.
- We additionally give a more constructive answer: a recursive procedure, inspired by our proof of completeness. It generates a proof outline for any (valid) goal $\Gamma,\emptyset\vdash t$, assuming that Γ is well-structured. We sketch the procedure (glossing over applications of the rules $conseq$ and $Xcpt$) and argue informally that this procedure terminates and thus one can conclude that the proof is finitary.
  Assume that $M = \{(C,sig).\ \mathsf{is\_methd}\ \Gamma\ C\ sig\}$ is the set of methods in Γ, the function $spec = \lambda tf\ (C,sig).\ \{(Pre\ C\ sig)\}\ tf\ C\ sig{-}{\succ}\ \{Post\ C\ sig\}$ forms a triple with the (most general) method specification, $MSpecs = spec\ \mathsf{Methd}$ “ M abbreviates the set of all method specifications, and $specb = spec\ (\mathsf{body}\ \Gamma)$ gives the body of a method with its specification. Note that M and thus also $MSpecs$ and $specb$ “ M are finite.
  For showing $\Gamma,\emptyset\vdash t$ we use the following initial proof tree.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma,MSpecs|\vdash MSpecs}\,asm}{\Gamma,MSpecs\vdash specb\ m_1}\ \ \underset{\ldots}{Methd\_asm(MSpecs, specb\ m_i)}\ \ \dfrac{\dfrac{}{\Gamma,MSpecs|\vdash MSpecs}\,asm}{\Gamma,MSpecs\vdash specb\ m_{|M|}}}{\Gamma,MSpecs\vdash specb\ ``M}\ |M|*insert+empty}{\dfrac{\Gamma,\emptyset|\vdash MSpecs}{\Gamma,\emptyset\vdash t}\ Methd\_asm(\emptyset,t)}\ Methd$$

$Methd\_asm(A,t)$ is a proof procedure used for showing $\Gamma,A|\vdash MSpecs \longrightarrow \Gamma,A\vdash t$. The procedure assumes $\Gamma,A|\vdash MSpecs$ and calls the recursive procedure $prove\_triple(t)$. (One could alternatively use the rule cut and a proof procedure $Methd\_asm'(A,t)$ proving $\Gamma,A{\cup}MSpecs\vdash t$.) We define $prove\_triple(\{P\}\ t{\succ}\ \{Q\})$ by case distinction on t.

case $\mathsf{Methd}\ C\ sig$**:** use $weaken$ and the assumption.

case $\mathsf{init}\ C$**:** prove $\Gamma,A\vdash\{P \wedge.\ \mathsf{initd}\ C\}\ t{\succ}\ \{Q\}$ using contradiction or $Done$;
prove $\Gamma,A\vdash\{P \wedge.\ \mathsf{Not} \circ \mathsf{initd}\ C\}\ t{\succ}\ \{Q\}$ trying contradiction first and resort to $Init$ plus recursion only if unsuccessful.

case $\forall b.\ \Gamma,A \vdash \{P' \leftarrow = b\}\ tf\ (\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2) \succ \{Q'\}$**:** expand to both cases True and False and proceed recursively.

case $\forall D\ l.\ \Gamma,A \vdash \{R'\ D\ l\}\ \mathsf{Methd}\ D\ sig {-}\!\!\succ \{S'\ l\}$**:** take a fixed but arbitrary D, exploit the precondition $R'\ D\ l$ in order to restrict the variety of D, and use $weaken$ and the assumption.

all other universal quantifications: proceed recursively for any fixed but arbitrary value since the proof is uniform.

otherwise: use the canonical rule plus recursion.

The procedure $prove\_triple(t)$ terminates (assuming that the proofs of the emerging side conditions terminate) because for all recursive calls the number of quantifiers is reduced or the Java$^{\ell ight}$ terms involved become structurally smaller, except for $\mathsf{init}\,C$ where the number of uninitialized classes is reduced.

12. Example

To illustrate our approach and for gaining experience how our Hoare logic behaves in practice, we use the following (artificial) example.

```
interface HasFoo {
  public Base foo(Base z);
}
class Base implements HasFoo {
  static boolean arr[] = new boolean[2];
  HasFoo vee;
  public Base foo(Base z) {
    return z;
  }
}
class Ext extends Base {
  int vee;
  public Base foo(Base z) {
    ((Ext) z).vee = 1;
    return null;
  }
}

Base e = new Ext();
try {
  e.foo(null);
}
catch(NullPointerException z) {
  while(Ext.arr[2]) ;
}
```

The program fragment consists of three simple but complete type declarations and a block of statements that might occur in any method body that has access to these declarations. All important features of Java^{light} are taken into account.

We can prove that the test program fragment terminates (if at all — this is partial correctness) abruptly with an `IndOutBound` exception:

```
 tprg,∅⊢{Pre} .test. {λY σ Z. fst σ = Some IndOutBound}
where
 Pre ≡ Normal (λY σ Z. heap_free 4 σ ∧ ¬initd Base σ ∧ ¬initd Ext σ)
 tprg ≡ ([(HasFoo,HasFooInt)], [(Base,BaseCl),(Ext,ExtCl)] @ standard_classes)
 arr_viewed_from C ≡ {Base,True}(Cast (Class C) (Lit Null))..arr
 test ≡ e :== new Ext;
        try Expr({CTBase,CTBase,IntVir}!!e..foo({[Class Base]}[Lit Null]))
        catch((SXcpt NullPointer) z)
          (while(Acc (Acc (arr_viewed_from Ext)[Lit (Intg 2)])) Skip)
```

That is, we derive from the empty set of assumptions that if initially there are at least four free locations on the heap and the classes `Base` and `Ext` are not initialized then after termination of the program the exception `IndOutBound` has been thrown. This property relies on a bunch of more or less implicit properties of the program control involving class initialization, dynamic binding, and actual values of method parameters. One could of course prove also other properties like `((Ext)e).vee == 0`.

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, and in the loop condition within the exception handler the static array `arr`, which has been initialized meanwhile, is accessed at a non-existing index.

We make our way through the control flow as directed by the syntax in the usual "backwards" style where the current postcondition is fully known and directs the instantiations of schematic variables typically contained in the precondition mostly automatically. That is, we can apply the syntax-directed Hoare logic rules mostly without making use of the rule of consequence and explicitly instantiating assertions. We have reached this desirable convenience by designing the rules such that the postcondition of the triple in their consequent consists (typically) solely of a free assertion variable that can always be instantiated as required in the application. Yet in a few places explicit instantiations are needed, and sometimes we deliberately use them in order to manually simplify assertions.

This example proof takes about 130 steps, 50 of which are applications of syntax-directed Hoare logic rules. We apply the rule of consequence 13 times and do about 20 explicit instantiations of schematic assertion variables. The simplifier or classical reasoner (or their combination) is called about 40 times. One third of the proof deals with class initialization.

See [47, §6.3.4] for a commented proof outline.

13. Conclusion

13.1. Summary and Experience

Extensions of Hoare logic We have introduced a Hoare logic for a (rather extensive) subset of Java Card. Its design raised interesting general issues like representation of auxiliary variables and side-effecting expressions with their results, and the exact notion of validity. Furthermore, strengthened versions of the consequence and method call rule emerged as well as new solutions for handling exceptions and dependent code.

Unfortunately, many of the rules given are quite complex and thus apparently not easy to use. This is in part due to our semantical notion of assertions better suited for meta-theoretical investigations than for practical use. 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. This complexity inevitably carries over to the axiomatic semantics.

Meta-theoretical Proofs We have proven two main theorems.

Theorem 1. *For well-formed programs our axiomatic semantics is sound w.r.t. its operational counterpart.*

One interesting aspect of the proof of soundness is to find a suitable notion of validity capable of capturing an inductive argument on the recursive depth of procedure calls. The other noticeable thing is the insight that type safety is required to prove correct the rule for handling dynamic binding.

Theorem 2. *For well-structured programs our axiomatic semantics is relatively complete.*

The proof of completeness is non-trivial because next to structural induction it requires nested auxiliary inductions for handling mutual recursion and class initialization. The MGF approach has been successfully refined and applied. Well-formedness and well-typedness play only a minor role here.

Application As experience with our example shows, verifying programs heavily dealing with exceptions and class initialization is tedious, though further machine support might be a relief. Both our formalization and Isabelle itself as the underlying theorem proving system is tailored more towards meta-theory rather than proofs on concrete systems of realistic size. Thus large-scale program verification would require adaptations of the model and extensions to the user interface and proof management, or possibly even transfer of the Hoare logic rules to a specialized integrated program development and verification tool like JIVE [38].

After all, using an axiomatic semantics like ours for program verification helps to concentrate on the interesting properties of a program (rather than fiddling with details of the state as with an operational semantics) and provides powerful tools for dealing with loops and recursion. This general experience carries over from procedural to object-oriented languages like Java.

Machine Support 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 1600 lines of (already quite condensed) proof scripts on a highly complex subject, otherwise there would be plenty of opportunity for omissions and inaccuracies like type errors and inconsistencies. Moreover, the sheer number of definitions to keep in mind and 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. Due to the number of rules in the semantics, in the inductive proofs there are many cases involving a great amount of detail to be considered, for which the partial automation of the theorem prover is of great help.

13.2. Further Work

There are several ways in which this work can — and probably will — be extended and applied.

Extensions of the Model Two important features of the static semantics, name spaces and visibility control, are still missing. It should not be too difficult to add them to the formalization and adapt the proofs accordingly. This is part of the work to be done for the new European Project VerifiCard [29].

Program Verification Our Hoare logic is not yet fully suited for actual application in program verification. It should be helpful — and not difficult — to identify and deduce simpler specialized versions of many rules more convenient to apply in standard, *e.g.* exception-free, situations. Furthermore, better support by tailored verification tools like an automatic verification condition generator and an some advanced methodology for handling *e.g.* method specifications and object references will surely ease the pain.

Support for Program Design Within project Bali [42], work on verifying the implementation of high-level specifications is planned: formalizing the *Object Constraint Language (OCL)* of UML also within Isabelle/HOL and tightly connecting it with our axiomatic semantics.

ACKNOWLEDGEMENTS

This work was done within the Project Bali funded by the DFG. I thank Tobias Nipkow, Arnd Poetzsch-Heffter, Peter Müller, Thomas Kleymann, Martin Hofmann and Francis Tang for inspiring discussions on the topic.
I furthermore thank Tobias Nipkow, Arnd Poetzsch-Heffter, Gerwin Klein, Marieke Huisman and some anonymous referees for their constructive comments on earlier versions of this article.

REFERENCES

1. Martín Abadi and K. Rustan M. Leino. A logic of object-oriented programs. Technical Report SRC-161, Compaq SRC, 1998.
2. Peter Aczel. A system of proof rules for the correctness of iterative programs – some notational and organisational suggestions. Unpublished, 1982.
3. Pierre America and Frank de Boer. A proof theory for a sequential version of POOL. Unpublished?
4. Krzysztof R. Apt. Ten years of Hoare logic: A survey — part I. *ACM Trans. on Prog. Languages and Systems*, 3:431–483, 1981.
5. David Aspinall, Healfdene Goguen, Thomas Kleymann, and Dilip Sequeira. *Proof General*, 1999.
6. Isabelle Attali, Denis Caromel, and Marjorie Russo. Oasis project: Java semantics. `http://www-sop.inria.fr/oasis/java/java_sem.html`.
7. Isabelle Attali, Denis Caromel, and Marjorie Russo. A formal executable semantics for Java. In *OOPSLA'98 Workshop on Formal Underpinnings of Java*, 1998.
8. Hans-Juergen Boehm. Side Effects and Aliasing Can Have Simple Axiomatic Descriptions. *ACM Transactions on Programming Languages and Systems*, 7(4):637–655, 1985.
9. 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/Elsevier, 1992.
10. Alonzo Church. A formulation of the simple theory of types. *J. Symbolic Logic*, 5:56–68, 1940.
11. Stephen A. Cook. Soundness and completeness of an axiom system for program verification. *SIAM Journal on Computing*, 7(1):70–90, 1978.
12. Frank de Boer. A WP-calculus for OO. In *Foundations of Software Science and Computation Structures*, volume 1578 of *Lect. Notes in Comp. Sci.*, pages 135–149. Springer-Verlag, 1999.
13. 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.
14. Michael J. C. Gordon and Thomas F. Melham, editors. *Introduction to HOL: a theorem-proving environment for higher order logic*. Cambridge University Press, 1993.
15. Michael J. C. Gordon, Robin Milner, and C.P. Wadsworth. *Edinburgh LCF: a Mechanised Logic of Computation*, volume 78 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 1979.
16. Gerald A. Gorelick. A complete axiomatic system for proving assertions about recursive and non-recursive programs. Technical Report 75, Department of Computer Science, University of Toronto, 1975.
17. James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.
18. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 1969.
19. Martin Hofmann. Semantik und Verifikation. Lecture notes, in German, 1997.
20. Martin Hofmann and David von Oheimb. Handling mutual recursion. Personal Communication, April 1999.
21. Martin Hofmann and Francis Tang. Implementing a program logic of objects in a higher-order logic theorem prover. In J. Harrison and M. Aagaard, editors, *Theorem Proving in Higher Order Logics: 13th International Conference, TPHOLs 2000*, volume 1869 of *Lect. Notes in Comp. Sci.*, pages 267–282. Springer-Verlag, 2000.
22. The Isabelle/HOL library. `http://isabelle.in.tum.de/library/HOL/`.
23. Peter V. Homeier and David F. Martin. A mechanically verified verification condition generator. *The Computer Journal*, 38:131–141, 1995.
24. Peter V. Homeier and David F. Martin. Mechanical verification of mutually recursive procedures. In M. A. McRobbie and J. K. Slaney, editors, *Proceedings of the 13th Int. Conference on Automated Deduction*, volume 1104 of *Lect. Notes in Comp. Sci.*, pages 201–215. Springer-Verlag, 1996.
25. Marieke Huisman. *Java program verification in Higher-order logic with PVS and Isabelle*. PhD thesis, University of Nijmegen, 2001. `http://www-sop.inria.fr/oasis/personnel/Marieke.Huisman/thesis.ps.gz`.
26. Marieke Huisman and Bart Jacobs. Java program verification via a Hoare logic with abrupt termination. In *Fundamental Approaches to Software Engineering*, volume 1783 of *Lect. Notes in Comp. Sci.*, pages 284–303. Springer-Verlag, 2000.
27. Bart Jacobs, Joachim van den Berg, Marieke Huisman, Martijn van Berkum, Ulrich Hensel, and Hendrik Tews. Reasoning about Java classes (preliminary report). In *ACM Symp. Object-Oriented Programming: Systems, Languages and Applications*, pages 329–340, 1998.
28. Bart Jacobs et al. Loop project. `http://www.cs.kun.nl/˜bart/LOOP/`.
29. Bart Jacobs et al. Project Verificard. `http://www.cs.kun.nl/VerifiCard/`.
30. Bart Jacobs and Eric Poll. A logic for the Java Modeling Language JML. Technical Report CSI-R0018, CSI, 2000. `http://www.cs.kun.nl/csi/reports/info/CSI-R0018.html`.

31. Cliff B. Jones. *Systematic Program Development Using VDM*. International Series in Computer Science. Prentice-Hall, 2nd edition, 1990.
32. Thomas Kleymann. Hoare logic and VDM: Machine-checked soundness and completeness proofs. Ph.D. Thesis, ECS-LFCS-98-392, LFCS, 1998.
33. Tomasz Kowaltowski. Axiomatic approach to side effects and general jumps. *Acta Informatica*, 7:357–360, 1977.
34. Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06l, Department of Computer Science, Iowa State University, 1998. See also `http://www.cs.iastate.edu/~leavens/JML.html`.
35. Gary T. Leavens and William E. Weihl. Subtyping, modular specification, and modular verification for applicative object-oriented programs. Technical Report 92-28d, Department of Computer Science, Iowa State University, 1992. revised 1994.
36. K. Rustan M. Leino. *Toward reliable modular programs*. PhD thesis, California Institute of Technology, 1995. Technical Report CS-TR-95-03.
37. K. Rustan M. Leino, James B. Saxe, and Raymie Stata. Checking Java programs via guarded commands. In B. Jacobs, G.T. Leavens, P. Müller, and A. Poetzsch-Heffter, editors, *Formal Techniques for Java Programs*. Technical Report 251, Fernuniversität Hagen, 1999. `http://gatekeeper.dec.com/pub/DEC/SRC/technical-notes/abstracts/src-tn-1999-002.html`.
38. Jörg Meyer and Arnd Poetzsch-Heffter. An architecture for interactive program provers. In S. Graf and M. Schwartzbach, editors, *TACAS00, Tools and Algorithms for the Construction and Analysis of Systems*, volume 1785 of *Lect. Notes in Comp. Sci.*, pages 63–77, 2000.
39. Peter Müller. *Modular Specification and Verification of Object-Oriented Programs*. PhD thesis, FernUniversität Hagen, 2001. To appear.
40. Peter Müller and Arnd Poetzsch-Heffter. Universes: A type system for controlling representation exposure. In A. Poetzsch-Heffter and J. Meyer, editors, *Programming Languages and Fundamentals of Programming*. Fernuniversität Hagen, 1999. Technical Report 263, `http://www.informatik.fernuni-hagen.de/pi5/publications.html`.
41. Tobias Nipkow. *Isabelle/HOL. The Tutorial*, 1999.
42. Tobias Nipkow, David von Oheimb, Cornelia Pusch, and Gerwin Klein. Project Bali. `http://isabelle.in.tum.de/Bali/`.
43. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. Isabelle's logics: HOL. In *Isabelle: A Generic Theorem Prover*, volume 828 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 1994. Up-to-date version: `http://isabelle.in.tum.de/doc/logics-HOL.pdf`.
44. 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*, volume 1738 of *Lect. Notes in Comp. Sci.*, pages 168–180. Springer-Verlag, 1999. `http://isabelle.in.tum.de/Bali/papers/FSTTCS99.html`.
45. David von Oheimb. Axiomatic semantics for Java$^{\ell ight}$. In S. Drossopoulou, S. Eisenbach, B. Jacobs, G.T. Leavens, P. Müller, and A. Poetzsch-Heffter, editors, *Formal Techniques for Java Programs*. Technical Report 269, 5/2000, Fernuniversität Hagen, Fernuniversität Hagen, 2000. `http://isabelle.in.tum.de/Bali/papers/ECOOP00.html`.
46. David von Oheimb. Axiomatic semantics for Java$^{\ell ight}$ in Isabelle/HOL. Technical Report CSE 00-009, Oregon Graduate Institute, 2000. TPHOLs 2000 Supplemental Proceedings; paper available at `http://isabelle.in.tum.de/Bali/papers/TPHOLs00.html`.
47. David von Oheimb. *Analyzing Java in Isabelle/HOL: Formalization, Type Safety and Hoare Logic*. PhD thesis, Technische Universität München, 2001. `http://www4.in.tum.de/~oheimb/diss/`.
48. 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.* Springer-Verlag, 1999. `http://isabelle.in.tum.de/Bali/papers/Springer98.html`.
49. David L. Parnas. A technique for software module specification with examples. *Communications of the ACM*, 15(5):330–336, 1972.
50. Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 1994. For an up-to-date description, see `http://isabelle.in.tum.de/`.
51. 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 *Lect. Notes in Comp. Sci.*, pages 162–176. Springer-Verlag, 1999.
52. Thomas Schreiber. Auxiliary variables and recursive procedures. In *Theory and Practice of Software Development*, volume 1214 of *Lect. Notes in Comp. Sci.*, pages 697–711. Springer-Verlag, 1997.
53. Christopher Strachey. Fundamental concepts in programming languages. *Higher-Order and Symbolic Computation*, 13(1–2):11–49, 2000. `http://www.wkap.nl/oasis.htm/257993`.
54. Java Card technology. Sun Microsystems, `http://java.sun.com/products/javacard/`, 1999.

55. 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.
56. Markus Wenzel. A formulation of Hoare logic suitable for Isar. `http://isabelle.in.tum.de/library/HOL/Isar_examples/Hoare.html`, 2000.

Index