Hoare Logic for NanoJava: Auxiliary Variables, Side Effects and Virtual Methods revisited

David von Oheimb and Tobias Nipkow
http://isabelle.in.tum.de/Bali/

Fakultät für Informatik, Technische Universität München

Abstract. We define NanoJava, a kernel of Java tailored to the investigation of Hoare logics. We then introduce a Hoare logic for this language featuring an elegant approach for expressing auxiliary variables: by universal quantification on the outer logical level. Furthermore, we give simple means of handling side-effecting expressions and dynamic binding within method calls. The logic is proved sound and (relatively) complete using Isabelle/HOL.

Keywords: Hoare logic, Java, Isabelle/HOL, auxiliary variables, side effects, dynamic binding.

1 Introduction

Java appears to be the first widely used programming language that emerged at a time at which formal verification was mature enough to be actually feasible. For that reason the past few years have seen a steady stream of research on Hoare logics for sequential parts of Java [24, 7, 6, 9, 21, 22], mostly modeled and analyzed with the help of a theorem prover. Since even sequential Java is a formidable language in terms of size and intricacies, there is no Hoare logic for all of it as yet. In terms of language constructs, von Oheimb [22] covers the largest subset of Java. However, as a consequence, this Hoare logic is quite complex and it is difficult to see the wood for the trees. Therefore Nipkow [17] selected some of the more problematic or technically difficult language features (expressions with side effects, exceptions, recursive procedures) and dealt with their Hoare logics in isolation. Although each of these features admits a fairly compact proof system, it remains to demonstrate that their combination in one language is still manageable.

In a sense, NanoJava has been designed with the same aim as Featherweight Java [8]: to have a kernel language for studying a certain aspect of Java. In the case of Featherweight Java, Java's module system is under scrutiny, in NanoJava it is Hoare logic. This explains why, despite of some similarities, we could not just start with Featherweight Java: it was designed for a different purpose; being purely functional, it would not have done us any good.

Starting from μJava [18] we have isolated NanoJava, a Java-like kernel of an object-oriented language. The purpose of this paper is to present the language

NanoJava as a vehicle to convey new techniques for representing Hoare logics (for partial correctness). Next to the Hoare logic we give also an operational semantics such that we can conduct soundness and completeness proofs. Because such proofs have a checkered history in the literature (e.g. the proof system for recursive procedures by Apt [3] was later found to be unsound [2]), the whole development was carried out in the theorem prover Isabelle/HOL [19]. In fact, this very paper is generated from the Isabelle theories, which are documents that can both be machine-checked and rendered in LaTeX. Thus every formula quoted in this paper as a theorem actually is one. The full formalization including all proofs is available online from `http://isabelle.in.tum.de/library/HOL/NavoJava/`.

Our general viewpoint on Hoare logic is that when conducting rigorous analysis (using a theorem prover or not), in particular metatheory, making the dependency of assertions on the program state is indispensable. Furthermore, syntactic treatments of assertions lead to awkward technical complications, namely term substitutions and syntactic side conditions like variable freshness hard to deal with in a fully formal way. Thirdly, for our purposes, constructing and using an assertion language of its own rather than re-using the metalogic for expressing assertions would only add unnecessary clutter. For these reasons, we use a semantic representation of assertions. Doing so, we can in particular replace fresh variables by (universally) bound variables and term substitutions by suitable state transformations.

Compared with previous Hoare logics, in particular fully rigorous ones like [10, 21], we introduce the following technical innovations: auxiliary variables are hidden via universal quantification at the meta-level; side-effecting expressions are treated more succinctly; the treatment of dynamic binding by von Oheimb is combined with the idea of virtual methods (a conceptually important abstraction enabling modular proofs) by Poetzsch-Heffter [24]. The latter technique may be applied to other object-oriented languages as well, whereas the enhanced treatment of side effects and in particular of auxiliary variables applies to Hoare logics for imperative languages in general.

1.1 Related work

Both Huisman and Jacobs [7] and Jacobs and Poll [9] base their work on a kind of denotational semantics of Java and derive (in PVS and Isabelle/HOL) a set of proof rules from it. They deal with many of the complexities of Java's state space, exception handling etc., but without (recursive) method calls. Therefore their rules and ours are quite incomparable. Neither do they investigate completeness. Their rationale is that they can always fall back on the denotational semantics if necessary.

Poetzsch-Heffter and Müller [24] present a Hoare logic for a kernel of Java and prove its soundness. In contrast to our semantic approach to assertions, which is most appropriate for meta theory as our primary concern, they emphasize tool support for actual program verification and use a syntactic approach. This has drawbacks for meta theory, but in the other hand side allows a more or less implicit use of auxiliary variables. More recently, Poetzsch-Heffter [23] has extended this work to a richer language and has also proved completeness. His rules are quite different from ours. In particular he does not use our extended rule of consequence but combines the usual consequence rule with substitution and invariance rules.

Other axiomatic semantics for object-oriented languages include the one by Leino [13], who does not discuss soundness or completeness because he considers only a weakest precondition semantics, and the object-calculus based language by Abadi and Leino [1], who state soundness but suspect incompleteness.

Kleymann [10, 11] gives a machine-checked Hoare logic for an imperative language with a single procedure without parameters, in particular motivating the use of auxiliary variables.

2 NanoJava

We start with an informal exposition of NanoJava. Essentially, NanoJava is Java with just classes. Statements are `skip`, sequential composition, conditional, `while`, assignments to local variables and fields. Expressions are `new`, cast, access to local variables and fields, and method call. The most minimal aspect is the type system: there are no basic types at all, only classes. Consequently there are no literals either; the `null` reference can be obtained because variables and fields are initialized to `null` (and need not be initialized by the programmer). As there are no booleans either, conditionals and loops test references for being `null` (in which case the loop terminates or the `else`-case is taken).

Because of the restriction to references as the only type, it may not be immediately apparent that NanoJava is computationally complete. Figure 1 shows how natural numbers can be simulated by a linked list of references. The natural number n is implemented by a linked list of $n+1$ `Nat`-objects: 0 is implemented by `new Nat()` (initializing `pred` to `null`), and +1 by `suc` (which appends one object to the list). Given two `Nat`-objects `m` and `n`, `m.eq(n)` determines if `m` and `n` represent the same number, and `m.add(n)` adds `m` to `n` non-destructively, i.e. by creating new objects.

3 Abstract syntax

Here we begin with the formal description of NanoJava. We do not show the full formalization in Isabelle/HOL but only the most interesting parts of it.

3.1 Terms

Programs contain certain kinds of names, which we model as members of some not further specified types. For the names of classes, methods, fields and local variables we use the Isabelle types *cname*, *mname*, *fname* and *vname*. It is

```
class Nat {

  Nat pred;

  Nat suc()
    { Nat n = new Nat(); n.pred = this; return n; }

  Nat eq(Nat n)
    { if (this.pred) if (n.pred) return this.pred.eq(n.pred);
                     else return n.pred;
      else if (n.pred) return this.pred; else return this.suc(); }

  Nat add(Nat n)
    { if (this.pred) return this.pred.add(n.suc()); else return n; }
}
```

Fig. 1. Emulating natural numbers

convenient to extend the range of "normal" local variables with special ones holding the *This* pointer, the (single) parameter and the result of the current method invocation, whereby for simplicity we assume that each method has exactly one parameter, called *Par*.

Using the concepts just introduced, we can define statements and expressions as

datatype *stmt*		
= *Skip*		
| *Comp stmt stmt*	(_; _)	
| *Cond expr stmt stmt*	(*If* '(_') _ *Else* _)	
| *Loop vname stmt*	(*While* '(_') _)	
| *LAss vname expr*	(_ = _)	— local assignment
| *FAss expr fname expr*	(_._ = _)	— field assigment
| *Meth cname* × *mname*		— virtual method
| *Impl cname* × *mname*		— method implementation
and *expr*		
= *NewC cname*	(*new* _)	
| *Cast cname expr*		
| *LAcc vname*		— local access
| *FAcc expr fname*	(_._)	— field access
| *Call cname expr mname expr*	({_}_._'(_'))	

The symbols in parentheses on the right hand side specify alternative mixfix syntax for some of the syntactic entities. Virtual methods *Meth* and method implementations *Impl* are intermediate statements that we use for modeling method calls, as will be explained in §4.2. The first subterm C of a method call expression $\{C\}e.m(p)$ is a class name holding the static type of the second subterm e. It will be used as an upper bound for the type dynamically computed during a method call.

3.2 Declarations

The only types we care about are classes and the type of the null pointer:

datatype $ty = NT \mid Class\ cname$

Programs are modeled as lists of class declarations, which contain field and method declarations. The details of their definition and the functions for accessing them are suppressed here as they are of little relevance for the Hoare logic. Also for reasons of space, we gloss over the definitions of type relations as well as the few concepts of well-structuredness required for technical reasons [21, §2.6.4].

Due to our defensive operational semantics given next, the typical notions of well-formedness and well-typedness are not required at all.

4 Operational Semantics

We employ an operational semantics as the primary semantical description of NanoJava. It is more or less standard and thus we can afford to give just the most essential and interesting aspects here.

4.1 Program State

The only values we deal with are references, which are either the null pointer or an address, i.e. a certain location (of some not further specified type *loc*) on the heap:

datatype $val = Null \mid Addr\ loc$

The program state can be thought of as an abstract datatype *state* for storing the values of the local variables (of the current method invocation) as well as the heap, which is essentially a mapping from locations to objects. There are a number of access and modification functions on the state. We typically introduce them on demand, except for two simple ones: $s\langle x\rangle$ stands for the value of the program variable x within state s, and $lupd(x\mapsto v)\ s$ for the state s where the value v has been assigned to the local variable x. The actual definitions of these auxiliary functions are not needed for the meta-theoretic proofs in this paper.

4.2 Evaluation rules

We write $s\ -c-n\rightarrow\ s'$ to denote that execution of the statement c from state s terminates with final state s'. The natural number n gives additional information about program execution, namely an upper bound for the recursive depth. This annotation will be required for the soundness proof of the Hoare logic. The evaluation of an expression e to a value v is written analogously as $s\ -e\succ v-n\rightarrow\ s'$.

Here we give only a selection of the most interesting non-standard execution rules.

For simplicity, and since we do not consider exceptions, we define our semantics in a defensive way: when things go wrong, program execution simply gets stuck. For example, evaluation of a field access *e.f* from an initial state s terminates in state s' if (and only if) the reference expression e evaluates to an address a, transforming the state s into s', and yields the value *get_field* s' a f (which is the contents of field f within the object at heap location a of the state s'):

$$\mathit{FAcc}\colon\ s\ -e{\succ}\mathit{Addr}\ a{-}n{\rightarrow}\ s' \Longrightarrow s\ -e.f{\succ}\mathit{get\_field}\ s'\ a\ f{-}n{\rightarrow}\ s'$$

The most complex rules of our Hoare logic are those concerning method calls. Therefore we give their operational counterparts here first, which should be easier to understand, in order to introduce the basic semantic concepts behind them.

As opposed to the rules for method calls we gave in earlier work, the *Call* rule given here is restricted to argument and result value passing (i.e., the context switch between caller and callee) whereas dynamic binding is handled by the *Meth* rule given thereafter. This not only makes the (still rather formidable) rule a bit simpler, but — more importantly — supports the concept of *virtual methods* [24].

The virtual method *Meth*(C,m) stands for the methods with name m available in class C (and possibly inherited in any subclass) as well as all methods overriding it in any subclass. In other words, the properties of *Meth*(C,m) are the intersection of all properties of method implementations possibly invoked (through dynamic binding) for invocations of m from a reference with static type C. Virtual methods enable not only the usual method specifications from the callee's point of view (involving in particular the local names of the method parameters[1]) but uniform verification of method calls from the caller's view.

$$\begin{aligned}\mathit{Call}\colon \llbracket\ & s0\ -e1{\succ}a{-}n{\rightarrow}\ s1;\ s1\ -e2{\succ}p{-}n{\rightarrow}\ s2;\\ & \mathit{lupd}(\mathit{This}{\mapsto}a)(\mathit{lupd}(\mathit{Par}{\mapsto}p)(\mathit{del\_locs}\ s2))\ -\mathit{Meth}(C{,}m){-}n{\rightarrow}\ s3\ \rrbracket \Longrightarrow\\ & s0\ -\{C\}e1.m(e2){\succ}s3\langle \mathit{Res}\rangle{-}n{\rightarrow}\ \mathit{set\_locs}\ s2\ s3\end{aligned}$$

First a notational remark: in a rule of the form $\llbracket\ A_1;\ A_2;\ \ldots\ A_n\ \rrbracket \Longrightarrow C$, the formulas A_i are the premises and C is the conclusion. After evaluating the reference expression *e1* and the (single) argument expression *e2*, the local variables of the intermediate state *s2* are deleted and the values of the parameter value and the *This* pointer are inserted as new local variables. After the corresponding virtual method has been called, its result is extracted as the value of the method call and the original local variables of *s2* are restored in the final state *s3*. Note that the first parameter of the auxiliary function *set_locs* is the whole state value rather than just the local variable part of it. We decided to do so in order to be able to keep the structure of type *state* opaque.

$$\begin{aligned}\mathit{Meth}\colon \llbracket\ & s\langle \mathit{This}\rangle = \mathit{Addr}\ a;\ D = \mathit{obj\_class}\ s\ a;\ D \preceq_C C;\\ & \mathit{init\_locs}\ (D{,}m)\ s\ -\mathit{Impl}(D{,}m){-}n{\rightarrow}\ s'\ \rrbracket \Longrightarrow\\ & s\ -\mathit{Meth}(C{,}m){-}n{\rightarrow}\ s'\end{aligned}$$

[1] Note that here, as well as in [24], matters are somewhat simplified because the method parameter names are the same for all methods.

Evaluating the virtual method *Meth*(C,m) means extracting the address a of the receiver object (from the *This* pointer), looking up its dynamic type D, which must be a subclass of C, and calling the implementation of m available in class D (after initializing its local variables).

$$\begin{array}{ll} \textit{Impl}\colon & s\ -\textit{body}\ \textit{Cm}{-}n{\rightarrow}\ s' \Longrightarrow \\ & s\ -\textit{Impl}\ \textit{Cm}{-}n{+}1{\rightarrow}\ s' \end{array}$$

The only thing that remains to be done for a method call is to unfold the method body, using a suitable auxiliary function *body* which yields the corresponding (typically compound) statement. Note the increase of the recursive depth from n to $n + (\mathit{1}{::}'a)$. The pair of method class and name is represented by the single free variable *Cm*. One might think of merging the *Impl* and *Meth* rules, but it makes sense to keep virtual methods and method implementations apart in order to separate concerns, which pays off in particular for the axiomatic semantics.

5 Hoare logic concepts

This section and the following one form the core of this article. First, we describe important basic concepts of our axiomatic semantics of NanoJava.

5.1 Assertions

One of the most crucial concepts of a Hoare logic is the notion of *assertions* used as propositions on the program state before and after term execution. The assertion language and its underlying logic strongly determine the expressiveness and completeness of the resulting verification logic.

We take a semantic view of assertions, thus an assertion is nothing but a predicate on the state:

types $\mathit{assn} = \mathit{state} \Rightarrow \mathit{bool}$

This technique, already used in [15], short-circuits the logic used for the assertions with the logic of the underlying proof system, in our case Isabelle/HOL. Thus we do not have to invent a new assertion logic and worry about its expressiveness, and our notion of completeness will be *relative* in the sense of Cook [4].

5.2 Side Effects

Since expressions can have side effects, we need a Hoare logic for expression evaluation as well. For a review of different approaches to this issue, see [22, §4.3]. In essence, assertions used in connection with expressions must be able to refer to the expression results. Kowaltowski [12] shows how to achieve this for a syntactic view on assertions. Since we prefer a semantic view, we can avoid technical complications with substitutions and variable freshness conditions (cf. §6.3), and assertions simply receive the current result as an extra parameter:

types $\mathit{vassn} = \mathit{val} \Rightarrow \mathit{assn}$

Having decided on the assertion types, we can define the Hoare triples for statements and expressions as follows:

types

$$\begin{aligned} \mathit{triple} &= \mathit{assn} \times \mathit{stmt} \times \mathit{assn} \\ \mathit{etriple} &= \mathit{assn} \times \mathit{expr} \times \mathit{vassn} \end{aligned}$$

In the approach that we promoted in [22, §4.3], the type *vassn* is used not only in the postconditions, but also in the preconditions of *etriple*s. This was just for uniformity reasons, and in our definition of validity of such triples given in [22, §6.1], the value parameter was effectively ignored for the precondition. The variant given here is simpler.

5.3 Auxiliary Variables

The most notable novelty and simplification (as compared to the approach promoted by Kleymann [11]) achieved by the work presented here concerns the representation of *auxiliary variables*.

Auxiliary variables are required to relate values between pre- and postconditions. For example, in the triple $\{x{=}Z\}\ y{:=}42\ \{x{=}Z\}$ the (logical) variable Z is used to remember the value of the program variable x, such that one can express that x does not change while y is assigned to. From the logical point of view, Z is implicitly universally quantified, which has to take place somewhere outside the triple such that both occurrences are bound. In case the whole triple occurs negatively (e.g., as an assumption as will be required for handling recursive methods), care is needed to put the quantifier not too far outside.

In [22, §4.2 and §6.1], we followed Kleymann implementing auxiliary variables by extra parameters of assertions that are universally quantified within the definition of validity. Here, in contrast, we leave auxiliary variables and the universal quantifications on them as implicit as possible. That is, auxiliary variables are mentioned in assertions only when they are actually needed, and explicit universal quantification is used only if the concerning triple appears negatively. In other words, we leave the mechanism for dealing with auxiliary variables to the outer logical level.

6 Hoare logic rules

This section gives the full list of Hoare logic rules the constitute the axiomatic semantics of NanoJava.

We write $A \mathrel{||\!\!-} C$ to denote that from the *antecedent* A (acting as a set of assumptions) the consequent C can be derived. Both sets consist of statement triples, i.e. the relation $\_ \mathrel{||\!\!-} \_$ has type (*triple set* $\times$ *triple set*) *set*. If the set C contains just a single triple, we write $A \vdash \{P\}\ c\ \{Q\}$. For expression triples, we use the relation $\_ \vdash_e \_$ of type (*triple set* $\times$ *etriple*) *set*.

6.1 Structural rules

We require only a few structural, i.e. non-syntax-directed rules. The first one is the assumption rule:

Asm: $[\![\ a \in A\ ]\!] \Longrightarrow A \mid\!\vdash \{a\}$

The next two are used to construct and destruct sets in the consequent, i.e. to introduce and eliminate conjunctions on the right-hand side:

ConjI: $[\![\ \forall c \in C.\ A \mid\!\vdash \{c\}\ ]\!] \Longrightarrow A \mid\!\vdash C$

ConjE: $[\![\ A \mid\!\vdash C;\ c \in C\ ]\!] \Longrightarrow A \mid\!\vdash \{c\}$

The final two rules are a further development of the consequence rule due to Morris [14], championed by Kleymann [11], and re-formulated by Nipkow [17].

$$\begin{array}{l} \textit{Conseq}\colon [\![\ \forall Z.\ A \vdash \{P'\ Z\}\ c\ \{Q'\ Z\}; \\ \qquad \forall s\ t.\ (\forall Z.\ P'\ Z\ s \longrightarrow Q'\ Z\ t) \longrightarrow (P\ s \longrightarrow Q\ t)\ ]\!] \Longrightarrow \\ \qquad\quad A \vdash \{P\}\ c\ \{Q\ \} \end{array}$$

$$\begin{array}{l} \textit{eConseq}\colon [\![\ \forall Z.\ A \vdash_e \{P'\ Z\}\ e\ \{Q'\ Z\}; \\ \qquad \forall s\ v\ t.\ (\forall Z.\ P'\ Z\ s \longrightarrow Q'\ Z\ v\ t) \longrightarrow (P\ s \longrightarrow Q\ v\ t)\ ]\!] \Longrightarrow \\ \qquad\quad A \vdash_e \{P\}\ e\ \{Q\ \} \end{array}$$

Within these two rules, as usual the use of auxiliary variables Z partially has to be made explicit in order to allow the adaptation (i.e., specialization with different values) of auxiliary variables for the assertions P' and Q'. The explicit universal quantification on Z in the first premise of each rule is required just because the appearances of Z are on the implication-negative side and thus implicit quantification would be existential rather than universal. Still, due to our new approach to auxiliary variables and side-effecting expressions, these rules are simpler than those given e.g. in [21, 22]: they are closer to the well-known standard form of the consequence rule and require fewer explicit quantifications.

6.2 Standard rules

The rules for standard statements and expressions appear (almost) as usual, except that side effects are taken into account for the condition of *If _ Then _ Else* and for variable assignments. Thus we comment only on those parts of the rules deviating from the standard ones.

Skip: $A \vdash \{P\}\ \textit{Skip}\ \{P\}$

$$\begin{array}{l} \textit{Comp}\colon [\![\ A \vdash \{P\}\ c1\ \{Q\};\ A \vdash \{Q\}\ c2\ \{R\}\ ]\!] \Longrightarrow \\ \qquad A \vdash \{P\}\ c1;\ c2\ \{R\} \end{array}$$

LAcc: $A \vdash_e \{\lambda s.\ P\ (s\langle x\rangle)\ s\}\ \textit{LAcc}\ x\ \{P\}$

The rule for access to a local variable x is reminiscent of the well-known assignment rule: in order to derive the postcondition P, one has to ensure the precondition P where the current value of x is inserted for its result parameter. The lambda abstraction (and later application again) on s is used to peek at the program state.

$LAss$: $[\![\ A \vdash_e \{P\}\ e\ \{\lambda v\ s.\ Q\ (lupd(x{\mapsto}v)\ s)\}\ ]\!] \Longrightarrow$
$\quad A \vdash\ \{P\}\ x{=}e\ \{Q\}$

In the postcondition of the premise of the $LAss$ rule, we can refer to the result of e via the lambda abstraction on v. This value is used to modify the state at the location x before it is fed to the assertion Q.

$Cond$: $[\![\ A \vdash_e \{P\}\ e\ \{Q\};$
$\quad \forall v.\ A \vdash \{Q\ v\}\ (if\ v \neq Null\ then\ c1\ else\ c2)\ \{R\}\ ]\!] \Longrightarrow$
$\quad A \vdash \{P\}\ If(e)\ c1\ Else\ c2\ \{R\}$

The second premise of the $Cond$ rule can handle both branches of the conditional statement uniformly by employing the *if* _ *then* _ *else* _ of the metalogic HOL. This is possible because the statement between the pre- and postcondition is actually a meta-level expression that can depend on the value v of the condition e, as obtained through the precondition $Q\ v$. Note that the universal quantification over v around the triple makes v available throughout the triple, in particular the statement expression in the middle.
This technique for describing the dependency of program terms on previously calculated values, which will be crucial for handling dynamic binding in the *Meth* rule below, has been introduced in [21, §5.5]. If we had standard Booleans, we could expand all possible cases for v (viz. *true* and *false*) and write the *Cond* rule in the more familiar way:
$[\![A \vdash_e \{P\}\ e\ \{Q\};\ A \vdash \{Q\ true\}\ c1\ \{R\};\ A \vdash \{Q\ false\}\ c2\ \{R\}]\!]$
$\Longrightarrow A \vdash \{P\}\ If\ (e)\ c1\ Else\ c2\ \{R\}$

$Loop$: $[\![\ A \vdash \{\lambda s.\ P\ s \wedge s\langle x\rangle \neq Null\}\ c\ \{P\}\ ]\!] \Longrightarrow$
$\quad A \vdash \{P\}\ While(x)\ c\ \{\lambda s.\ P\ s \wedge s\langle x\rangle = Null\}$

The *Loop* rule appears almost as usual except that we consider the loop condition to be fulfilled as long as the given program variable holds a nonzero reference. Allowing for arbitrary side-effecting expressions as loop conditions is possible [22, §9.1] but not worth the technical effort.

6.3 Object-oriented rules

The rules dealing with object-oriented features are less common and therefore deserve more comments. Where needed, we introduce auxiliary functions on the program state on the fly.
$FAcc$: $[\![\ A \vdash_e \{P\}\ e\ \{\lambda v\ s.\ \forall a.\ v{=}Addr\ a \longrightarrow Q\ (get\_field\ s\ a\ f)\ s\}\ ]\!] \Longrightarrow$
$\quad A \vdash_e \{P\}\ e.f\ \{Q\}$

The operational semantics for field access given in §4.2 implies that in order to ensure the postcondition Q it is sufficient that as the postcondition of the reference expression e, Q holds for the contents of the field referred to, under the assumption that the value of e is some address a.

$$\begin{array}{l} \textit{FAss}\text{: } [\![\ A \vdash_e \{P\}\ \textit{e1}\ \{\lambda v\ s.\ \forall a.\ v{=}\textit{Addr}\ a \longrightarrow Q\ a\ s\};\\ \qquad\quad \forall a.\ A \vdash_e \{Q\ a\}\ \textit{e2}\ \{\lambda v\ s.\ R\ (\textit{upd\_obj}\ a\ f\ v\ s)\}\ ]\!] \Longrightarrow\\ \qquad\quad A \vdash\ \{P\}\ \textit{e1.f}{=}\textit{e2}\ \{R\} \end{array}$$

Field assignment handles the value of the reference expression *e1* in the same way except that the address a has to be passed to the triple in the second premise of the rule. We achieve this by passing a as an extra parameter to the intermediate assertion Q and universally quantifying over a around the second triple. This technique, which will be applied also for the *Call* rule below, uses only standard features of the metalogic and is therefore technically less involved than the substitutions to fresh intermediate variables, as employed by Kowaltowski [12]. The auxiliary function *upd_obj a f v s* updates the state s by changing the contents of field f (within the object located) at address a to the value v.

$$\begin{array}{l} \textit{NewC}\text{:}\quad A \vdash_e \{\lambda s.\ \forall a.\ \textit{new\_Addr}\ s = \textit{Addr}\ a \longrightarrow P\ (\textit{Addr}\ a)\ (\textit{new\_obj}\ a\ C\ s)\}\\ \qquad\qquad \textit{new}\ C\ \{P\} \end{array}$$

The rule for object creation uses the functions *new_Addr* selecting a vacant location a on the heap (if possible) as well as *new_obj a C s* which updates the state s by initializing an instance of class C and allocating it at address a. Note that the result of the *new C* expression is the value *Addr a*, given to P as its first argument.

$$\begin{array}{l} \textit{Cast}\text{: } [\![\ A \vdash_e \{P\}\ e\ \{\lambda v\ s.\ (\textit{case}\ v\ \textit{of}\ \textit{Null} \Rightarrow \textit{True}\\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \mid \textit{Addr}\ a \Rightarrow \textit{obj\_class}\ s\ a \preceq_C C) \longrightarrow Q\ v\ s\}\ ]\!] \Longrightarrow\\ \qquad\quad A \vdash_e \{P\}\ \textit{Cast}\ C\ e\ \{Q\} \end{array}$$

In our operational semantics, evaluating a type cast gets stuck if the dynamic type of the object referred to is not a subtype of the given class C. Thus the postcondition Q in the rule's premise needs to be shown only if the value of the reference is a null pointer or the type given by *obj_class s a* is a subclass of C.

$$\begin{array}{l} \textit{Call}\text{: } [\![\ A \vdash_e \{P\}\ \textit{e1}\ \{Q\};\\ \qquad\quad \forall a.\ A \vdash_e \{Q\ a\}\ \textit{e2}\ \{R\ a\};\\ \qquad\quad \forall a\ p\ \textit{ls}.\ A \vdash \{\lambda s'.\ \exists s.\ R\ a\ p\ s \land \textit{ls} = s\ \land\\ \qquad\qquad\qquad\qquad s' = \textit{lupd}(\textit{This}{\mapsto}a)(\textit{lupd}(\textit{Par}{\mapsto}p)(\textit{del\_locs}\ s))\}\\ \qquad\qquad\qquad\qquad \textit{Meth}(C{,}m)\ \{\lambda s.\ S\ (s\langle \textit{Res}\rangle)\ (\textit{set\_locs}\ \textit{ls}\ s)\}\ ]\!] \Longrightarrow\\ \qquad\quad A \vdash_e \{P\}\ \{C\}\textit{e1}.m(\textit{e2})\ \{S\} \end{array}$$

The rule for method calls closely resembles the operational semantics, too. The values of both subexpressions are passed on like in the *FAss* rule above. A third universal quantification is needed in order to transfer the value *ls* of the existentially quantified pre-state s (before the local variables are modified, resulting in state s') to the postcondition of the third triple where the original local variables have to be restored.

$$\begin{array}{l} \textit{Meth}\text{: } [\![\ \forall D.\ A \vdash \{\lambda s'.\ \exists s\ a.\ s\langle \textit{This}\rangle = \textit{Addr}\ a \land D = \textit{obj\_class}\ s\ a \land D \preceq_C C\ \land\\ \qquad\qquad\qquad P\ s \land s' = \textit{init\_locs}\ (D{,}m)\ s\}\ \textit{Impl}(D{,}m)\ \{Q\}\ ]\!] \Longrightarrow\\ \qquad\quad A \vdash \{P\}\ \textit{Meth}(C{,}m)\ \{Q\} \end{array}$$

The rule for virtual methods requires proving the desired property for all method implementations possibly called when taking dynamic binding into account. This is equivalent to the combination of the *class-rule* and *subtype-rule* (plus some structural rules) given in [24], except that the set of implementations is con-

strained not only by the dynamic type D of the *This* pointer, but also by the set of subclasses of C.
This new combination of techniques dealing with dynamic binding has the following advantages. In contrast to the rules given in [24], the variety of possible implementations is captured by a single rule, and the user may immediately exploit the fact that this variety is bound by the statically determined class C. In contrast to the version given in [21, 22], the abstraction of a virtual method allows one to prove the properties of such methods once and for all (independently of actual method calls) and exploiting them for any number of method calls without having to apply the *Meth* rule again.

Method implementations are handled essentially by unfolding the method bodies. Additionally, as usual, the set of assumptions A has to be augmented in order to support recursive calls. For supporting mutual recursion, it is furthermore very convenient to handle a whole set of methods Ms simultaneously [20].

$$\begin{array}{l} [\![\ A \cup\ (\bigcup Z.\ (\lambda Cm.\ (P\ Z\ Cm,\ \mathit{Impl}\ Cm,\ Q\ Z\ Cm))\text{`}Ms)\ \Vdash \\ \qquad\quad (\bigcup Z.\ (\lambda Cm.\ (P\ Z\ Cm,\ \mathit{body}\ Cm,\ Q\ Z\ Cm))\text{`}Ms)\ ]\!] \Longrightarrow \\ \quad A \Vdash (\bigcup Z.\ (\lambda Cm.\ (P\ Z\ Cm,\ \mathit{Impl}\ Cm,\ Q\ Z\ Cm))\text{`}Ms) \end{array}$$

Recall that each $Cm \in Ms$ is a method identifier consisting of a class and method name. For any f, the HOL expression f ‘ Ms denotes the set of all $f\ Cm$ where $Cm \in Ms$. Thus each term $(\lambda Cm.\ (P\ Z\ Cm,\ ...\ Cm,\ Q\ Z\ Cm))\text{`}Ms$ denotes a set of triples indexed by Cm ranging over Ms. The use of auxiliary variables Z has to be made explicit here because the additional assumptions, namely that the implementations involved in any recursive invocations already fulfill the properties to be proved, have to be available for all Z. Within antecedents, this universal quantification can be expressed using the set union operator $\bigcup Z$, and for uniformity we have written the other two quantifications the same way. Note that the semantics of forming a union of sets of Hoare triples is logical conjunction and thus is essentially the same as when using an universal quantifier, which is not possible within the antecedents part of the derivation relation $\Vdash$.
There is a variant of the above rule more convenient to apply. Wherever possible, it uses the standard universal quantifier or even makes quantification implicit:

$$\begin{array}{lll} \mathit{Impl}\text{:}\ [\![\ \forall Z2.\ A \cup (\bigcup Z1. & (\lambda Cm.\ (P\ Z1\ Cm,\ \mathit{Impl}\ Cm,\ Q\ Z1\ Cm))\text{`}Ms) \Vdash & \\ & (\lambda Cm.\ (P\ Z2\ Cm,\ \mathit{body}\ Cm,\ Q\ Z2\ Cm))\text{`}Ms &]\!] \Longrightarrow \\ \qquad\qquad A \quad \Vdash & (\lambda Cm.\ (P\ Z3\ Cm,\ \mathit{Impl}\ Cm,\ Q\ Z3\ Cm))\text{`}Ms & \end{array}$$

7 Example

As an example of a proof in our Hoare logic we formalize (part of) the definition of class `Nat` given in §2 and prove that the virtual method `add` is homomorphic wrt. lower bounds:

$$\{\} \vdash \{\lambda s.\ s{:}s\langle \mathit{This}\rangle \geq X \wedge s{:}s\langle \mathit{Par}\rangle \geq Y\}\ \mathit{Meth}(\mathit{Nat},\mathit{add})\ \{\lambda s.\ s{:}s\langle \mathit{Res}\rangle \geq X+Y\}$$

where the relation $s{:}v \geq n$ means that the value v represents at least the number n, i.e. within state s, the chain of `Nat`-objects starting with v has more than n elements. We consider the lower bound rather than the exact element count in order to avoid problems with non-well-founded chains, e.g. circular ones.

The above proposition is typical for a method specification, in two senses. First, it refers to local variables (including the pointer to the receiver of the method call, the parameter and the result variables) from the perspective of the called method. Second, it makes essential use of auxiliary variables (X and Y, bound at the meta-level): if *This* initially represents at least the number X and *Par* the number Y, then the result of the method represents at least $X + Y$.

The proof consists of 56 user interactions including 32 Hoare logic rule applications (with 6 explicit instantiations of assertion schemes, while all remaining ones are computed by unification). We further make use of many simple properties of the functions acting on the program state and a few properties of the relation $\_:\_ \geq \_$. We comment only on the most interesting of the 32 major steps here. For details see `http://isabelle.in.tum.de/library/HOL/NanoJava/Example.html`.

Steps 1-3 are the application of the *Meth* and *Conseq* rule and the admissible rule $A \vdash \{\lambda s.\ \mathit{False}\}\ c\ \{Q\}$. Dynamic binding is trivial here because `Nat` does not have subclasses. Thus after some cleanup, our goal reduces to

$$\{\} \vdash \{\lambda s.\ s{:}s\langle \mathit{This}\rangle \geq X \wedge s{:}s\langle \mathit{Par}\rangle \geq Y\}\ \mathit{Impl}(\mathit{Nat},\mathit{add})\ \{\lambda s.\ s{:}s\langle \mathit{Res}\rangle \geq X{+}Y\}$$

Step 4 is to apply a derived variant of the *Impl* rule and to unfold the method body, which yields

$$(\bigcup (X,Y).\ \{A\ X\ Y\}) \vdash \{\lambda s.\ s{:}s\langle \mathit{This}\rangle \geq n \wedge s{:}s\langle \mathit{Par}\rangle \geq m\}$$
$$\mathit{If}\ (\mathit{LAcc\ This.pred})\ \mathit{Res} = \{\mathit{Nat}\}\mathit{LAcc\ This.pred.add}(\{\mathit{Nat}\}\mathit{LAcc\ Par.suc}(<>))$$
$$\mathit{Else\ Res} = \mathit{LAcc\ Par}\ \{\lambda s.\ s{:}s\langle \mathit{Res}\rangle \geq n{+}m\}$$

where $A\ X\ Y$ is the single triple appearing as the conclusion of Step 3 which now acts as assumption for any recursive calls of $\mathit{Impl}(\mathit{Nat},\mathit{add})$.

Steps 5-12 simply follow the syntactic structure of the NanoJava terms and arrive at the recursive call of *add*:

$$v \neq \mathit{Null} \Longrightarrow (\bigcup (X,Y).\ \{A\ X\ Y\}) \vdash_e \{\lambda s.\ s{:}s\langle \mathit{This}\rangle \geq n \wedge s{:}s\langle \mathit{Par}\rangle \geq m \wedge$$
$$(\exists\, a.\ s\langle \mathit{This}\rangle = \mathit{Addr}\ a \wedge v = \mathit{get\_field}\ s\ a\ \mathit{pred})\}\ \{\mathit{Nat}\}\mathit{LAcc\ This.pred.}$$
$$\mathit{add}(\{\mathit{Nat}\}\mathit{LAcc\ Par.suc}(<>))\ \{\lambda v\ s.\ \mathit{lupd}(\mathit{Res}{\mapsto}v)\ s{:}v \geq n{+}m\}$$

Steps 13 applies the *Call* rule, giving suitable instantiations for the new intermediate assertions Q and R.

Steps 14-16 deal with evaluating the receiver expression of the call to `add`.

Steps 17-19 apply the *Meth* rule to the second subgoal, which concerns the term $\mathit{Meth}(\mathit{Nat},\mathit{add})$, and after some post-processing this subgoal becomes

$$v \neq \mathit{Null} \Longrightarrow (\bigcup (X,Y).\ \{A\ X\ Y\}) \vdash \{?P\}\ \mathit{Impl}(\mathit{Nat},\mathit{add})\ \{?Q\}$$

where $?P$ and $?Q$ are assertion schemes that may depend on n and m.

Step 20 is the most interesting one of this example: It applies the *Asm* rule after explicitly specializing the universally quantified pair of values $(X,\ Y)$ to $(\mathit{if}\ n{=}0\ \mathit{then}\ 0\ \mathit{else}\ n{-}1,\ m{+}1)$. Recall that n and m are the lower bounds for *This* and *Par* within the current invocation of `add`.

Steps 21-32 deal with evaluating the parameter expression of the call to `add`. This is analogous to the steps before except that no recursion is involved.

8 Equivalence of Operational and Axiomatic Semantics

8.1 Validity

We define validity of Hoare triples with respect to the operational semantics given in §4. The validity of statements is the usual one for partial correctness:

$$\models \{P\}\ c\ \{Q\} \ \equiv\ \forall s\ t.\ P\ s \longrightarrow (\exists n.\ s\ {-c{-}n{\to}}\ t) \longrightarrow Q\ t$$

The improvement here in comparison to [22, §6] is that the references to auxiliary variables Z are not required any more.

The validity of expressions additionally passes the result of the expression to the postcondition:

$$\models_e \{P\}\ e\ \{Q\} \ \equiv\ \forall s\ v\ t.\ P\ s \longrightarrow (\exists n.\ s\ {-e{\succ}v{-}n{\to}}\ t) \longrightarrow Q\ v\ t$$

For the soundness proof we need variants of these definitions where the recursive depth is not existentially quantified over but exported as an extra numerical parameter of the judgments, e.g. $\models n{:}\ (P,\ c,\ Q) \equiv \forall s\ t.\ P\ s \longrightarrow s\ {-c{-}n{\to}}\ t \longrightarrow Q\ t$. This gives rise to the equivalences $\models \{P\}\ c\ \{Q\} = (\forall n.\ \models n{:}\ (P,\ c,\ Q))$ and $\models_e \{P\}\ e\ \{Q\} = (\forall n.\ \models n{:}_e\ (P,\ e,\ Q))$.

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

$$||\!\models n{:}\ T \ \equiv\ \forall t{\in}T.\ \models n{:}\ t$$

Finally, we extend the notion of validity to judgments with sets of statement triples in both the antecedent and consequent:

$$A\ ||\!\models C \ \equiv\ \forall n.\ ||\!\models n{:}\ A \longrightarrow ||\!\models n{:}\ C$$

and analogously to judgments with sets of statement triples in the antecedent and a single expression triple in the consequent:

$$A\ ||\!\models_e t \ \equiv\ \forall n.\ ||\!\models n{:}\ A \longrightarrow \models n{:}_e\ t$$

Note that this handling of antecedents is stronger than the one that might be expected, viz. $(\forall n.\ ||\!\models n{:}\ A) \longrightarrow (\forall n.\ ||\!\models n{:}\ C)$. For an empty set of assumptions A, both variants are equivalent and coincide with the standard notion of validity.

8.2 Soundness

Soundness of the Hoare logic means that all triples derivable are valid, i.e. $\{\} \vdash \{P\}\ c\ \{Q\} \Longrightarrow\ \models \{P\}\ c\ \{Q\}$, and analogously for expressions.

We prove soundness by simultaneous induction on the derivation of $||\!\vdash$ and $||\!\vdash_e$. All cases emerging during the proof are straightforward, except for the *Loop* rule where an auxiliary induction on the derivation of the evaluation judgment is required (in order to handle the loop invariant) and the *Impl* rule where an induction on the recursive depth is required (in order to justify the additional assumptions). The proof takes about 50 steps (user interactions).

For more details of the proof see [22, §10] though matters are simplified here because of our defensive operational semantics. In particular, the evaluation of *Meth* gets stuck if the dynamic type computed from the receiver expression of the method call does not conform to the expected static type. This relieves us from expressing, proving and exploiting type safety for NanoJava, a major endeavor that we had to make for the language(s) given in [21, 22]. [2]

8.3 (Relative) Completeness

Relative completeness of the Hoare logic means that all valid triples are derivable from the empty set of assumptions (if we assume that all valid side conditions can be proved within the meta logic), i.e. $\models \{P\}\ c\ \{Q\} \Longrightarrow \{\} \vdash \{P\}\ c\ \{Q\}$, and analogously for expressions.

We prove this property with the Most General Formula (MGF) approach due to Gorelick [5]. The Most General Triple for a statement c,
$MGT\ c\ Z\ \equiv\ \{\lambda s.\ Z = s\}\ c\ \{\lambda t.\ \exists\, n.\ Z\ -c-n\rightarrow\ t)$
expresses essentially the operational semantics of c: if we bind the initial state using the auxiliary variable Z then the final state t referred to in the postcondition is exactly the one obtained by executing c from Z. The MGF states that the MGT is derivable without assumptions for any Z: $\forall\, Z.\ \{\}\ ||\!\!- \{MGT\ c\ Z\}$. The MGT and MGF for expressions are defined analogously.

In contrast to earlier applications of the MGF approach, in particular [22, §11], here the auxiliary variables Z are bound at the meta level and not within the assertions. This makes the MGFs easier to understand and manipulate.

If we manage to prove the MGF for all terms, relative completeness follows easily by virtue of the consequence rule and the definitions of validity. This has to be done basically by structural induction on the terms. The problem of structural expansion (rather than reduction) during method calls is solved by assuming first that the MGFs for all method implementations are fulfilled. Thus the main effort lies in proving the lemma $\forall\, M\, Z.\ A\ ||\!\!- \{MGT\ (\mathit{Impl}\ M)\ Z\} \Longrightarrow (\forall\, Z.\ A\ ||\!\!- \{MGT\ c\ Z\}) \wedge (\forall\, Z.\ A\ ||\!\!-_e\ MGT_e\ e\ Z)$. Note that the free variables c and e denote any statement or expression, irrespectively if they appear in M or not.

Using the lemma and applying the structural rules *Impl*, *ConjI*, *ConjE* and *Asm*, we can then prove $\{\}\ ||\!\!- \{MGT\ (\mathit{Impl}\ M)\ Z\}$ and from this — and using the lemma again — the MGF and thus relative completeness is straightforward. The proof takes about 100 steps. More detail on the proof as well as some some proof-theoretical remarks may be found in [22, §11].

---

[2] Of course, the theorem prover would assist us in re-using the earlier developments, but still manual adaptations would be required, and it is of course better to avoid technically difficult matters entirely if possible.

9 Concluding remarks

We have presented new solutions for technically difficult issues of Hoare logic and applied them to a Java-like object-oriented kernel language. Although this is unlikely to be the definitive word on the subject, it is a definite improvement over previous such Hoare logics, in particular regarding simplicity and succinctness. This is because we have tuned the logic towards ease of mathematical analysis. Thus we could show that soundness and completeness proofs need not be hard — they can even be machine-checked.

This brings us to a hidden theme of this research: analyzing logics with a theorem prover pays. Although the main benefit usually advertised is correctness (which is indeed an issue in the literature on Hoare logics), we feel the following two points are at least as valuable.

Occam's razor: the difficulty of machine-checked proofs enforces a no-frills approach and often leads to unexpected simplifications.

Incrementality: once you have formalized a certain body of knowledge, incremental changes are simplified: the prover will tell you which proofs no longer work, thus freeing you from the tedium of having to go through all the details once again, as you would have to on paper.

Finally we should comment on how to extend our work from partial to total correctness. Since NanoJava is deterministic, we conjecture that the rules for loops, recursion and consequence by Kleymann [11] should carry over easily. In the presence of unbounded nondeterminism, things become more difficult, but we have already treated this situation in isolation [16] and are confident that it should also carry over easily into an object-oriented context.

References

1. M. Abadi and K. R. M. Leino. A logic of object-oriented programs. In *Theory and Practice of Software Development*, volume 1214 of *Lect. Notes in Comp. Sci.*, pages 682–696. Springer-Verlag, 1997.
2. P. America and F. de Boer. Proving total correctness of recursive procedures. *Information and Computation*, 84:129–162, 1990.
3. K. R. Apt. Ten years of Hoare logic: A survey — part I. *ACM Trans. on Prog. Languages and Systems*, 3:431–483, 1981.
4. S. A. Cook. Soundness and completeness of an axiom system for program verification. *SIAM Journal on Computing*, 7(1):70–90, 1978.
5. G. 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.
6. M. Huisman. *Java program verification in Higher-order logic with PVS and Isabelle.* PhD thesis, University of Nijmegen, 2001.
7. M. Huisman and B. 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.

8. A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, Oct. 1999. Full version in ACM Transactions on Programming Languages and Systems (TOPLAS), 2001.
9. B. Jacobs and E. Poll. A logic for the Java Modeling Language JML. In H. Hussmann, editor, *Fundamental Approaches to Software Engineering*, volume 2029 of *Lect. Notes in Comp. Sci.*, pages 284–299. Springer-Verlag, 2001.
10. T. Kleymann. Hoare logic and VDM: Machine-checked soundness and completeness proofs. Ph.D. Thesis, ECS-LFCS-98-392, LFCS, 1998.
11. T. Kleymann. Hoare logic and auxiliary variables. *Formal Aspects of Computing*, 11:541–566, 1999.
12. T. Kowaltowski. Axiomatic approach to side effects and general jumps. *Acta Informatica*, 7:357–360, 1977.
13. K. R. M. Leino. Ecstatic: An object-oriented programming language with an axiomatic semantics. In *Fourth International Workshop on Foundations of Object-Oriented Programming (FOOL 4)*, 1997.
14. J. Morris. Comments on "procedures and parameters". Undated and unpublished.
15. T. Nipkow. Winskel is (almost) right: Towards a mechanized semantics textbook. In V. Chandru and V. Vinay, editors, *Foundations of Software Technology and Theoretical Computer Science*, volume 1180 of *Lect. Notes in Comp. Sci.*, pages 180–192. Springer-Verlag, 1996.
16. T. Nipkow. Hoare logics for recursive procedures and unbounded nondeterminism. Draft, 2001.
17. T. Nipkow. Hoare logics in Isabelle/HOL. In *Proof and System-Reliability*, 2002.
18. T. Nipkow, D. v. Oheimb, and C. Pusch. μJava: Embedding a programming language in a theorem prover. In F. Bauer and R. Steinbrüggen, editors, *Foundations of Secure Computation*, pages 117–144. IOS Press, 2000. `http://isabelle.in.tum.de/Bali/papers/MOD99.html`.
19. T. Nipkow and L. Paulson. *Isabelle/HOL. The Tutorial*, 2001. `http://isabelle.in.tum.de/doc/tutorial.pdf`.
20. D. v. Oheimb. Hoare logic for mutual recursion and local variables. In C. P. 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`.
21. D. v. 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/`.
22. D. v. Oheimb. Hoare logic for Java in Isabelle/HOL. *Concurrency and Computation: Practice and Experience*, 13(13), 2001. `http://isabelle.in.tum.de/Bali/papers/CPE01.html`.
23. A. Poetzsch-Heffter. Personal communication, Aug. 2001.
24. A. Poetzsch-Heffter and P. Müller. A programming logic for sequential Java. In S. Swierstra, editor, *Programming Languages and Systems (ESOP '99)*, volume 1576 of *Lect. Notes in Comp. Sci.*, pages 162–176. Springer-Verlag, 1999.