Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 176 (2007) 29–46
www.elsevier.com/locate/entcs

Java+ITP: A Verification Tool Based on Hoare Logic and Algebraic Semantics[1]

Ralf Sasse and José Meseguer[2]

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA

Abstract

Java+ITP is an experimental tool for the verification of properties of a sequential imperative subset of the Java language. It is based on an algebraic continuation passing style (CPS) semantics of this fragment as an equational theory in Maude. It supports compositional reasoning in a Hoare logic for this Java fragment that we propose and prove correct with respect to the algebraic semantics. After being decomposed, Hoare triples are translated into semantically equivalent first-order verification conditions (VCs) which are then sent to Maude's Inductive Theorem Prover (ITP) to be discharged. The long-term goal of this project is to use extensible and modular rewriting logic semantics of programming languages, for which CPS axiomatizations are indeed very useful, to develop similarly extensible and modular Hoare logics on which *generic* program verification tools can be based.

Keywords: algebraic semantics; Hoare logic; program verification; Java

1 Introduction

This work is part of a broader effort, namely, the *rewriting logic semantics project*, to which a number of authors are contributing (see the recent surveys [19,18] and references there). The overall goal is to use rewriting logic semantic definitions of programming languages, including concurrent ones, and languages like Maude to generate efficient language implementations, including interpreters and compilers, and also sophisticated program analysis tools for those languages, including invariant checkers for infinite-state programs, model checkers, and theorem provers.

One of the appealing features of all these tools is their *genericity*: by exploiting a common underlying semantics and maximizing the modularity of language definitions it is often possible to generate program analysis tools for different languages in a generic way, using a common infrastructure, yet with competitive performance.

[1] This research has been supported by the ONR Grant N00014-02-1-0715.

[2] Email: {rsasse,meseguer}@cs.uiuc.edu

doi:10.1016/j.entcs.2007.06.006

In the case of interpreters, invariant checkers, and model checkers this has been convincingly demonstrated for many languages, including large subsets of Java and the JVM (see [19,18] for a detailed discussion of different such language case studies).

For the case of theorem provers the situation is less advanced. One unresolved and exciting research issue is finding generic and modular *program logics* in the Hoare style [12], to mathematically justify such logics on the basis of their rewriting logic semantics, and to develop generic theorem proving technology to support reasoning with such program logics in different languages. We are not there yet. In fact, we think that a sound way to approach this quite ambitious goal is to gather empirical evidence through case studies to help us find the outlines of such generic and modular program logics. This paper makes some advances in this direction by focusing on a subset of sequential Java. Specifically we:

(i) Adapt the Maude-based continuation passing style (CPS) rewriting logic semantics for a large fragment of Java given in [7] by adding to it extra features making it suitable for theorem proving purposes. Although we focus for the moment on a modest sequential fragment, there is ample evidence, both in Java and in other languages (see the discussions in [19,18]), supporting the claim that CPS-based rewriting logic definitions are modular and extensible; therefore, we believe that our present work will naturally extend to more ambitious language fragments in Java and in other languages.

(ii) Develop a Hoare logic for this fragment and mathematically justify the correctness of our Hoare rules based on the CPS semantics. Even for this modest fragment this turns out to be nontrivial, because some of the standard Hoare rules, including the rules for conditionals and for while loops, are in fact *invalid* and have to be properly generalized in order to be applicable to Java programs.

(iii) Develop a mechanization of this Hoare logic supporting: (i) compositional reasoning with the Hoare rules to decompose Hoare triples into simpler ones; (ii) generation of first-order *verification conditions* (VCs); and (iii) discharging of such VCs by Maude's inductive theorem prover (ITP) [4] using the underlying CPS semantics. Java+ITP has been developed as an extension of Maude's ITP and is entirely written in Maude.

Although Java+ITP is primarily a research vehicle to help us advance the longer-term goal of developing generic logics of programs and generic program verifiers based on modular rewriting logic semantic definitions, we have also found it quite useful as a *teaching tool* at the University of Illinois at Urbana-Champaign to teach graduate students and seniors the essential ideas of algebraic semantics and Hoare logic. It has been used quite extensively by students on a graduate course on Program Verification (CS 476) and will also be used this Winter on a Formal Methods graduate course (CS 477).

The conceptual basis of Java+ITP is exactly what one would expect of any theorem proving tool based on a language's rewriting logic semantics. As already mentioned, the CPS semantics of our Java fragment is axiomatized in Maude. Since we focus for the moment on a sequential fragment, this defines an *equational theory*

JAVAX. Therefore, the language's *mathematical semantics* is precisely the *initial algebra* $T_{\texttt{JAVAX}}$. We use this mathematical model $T_{\texttt{JAVAX}}$ to justify the semantics of our Hoare rules. Similarly, the first-order VCs associated to Hoare triples are then *inductive goals* that are claimed to be satisfied by the initial model $T_{\texttt{JAVAX}}$, and that Maude's ITP tries to discharge using the equational theory JAVAX. Therefore, for this fragment we are within the well known *algebraic semantics* framework [10]; however, in future extensions including threads and concurrency, the semantics will instead be given by a rewrite theory, and the inductive reasoning will be based on the initial model of such a rewrite theory.

There is a substantial body of related work on Java logics, semantics and theorem proving tools, such as, for example, [16,14,13,15,9,20,17,2,3]. We discuss this related work in Section 6; we also discuss there work closer to ours such as the Maude ITP [4], on which our tool is based, the ASIP-ITP tool [6,22], and of course the JavaFAN project [7,8], to which this work contributes at the theorem proving level. The rest of the paper is organized as follows. The CPS semantics of our Java fragment is summarized in Section 2. The first-order semantics of Hoare triples based on the initial algebra semantics of the language is explained in Section 3. Our Hoare logic and its justification are treated in Section 4. The mechanization of such a logic in the Java+ITP tool, and its use in examples are discussed in Section 5. Section 6 treats related work and conclusions. The related technical report [24] contains a mathematical proof of correctness for the loop rule, and two proof scripts for Java programs.

2 Algebraic Semantics of a Sequential Java Subset

We present some of the highlights of the semantics of our chosen Java subset. We do not show the whole syntax, state infrastructure and actual semantics because of space limitations. However, the whole definition is available on the web at [23]. The Java fragment we are interested in includes arithmetic expressions, assignments, sequential composition and loops. Our semantics uses a continuation passing style (CPS) approach. This has the advantage of making our semantic definitions easily extensible to accommodate additional Java features in the future. For example, exceptions, objects, multi-threading and all other Java features can be expressed using a CPS style as shown by the prototype version in [7]. Our specification is similar in style to the prototype interpreter for a much bigger Java subset in [7], but has some differences/optimizations that take advantage of the sequential nature of our chosen subset. We illustrate our semantics by making explicit its state infrastructure and showing the syntax and semantics for a few selected features .

2.1 The State Infrastructure for Java

To be able to describe the *semantics* of Java we must specify how the *execution* of programs affects the *state infrastructure*, which contains the values for the program variables and other state information. The state infrastructure is defined by the following modules, where we separately specify the locations, environments, values,

stores and continuations that make up the state.

A program variable will not be directly mapped to its value but to a *location* in the store. This leads to a two-level mapping, of variables to locations and of locations to values. The `LOCATION` module defines what a location is, an example location is `l(17)`. It also shows how to concatenate multiple locations together, as we generally work on lists of expressions, etc.

```
fmod LOCATION is
  protecting INT .
  sorts Location LocationList .
  subsort Location < LocationList .
  op noLoc : -> LocationList .
  op _,_ : LocationList LocationList -> LocationList [assoc id: noLoc] .
  op l : Nat -> Location .
endfm
```

The `ENVIRONMENT` module defines an *environment* as a finite map from names to locations and also gives equations which define how it can be updated. It imports the `NAME` module that defines names, lists of names and equality on names.

```
fmod ENVIRONMENT is protecting LOCATION .
  protecting NAME .
  sort Env .
  op noEnv : -> Env .
  op [_,_] : Name Location -> Env .
  op __ : Env Env -> Env [assoc comm id: noEnv] .
  vars X Y : Name .  vars Env : Env .  vars L L' : Location .
  var Xl : NameList .  var Ll : LocationList .
  op _[_<-_] : Env NameList LocationList -> Env .
  op _[_<-_] : Env Name Location -> Env .
  eq Env[() <- noLoc] = Env .
  eq Env[X,Y,Xl <- L,L',Ll] = (Env [X <- L]) [Y,Xl <- L',Ll] .
  eq ([X,L] Env)[X <- L'] = ([X,L'] Env) .
  ceq ([Y, L] Env)[X <- L'] = [Y, L] (Env [X <- L'])
    if equalName(Y, X) = false .
  eq noEnv [X <- L'] = [X,L'] .
endfm
```

For example, the environment

```
([’X, l(1)] [’Y, l(2)]) [’X,’Y,’Z <- l(3),l(4),l(5)]
```

evaluates to `[’X,l(3)] [’Y,l(4)] [’Z,l(5)]`.

Values and *stores* are defined in the `VALUE` and `STORE` modules below. No equations are given for the store (unlike for the environment). This is due to our wish to stay extensible, which suggests that changes to the store should not be done here, but should instead be done in conjunction with, at least in a multi-threaded case, the currently working thread.

```
fmod VALUE is
  sorts Value ValueList .
  subsort Value < ValueList .
  op noVal : -> ValueList .
  op _,_ : ValueList ValueList -> ValueList [assoc id: noVal] .
  op [_] : ValueList -> Value .
endfm

fmod STORE is protecting LOCATION .
  extending VALUE .
  sort Store .
  op noStore : -> Store .
  op [_,_] : Location Value -> Store .
  op __ : Store Store -> Store [assoc comm id: noStore] .
endfm
```

Environments and stores are defined in a very concrete way for this language. Using a more abstract environment/store concept would have its advantages from

the point of view of program verification, as shown in [6,22] for a very simple language. But a more abstract concept of environment/stores does not work nicely with the side-effects and hiding that are possible in our language, for which the concrete variant we have chosen is preferable. Furthermore, this will make it easier to extend this subset of Java to a more complete version of Java in the future. In contrast, a more abstract definition of state would not allow more complex information, like exception, loop, or lock information, to be explicitly stored.

Within *continuations*, which we define next, all the execution context is stored. This can be viewed as "the rest of the program" which needs to be executed. The two operators shown here are two different ending points of an execution. Within the semantics we will define other operators with co-domain `Continuation` as needed. For example every expression of sort `Exp` can be put on the top (i.e. at the front) of a continuation.

```
fmod CONTINUATION is
  sort Continuation .
  op stop : -> Continuation .
  op res : -> Continuation .
endfm
```

The *state* is made up of state attributes, which are the environment, store, output and a counter for the next free memory location, each wrapped by some operator. Its structure is that of a set of such attributes obtained by the usual associative-commutative multiset union operator.

```
fmod STATE is extending ENVIRONMENT . extending STORE .
  extending CONTINUATION .
  sorts StateAttribute MyState .
  subsort StateAttribute < MyState .
  op empty : -> MyState .
  op _,_ : MyState MyState -> MyState [assoc comm id: empty] .
  op e : Env -> StateAttribute .
  op n : Nat -> StateAttribute .
  op m : Store -> StateAttribute .
  op out : Output -> StateAttribute .

  sorts SuperState WrappedState .
  subsort WrappedState < SuperState .
  op noState : -> WrappedState .
  op state : MyState -> WrappedState .
  op k : Continuation -> SuperState .
  op _,_ : WrappedState WrappedState
             -> WrappedState [assoc comm id: noState] .
  op _,_ : SuperState SuperState
             -> SuperState [assoc comm id: noState] .
endfm
```

The second set of sort declarations (and the operators for that) are needed because we do not want the context, i.e., the `Continuation`, to be part of the state, but only to be composable with it. So, instead of having `e(..), m(..), n(..), k(..)` we now have `state(e(..), m(..), n(..)), k(..)`.

Thanks to this structure we can check for *termination* of a program by simply checking the sort of the state. If it is of sort `SuperState`, there is still some continuation, and therefore code, left and the program has not yet terminated. If instead the resulting state is a `WrappedState`, we know that all code has been executed. The definition of what happens to an empty continuation needs to support this and does so.

2.2 Syntax and Semantics of Some Features

The Java fragment we are interested in includes arithmetic expressions, assignments, sequential composition and loops. Let us now take a look at the syntax and semantics of some features of our Java subset. We first discuss addition, then conditionals and finally loops.

Addition.

The syntax of addition is defined making use of the definition of generic expressions, which mainly just introduces the different possible forms of expressions.

```
fmod ARITH-EXP-SYNTAX is ex GENERIC-EXP-SYNTAX .
  op _+_ : Exp Exp -> Exp [prec 40 gather(E e)] .
  ...
```

The operator `+ ->` defined in `ARITH-EXP-SEMANTICS` allows us to evaluate an addition expression placed on top of a continuation. The first equation changes the evaluation of `E + E'` into first evaluating `(E, E')`, and then evaluating `-> +`. The second equation evaluates `-> +` by adding the two integers obtained by evaluating the expressions and placing the result on the top of the continuation stack.

```
fmod ARITH-EXP-SEMANTICS is protecting ARITH-EXP-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  op + -> _ : Continuation -> Continuation .
  vars E E' : Exp .  var K : Continuation .  vars I I' : Int .
  eq k((E + E') -> K) = k((E,E') -> + -> K) .
  eq k((int(I), int(I')) -> + -> K) = k(int(I + I') -> K) .
  ...
```

If-Then-Else.

In Java, the If-Then-Else construct does not actually contain a `then` but has instead the syntax specified in `IF-SYNTAX`, that imports `Statement`, a construct different from expressions since it does not create a return value. By the specified parsing precedences the dangling else problem is solved as in the Java Language Specification [11], that is, the `else` part belongs to the innermost `if`. We consider the If-Then as *syntactic sugar* and therefore give one equation in `IF-SYNTAX` which translates it into our If-Then-Else, meaning that we do not need to bother with it in the semantics at all. Also, one equation is enough for this desugaring.

```
fmod IF-SYNTAX is ex STATEMENT-SYNTAX .
  ex GENERIC-EXP-SYNTAX .
  op if__else_ : Exp Statement Statement -> Statement [prec 110] .
  op if__ : Exp Statement -> Statement [prec 115] .
  var E : Exp . var St : Statement .
  eq if E St = if E St else ; .
endfm
```

The evaluation of a conditional statement is split up into first evaluating the condition, while freezing the two code parts in the continuation, and then, once the condition is evaluated to either `true` or `false`, choosing the correct path. Note that we need to import boolean expressions here.

```
fmod IF-SEMANTICS is ex IF-SYNTAX . ex GENERIC-EXP-SEMANTICS .
  ex STATEMENT-SEMANTICS . ex BEXP-SEMANTICS .
  op ? (_,_) -> _ : Statement Statement Continuation -> Continuation .
  var E : Exp . vars St St' : Statement . var K : Continuation .
  eq k((if E St else St') -> K) = k(E -> ? (St, St') -> K) .
  eq k(bool(true) -> ? (St, St') -> K) = k(St -> K) .
  eq k(bool(false) -> ? (St, St') -> K) = k(St' -> K) .
endfm
```

While loops.

The syntax for while loops is straightforward. Note that the second argument of a `while` is a statement, but one can always wrap the sequential composition of several statements into a single block by using curly braces, e.g. {`S1 S2`}, with `S1` and `S2` statements. A block counts as a single statement again.

```
fmod WHILE-SYNTAX is ex STATEMENT-SYNTAX .
  ex GENERIC-EXP-SYNTAX .
  op while__ : Exp Statement -> Statement [prec 110] .
endfm
```

Defining the semantics of loops is now very easy by using the semantics of the conditional and unrolling the loop one step at a time:

```
fmod WHILE-SEMANTICS is ex WHILE-SYNTAX . ex GENERIC-EXP-SEMANTICS .
  ex STATEMENT-SEMANTICS . ex IF-SEMANTICS .
  var E : Exp . var St : Statement . var K : Continuation .
  eq k((while E St) -> K) = k(E -> ?({St while E St}, ;) -> K) .
endfm
```

2.3 An Interpreter for our Java Subset

The complete functional definition gives a precise mathematic axiomatization, in fact an *initial algebra semantics* of our chosen subset of Java that is sufficient for reasoning and program verification purposes. But since the semantic equations are ground confluent, the above semantic equations also give an *operational semantics* to this Java subset.

Indeed, we can *describe the execution of the language* by *algebraic simplification* with the equations from left to right. Therefore, our language definition has in essence given us an *interpreter* for our language. Note that in a few minor points we do not adhere to the strict syntax of Java because of some of the built-in types of Maude. For example, program variables are modeled with Maude quoted identifiers and therefore always have a quote (`'`) in front, and integers are wrapped with the operator `#i()` to avoid operations in Maude's built-in `INT` module to interfere with arithmetic operations in Java. With `initial` we create an initial empty state. By adding '`| CODE`' to any state, where '`CODE`' is some code fragment, of sort `BlockStatements`, we can compute the state resulting from executing that code fragment in the given state. Also, with `STATE[VAR]` the value of the variable `VAR` in a state `STATE` is returned. The equation accomplishing this is:

```
op _[_] : WrappedState Name -> Value .
var MYS: MyState . var X : Name . var L : Location . var Env : Env .
var V : Value . var M : Store .
eq state((MYS, e([X,L] Env), m([L,V] M)))[X] = V .
```

Some examples are:

```
red (initial | (int 'x = #i(1) ; int 'y = #i(20) ;
                {'x = #i(300) ; } 'x = 'x + 'y ;))['x] .

red (initial | (int 'x = #i(1) ; int 'y = #i(20) ;
                {int 'x = #i(300) ; } 'x = 'x + 'y ;))['x]  .
```

which return

```
rewrites: 86 in 10ms cpu (10ms real) (8600 rewrites/second)
result Value: int(320)
```

respectively, because of shadowing of the assignment to `'x` in the block,

```
rewrites: 86 in 0ms cpu (0ms real) (~ rewrites/second)
result Value: int(21)
```

A simple `swap` example, where `swap` is just a short-hand notation for the program defined by the equation `swap = (int 'T = 'X ; 'X = 'Y ; 'Y = 'T ;)`, indeed swaps the values of `'X` and `'Y`, the results are `5`, respectively `7`, as expected.

```
red (initial | (int 'X = #i(7) ; int 'Y = #i(5) ;
                swap))['X] .
red (initial | (int 'X = #i(7) ; int 'Y = #i(5) ;
                swap))['Y] .
```

A small factorial program

```
red (initial | (int 'n = #i(5) ; int 'c = #i(0) ; int 'x = #i(1) ;
                while ('c < 'n) { 'c = 'c + #i(1) ;
                                  'x = 'x * 'c ; })) ['x] .
```

computes the factorial of `5` and thus returns:

```
rewrites: 416 in 0ms cpu (0ms real) (~ rewrites/second)
result Value: int(120)
```

3 Hoare Triples

3.1 Pre and Post Conditions

Recall the `swap` program we have just shown in Sect. 2.3. A correctness specification for that example program, when done in an equational setting, could look like this:

```
(ctxState((int 'X ; 'int 'Y ;)) | (swap))['X] = (ctxState((int 'X ; 'int 'Y ;)))['Y]

(ctxState((int 'X ; 'int 'Y ;)) | (swap))['Y] = (ctxState((int 'X ; 'int 'Y ;)))['X]
```

We are able to verify this, using only the equations of our semantics and Maude's built-in equational simplification.

Now with S taking the place of `ctxState((int 'X ; int 'Y ;))`, and being aware that that represents all possible states in which this program can be run, the correctness specification can also be written like this:

$$(\forall \mathtt{I} : \mathtt{Int})(\forall \mathtt{J} : \mathtt{Int})(\mathtt{S})['\mathtt{Y}] = \mathtt{int}(\mathtt{I}) \wedge (\mathtt{S})['\mathtt{X}] = \mathtt{int}(\mathtt{J})$$
$$\Rightarrow (\mathtt{S} \mid (\mathtt{swap}))['\mathtt{X}] = \mathtt{int}(\mathtt{I}) \wedge (\mathtt{S} \mid (\mathtt{swap}))['\mathtt{Y}] = \mathtt{int}(\mathtt{J})$$

Here we have the implicit precondition that we are starting in a state where `'X` and `'Y` are declared as described above. We shall call the equation

$$(\mathtt{S})['\mathtt{Y}] = \mathtt{int}(\mathtt{I}) \wedge (\mathtt{S})['\mathtt{X}] = \mathtt{int}(\mathtt{J})$$

which is assumed to hold *before* the execution of the program, the *precondition*.

Note that this equation has a *single occurrence* of the state variable `S` in each equation, and can be thought of as a *state predicate*, having the integer variables `I` and `J` as *parameters*. Consider in the above specification the equation

$$(\mathtt{S} \mid (\mathtt{swap}))['\mathtt{X}] = \mathtt{int}(\mathtt{I}) \wedge (\mathtt{S} \mid (\mathtt{swap}))['\mathtt{Y}] = \mathtt{int}(\mathtt{J})$$

which is supposed to hold *after* the execution of a program. This can also be viewed as a state predicate, namely the state predicate

$$(\dagger)\ \ \texttt{(S)['X]} = \texttt{int(I)} \wedge \texttt{(S)['Y]} = \texttt{int(J)}$$

applied not to `S`, but instead to the state `S | (swap)` *after* the execution. We call (†) the *postcondition*. Note that it also has the integer variables `I` and `J` as extra parameters.

State Predicates.

This example suggests a general notion of *state predicate*, intuitively a *property* that holds or does not hold of a state, perhaps relative to some extra *data parameters*. Since in our Java subset the only data are integers, such parameters must be integer variables.

Therefore, for our language we can define a *state predicate* as a conjunction of equations

$$t_1 = t'_1 \ \wedge \ldots \wedge t_n = t'_n$$

in the module `JAVAX`, such that the set V of variables in all terms in the equations has *at most one* variable `S` of sort `State`, which may possibly appear more than once, and the remaining variables are all of sort `Int`.

One can of course generalize things further, by allowing an *arbitrary first-order formula* (with the same condition on its variables V) instead of just a conjunction of equations. Also, the notion extends naturally to other sequential languages which may have *other data structures* besides integers. However, in practice the above notion is quite general; among other things because, using an equationally defined equality predicate, we can express *arbitrary Boolean combinations* of equations (and therefore any quantifier-free formula) as a *single equation*.

3.2 Hoare Triples

The above example of our specification for `swap` is paradigmatic of a general way of specifying properties of a *sequential imperative program* p by means of a *Hoare triple* (after C.A.R. Hoare, see [12]),

$$\{A\}\ p\ \{B\}$$

where A and B are state predicates, called, respectively, the *precondition*, and *postcondition* of the triple.

In this notation, the specification of `swap` becomes rephrased as,

$$\{\texttt{(S)['Y]} = \texttt{int(I)} \wedge \texttt{(S)['X]} = \texttt{int(J)}\}\ \texttt{swap}\ \{\texttt{(S)['X]} = \texttt{int(I)} \wedge \texttt{(S)['Y]} = \texttt{int(J)}\}$$

Given our algebraic approach to the semantics of imperative programs, this is all just an (indeed very useful) *façon de parler* about an *ordinary first-order property* satisfied by the initial model of our language, namely the initial algebra $T_{\texttt{JAVAX}}$. The

module `JAVAX` is the module defining the semantics of our Java subset. It imports all other modules, defining the syntax, state infrastructure, and semantics.

Therefore, we define the *partial correctness* of a program `p` with respect to a Hoare triple by the equivalence,

$$T_{\texttt{JAVAX}} \models \{A\}\ \texttt{p}\ \{B\} \quad \Leftrightarrow$$

$$T_{\texttt{JAVAX}} \models (\forall V)\ A \wedge\ (\texttt{(S | p)}\ :\ \ \texttt{WrappedState}) \Rightarrow (B(\texttt{S/S | p})).$$

Here the `:` means sort membership, which in turn means that program `p` *terminates* when started in state `S`. Note that in the partial correctness interpretation the termination condition is on the lefthand side of the implication.

Our `swap` example thus becomes

$$T_{\texttt{JAVAX}} \models (\forall \texttt{I} : \texttt{Int})(\forall \texttt{J} : \texttt{Int})(\forall \texttt{S} : \texttt{State})\,\texttt{(S)['Y]} = \texttt{int(I)}\ \wedge\ \texttt{(S)['X]} = \texttt{int(J)}$$

$$\wedge\ \texttt{(S | swap)}\ :\ \ \texttt{WrappedState}$$

$$\Rightarrow \texttt{(S | (swap))['X]} = \texttt{int(I)} \wedge \texttt{(S | (swap))['Y]} = \texttt{int(J)}.$$

which is just our original correctness condition with the addition of the *termination condition* by the sort requirement. Of course, since `swap` was a terminating program, this was superfluous in that case, but it is not superfluous when loops are involved.

4 A Hoare Logic for our Java Subset and its Justification

An important contribution of Hoare was to propose his triples as a *compositional logic of programs*, by giving a collection of *inference rules* based on the *structure of the program text* to decompose the proof of correctness of more complex programs into proofs for simpler subprograms.

Hoare logic, however, is language-dependent: a Hoare rule valid for a construct in a given language may be invalid in another. For example, the classical Hoare rules for conditionals and for loops are both *invalid* even in our simple Java fragment and have to be suitably modified. It becomes therefore important to: (i) choose Hoare rules that adequately capture a given feature in a specific language and (ii) to mathematically *justify* the correctness of such a rule. For this second purpose, having a precise mathematical semantics of the language in question is an essential prerequisite. We therefore introduce a Hoare logic for our Java subset and justify the correctness of its rules based on our `JAVAX` formal semantics.

For example, to prove the correctness of a sequential composition `p q` he gave the rule,

$$\frac{\{A\}\ \texttt{p}\ \{B\} \qquad \{B\}\ \texttt{q}\ \{C\}}{\{A\}\ \ \texttt{p q}\ \ \{C\}}$$

which can be easily justified for our Java subset by analyzing both the semantic equations and the semantics of the triples.

Another rule of easy justification in our Java semantics is the rule for the skip program '`;`', which takes the form,

$$\frac{}{\{A\}\ \texttt{;}\ \{A\}}.$$

For *conditionals* we need to work a little harder, because the classical Hoare rule for conditional is *invalid*. The key difficulty is that evaluating a conditional's boolean expression may side-effect the state. Here, `evalTst(S, TE)` gives the boolean which the evaluation of the test expression `TE` in state `S` returns. Using the operator '`|`' we can separate the "execution" of a test expression from the rest of the program. So we have now overloaded `|`'s meaning to both combine a state and a program fragment and also to combine a state and an expression, but because of the different typings involved no ambiguity arises and this is not a problem. Furthermore, since in the Hoare triples we sometimes need to consider the execution of an expression for side effect purposes only in conjunction with the execution of a statement, our Hoare triples allow not only statements, but also expressions. Any "sequential composition" of them using the `|` symbol in an "ad-hoc" overloaded way is allowed. Note that such "compositions", though meaningful in terms of their effects on state, do not correspond to legal Java programs; however, they are needed in the Hoare rules. Of course, both uses of `|` are closely related, since, for example, given a state `s`, an expression `e`, and a statement `p` we semantically interpret the effect of `e | p` on `s` by the equation

$$\texttt{s | (e | p)} = \texttt{(s | e) | p}.$$

It is not possible to use the usual Java program concatenation `_ _` here, because the test expression is not of the same sort as the other statements. But using the `|` operator it can be evaluated first, so that its side effects change the state, and then the result gets thrown away and the execution continues as usual. This is our way to allow the expression to be used as if it were a statement, just for its side-effects.

The function `evalTst` evaluates a test expression in a given state to a boolean value. As it is a bit cumbersome to write this out multiple times in some of the rules, we overload our notation a little and use a test `t` in two different ways in the following. In the *property part* of a Hoare triple, `t` will stand for the equality `evalTst(S, t)` = `true`, with `S` the variable for the distinguished state for which that property has to hold, and similarly, $\neg$`t` will stand for `evalTst(S, t)` = `false`. That use of `t` (respectively its negation) only gives us the boolean value and does not change the state. Whenever the state `S` is not obvious, we will fall back on the `evalTst` notation. The other way we use `t` in is in the *code part* as usual (within `if` or `while` constructs) or just for its possible state change as described above. The different uses of `t` are illustrated in our Hoare rule for conditionals,

$$\frac{\{A \wedge \texttt{t}\}\ \texttt{t | p}\ \{B\}\ \ \{A \wedge \neg\texttt{t}\}\ \texttt{t | q}\ \{B\}}{\{A\}\ \texttt{if t p else q}\ \{B\}}$$

This captures the usual semantics of `if`, just as in the simpler languages, but in contrast here, since `t` can have side effects, we have '`t |`' in front of the execution

of the two branches of the conditional in the respective cases. It is not enough to know that `t` evaluates to either true or false, which is what the two properties assure, but `t` needs to be also *executed* for its possible side effects. This Hoare rule still simplifies things, since we now do not have to take a decision based on the test value anymore, but we just have to have the test expression executed. One could also give a sequential composition rule for '`|`' additionally. The extra effort here is necessary because of side effects!

Another very useful rule, of easy justification based on the semantics of Hoare Triples, is the *consequence* rule,

$$\frac{A \Rightarrow A_1 \quad \{A_1\}\ \mathtt{p}\ \{B_1\} \quad B_1 \Rightarrow B}{\{A\}\ \mathtt{p}\ \{B\}}$$

The most important rule in our language subset is the proof rule for the partial correctness of *while loops*. Here we face the same problem as with conditionals, because the loop condition can also have side effects. It takes the form,

$$\frac{\{A \wedge \mathtt{t}\}\ \mathtt{t}\ \mathtt{|}\ \mathtt{p}\ \{A\} \quad \{A \wedge \neg\mathtt{t}\}\ \mathtt{t}\ \{A \wedge \neg\mathtt{t}\}}{\{A\}\ \mathtt{while\ t\ p}\ \{A \wedge \neg\mathtt{t}\}}$$

This rule requires a somewhat more involved justification, which is done in the proof given in the technical report [24]. The state predicate A is called an *invariant* of the loop. This rule needs the additional Hoare triple for the test:

$$(\natural) \qquad \{A \wedge \neg\mathtt{t}\}\ \mathtt{t}\ \{A \wedge \neg\mathtt{t}\}$$

because of the way side effects can propagate with the loop unrolling. A loop works like this:

$$\mathtt{while\ t\ p} \ \rightarrow\ \mathtt{t\ |\ p\ |\ while\ t\ p} \ \rightarrow\ \ldots \ \rightarrow$$
$$\mathtt{t\ |\ p\ |\ \ldots\ |\ t\ |\ p\ |\ while\ t\ p} \ \rightarrow\ \mathtt{t\ |\ p\ |\ \ldots\ |\ t\ |\ p\ |\ t}$$

In the final state that is thus attained, the test `t` does not necessarily evaluate to `false`. In the state before the final state it did indeed evaluate to `false`, but its side effect could cause its next evaluation to be `true` again. To prevent this, the Hoare triple ($\natural$) has to be added to the proof obligation of the loop rule.

An example Java program where this problem appears is the following:

```
int 'i = #i(0) ; while ( !  ( ('i = 'i + #i(1) ) == #i(1))) ...
```

Here in the condition check `'i` is increased to 1, so the equality holds and therefore the negation is false and the loop is never entered. But if the condition were evaluated in this final state, `'i` would get the value 2, the equality would not hold and therefore the negation would hold. So here the condition is not false in the final state.

A Factorial Example.

Consider the factorial program in Section 2.3. To prove its correctness, intuitively that it correctly computes the factorial function, we first need to define

mathematically such a function, by defining an operator `facValue` and its defining equations,

```
op facValue : Int -> Int .
var I : Int .
ceq facValue(I) = 1 if 0 < I = false .
ceq facValue(I) = I * facValue(I - 1) if 0 < I = true .
```

To avoid complications with non-termination we have defined the factorial of a negative number to be 1.

We are only interested in the meaningful results of the factorial function. Therefore, we should give the requirement that the input variable `'N` is nonnegative as a *precondition*, yielding the specification,

$$\{(\texttt{S['N]} = \texttt{int(I)}) \wedge (\texttt{0<=I} = \texttt{true})\}\ \texttt{facx}\ \{\texttt{S['X]} = \texttt{int(facValue(I))}\}.$$

The above specification takes the point of view of a *customer* who specifies properties of the desired program. An *implementer* may then give to the customer the following `facx` program:

```
'C = #i(0) ; 'X = #i(1) ; while ('C < 'N) { 'C = 'C + #i(1) ; 'X = 'X * 'C ; }
```

The question, then, is how to prove this program correct. To do so we can:

- use the Hoare logic rules, which we have justified, and
- use inductive reasoning, since the correctness of Hoare triples reduces to satisfaction of first-order formulas in the initial model $T_{\texttt{JAVAX}}$.

A proof script of this program in our Java+ITP Tool is given in the technical report [24].

5 The Java+ITP Tool

The *latest* version (extended by us with support for this Java subset) of the ITP is downloadable from [23] together with the semantics of the Java fragment. It has an extension of the list of commands of the ITP specifically designed to support Hoare logic reasoning in our programming language. How the Java+ITP tool works in detail and is interfaced with Maude's ITP is also explained on the above-mentioned web-page.

5.1 Proving Hoare Triples in the ITP with the `javax` Command

In Java+ITP the `javax` command translates a Hoare triple into its semantically equivalent inductive theorem proving goal. For example, a goal consisting of the Hoare triple

$$\{P\}\ \texttt{C}\ \{Q\}$$

is translated into the (universally quantified) ITP goal

$$P\ \Rightarrow\ Q(S/(S\texttt{|}\ \texttt{C}))$$

where S is the distinguished variable of sort `WrappedState`.

5.2 Proving While Loops with *javax-inv*

The `javax` command allows a user to enter a Hoare triple goal into the ITP to prove the correctness of the program mentioned in the triple. However, the *compositional* approach favored by Hoare logic suggests that we should *first decompose* the original Hoare triple into simpler ones by using the Hoare logic inference system. For this reason, the Java+ITP tool not only does automate the entering of Hoare triples into the ITP. It also *automates the application of some Hoare rules*. For while loop programs, this is accomplished by means of the `javax-inv` command. We consider while loop programs of the general form `wlp` = `init loop`, with `loop` = `while t p`. That is, we allow the subprogram `init` to be executed *before* the while loop proper, since this is a very common situation.

The `javax-inv` command allows the specification of the following information about a while loop program `wlp` of the form just described:

- the *precondition* P and *postcondition* Q against which one wants to prove `wlp` correct.
- the *invariant* A that should be used to *decompose* the original Hoare triple into simpler ones using the Hoare rules.

The `javax-inv` command then does the following things:

- it applies the *composition* rule to:

$$\frac{\{P\}\ \texttt{init}\ \{A\} \qquad \{A\}\ \texttt{loop}\ \{Q\}}{\{P\}\ \texttt{init loop}\ \{Q\}}$$

- it then applies the *consequence* rule to further decompose the second subgoal

$$\frac{A \Rightarrow A \quad \{A\}\ \texttt{loop}\ \{A \wedge \neg \texttt{t}\} \quad (A \wedge \neg \texttt{t}) \Rightarrow Q}{\{A\}\ \texttt{loop}\ \{Q\}}$$

- it finally applies the *loop* rule:

$$\frac{\{A \wedge \texttt{t}\}\ \texttt{t | p}\ \{A\} \quad \{A \wedge \neg \texttt{t}\}\ \texttt{t}\ \{A \wedge \neg \texttt{t}\}}{\{A\}\ \texttt{while t p}\ \{A \wedge \neg \texttt{t}\}}$$

As a consequence, the following four subgoals are generated:

(i) $\{P\}$ `init` $\{A\}$

(ii) $\{A \wedge \texttt{t}\}$ `t | p` $\{A\}$

(iii) $(A \wedge \neg \texttt{t}) \Rightarrow Q$

(iv) $\{A \wedge \neg \texttt{t}\}$ `t` $\{A \wedge \neg \texttt{t}\}$.

The implementation of the `javax-inv` command in the ITP then implicitly applies the `javax` command to the Hoare triples (1), (2) and (4), so that we end up with the following four ITP goals:

(i) $P \Rightarrow A(S/S\,|\,\texttt{init})$

(ii) $(A \wedge \texttt{t}) \Rightarrow A(S/S\,|\,\texttt{t}\,|\,\texttt{p})$

(iii) $(A \wedge \neg\texttt{t}) \Rightarrow Q$

(iv) $(A \wedge \neg\texttt{t}) \Rightarrow A(S/S\,|\,\texttt{t}) \wedge (\texttt{evalTst}(S/S\,|\,\texttt{t}, \texttt{t}) = \texttt{false})$

5.3 Supporting Compositionality

From a user's perspective, the commands `javax` and `javax-inv` directly create the first order goals for their corresponding Hoare triples. However, by the way these commands are structured they do not allow application to more complex programs such as, for example, a program of the form `init1 loop1 init2 loop2`. But obviously, by applying the composition rule, with a suitable middle condition, a program like this could be split up into two parts, so that each could be treated by the commands that we have already discussed.

To allow this kind of compositional reasoning, Java+ITP provides several commands. First of all, to enter a Hoare triple into the tool *without* translating it into its corresponding first-order goal the `add-hoare-triple` command can be used. Furthermore, Java+ITP also offers a `decompose` command, which decomposes a Hoare triple and its code into two Hoare triples with a suitable middle condition (provided by the user). That is, given `{A} P {B}` with `A` and `B` state predicates and `P` a program we can decompose this into the two Hoare triples `{A} P1 {C}` and `{C} P2 {B}` with `C` a state predicate and `P1` and `P2` two programs, all three provided by the user giving the `decompose` command, where the two programs need to make up `P`, i.e. `P = P1 P2`.

After having decomposed in this way the original Hoare triple for a program into several simpler Hoare triples, Java+ITP then supports translating such simpler triples into first-order goals. This is accomplished with the commands `create-FO-goal-hoare`, and `create-FO-goal-hoare-inv`, which are the respective analogues of the `javax` and `javax-inv` commands.

This support for compositionality allows us to tackle more complicated programs, for example programs involving multiple loops. Using the above commands we can create a number of Hoare triple goals from just one starting goal and then can generate the respective first order goals for all of them and discharge them with the ITP.

5.4 An Example: A Binomial Coefficient Program

We show the usefulness of introducing Hoare triples, decomposing them and then proving the separate subgoals with an example of the binomial coefficient function $\binom{n}{k}$. The details of this decomposition and proof can be found in the technical report [24]. The main facts are the program:

```
op choose-program : -> BlockStatements .
eq choose-program = (
    int 'N ; int 'Nfac ; int 'K ; int 'Kfac ;
    int 'N-Kfac ; int 'BC ; int 'I ;

    'I = #i(0) ; 'Nfac = #i(1) ;
```

```
while ('I < 'N) { 'I = 'I + #i(1) ; 'Nfac = 'Nfac * 'I ; }

'I = #i(0) ; 'Kfac = #i(1) ;
while ('I < 'K) { 'I = 'I + #i(1) ; 'Kfac = 'Kfac * 'I ; }

'I = #i(0) ; 'N-Kfac = #i(1) ;
while ('I < ('N - 'K)) { 'I = 'I + #i(1) ; 'N-Kfac = 'N-Kfac * 'I ; }

 'BC = 'Nfac / ('Kfac * 'N-Kfac) ; ) .
```

and the property to be verified:

$$\{\texttt{int-val(S:WrappedState['N])} = \texttt{(N:Int)} \wedge \texttt{int-val(S:WrappedState['K])} = \texttt{K:Int}$$
$$\wedge\ \texttt{0 <= N:Int} = \texttt{true} \wedge \texttt{0 <= K:Int} = \texttt{true} \wedge \texttt{0 <= N:Int - K:Int} = \texttt{true}\}$$
$$\textbf{choose-program}$$
$$\{\texttt{int-val(S:WrappedState['Nfac])} = \texttt{(N:Int)!}$$
$$\wedge\texttt{int-val(S:WrappedState['Kfac])} = \texttt{(K:Int)!}$$
$$\wedge\texttt{int-val(S:WrappedState['N-Kfac])} = \texttt{(N:Int - K:Int)!}$$
$$\wedge\texttt{int-val(S:WrappedState['BC])} = \texttt{choose(N:Int, K:Int)}\}$$

The basic idea is then to give suitable middle conditions to split the Hoare triples apart. The empty lines of the program indicate the split positions within the program.

6 Related Work and Conclusions

We first discuss related work using rewriting logic and the Maude system [5]. The CPS style has been found to be quite expressive and extensible in several experiments in the rewriting semantics project [19,18]; it has in particular been used for Java in the JavaFAN project [7,8]. We have adopted this semantics in Java+ITP for extensibility reasons; but, as discussed in Section 2, we structured the state and added extra functionality to suit theorem proving uses. Java+ITP is an extension of Maude's ITP [4]. A project similar to ours, namely the ASIP+ITP tool [6,22], has been carried out by M. Clavel and J. Santa-Cruz at UCM in Madrid. While benefitting from their experience, we had to address and solve new research issues. ASIP+ITP is based on a considerably simpler programming language used by Goguen and Malcolm [10]; one whose expressions do not have any side-effects, whose variables can be directly mapped to values in memory, and where the whole semantics cannot be extended to accommodate new features. We are primarily interested in modularity and extensibility of programming languages and Hoare logics, and view Java+ITP as a research vehicle to advance those goals. Another difference is Java+ITP's support for compositional reasoning: in ASIP+ITP VCs for Hoare triples, including those for loops, can be generated, but triples cannot be decomposed into simpler ones. Still partially in this framework, W. Ahrendt, A. Roth and the first author report in [2] on a cross-validation of a Java semantics given in the rewriting semantics framework against the Java program transformation rules of the KeY prover [1].

Commenting more broadly on Java verification work, the Java Modeling Language, JML [3], is a good way to specify the relevant properties of programs. We

have not yet made use of JML in this work, but extending Java+ITP in this direction seems worthwhile. In [15], B. Jacobs, C. Marché and N. Rauch give an overview of the capabilities of different tools by comparing how they can deal with a real-world example program. They look at ESC/Java [9], Jive [20], Krakatoa [17] and the LOOP project [16]. ESC/Java [9] is only a checker which is neither sound nor complete. Jive [20] is based on Hoare logic, but it has no side-effects, at least not in the expressions which are used to take decisions, like the `if` and `while` test expressions. Therefore the resulting Hoare logic is much simpler. Krakatoa [17] uses a modeling of the Java heap and it makes use of several sub-tools which create the proof obligations for it. They work with Java but do not have a Hoare logic approach.

In the LOOP project [16], a denotational semantics of Java is formalized as a PVS theory. Java programs are compiled into semantical objects, and proofs are performed in the PVS theory directly. On top of that, a Hoare-style and a weakest precondition (*wp*) style calculus are formalized as a PVS theory, and are verified against the semantics within PVS. As opposed to 'usual' Hoare-style or *wp* calculi, these ones work on the *semantical* objects, not on the Java syntax. In his weakest-precondition reasoning work [14], B. Jacobs also works only on the semantical object level. Similarly, M. Huismann, in her thesis [13], also works on this semantic translation of the source code into the type theory (of PVS or Isabelle). The Hoare logic is given on that level only, not on the Java source code, which is the difference to our work.

In conclusion, we view Java+ITP as a research vehicle to investigate modularity and extensibility of programming languages and of Hoare logics. It has served us well for this purpose, by uncovering subtleties in the Hoare logic needed for Java not present in toy languages, and not even present in the Hoare logics of Java tools like Jive. Keeping the compositional Hoare logic reasoning at the source code level is also one of the goals that, in contrast to other approaches, we have advanced. But of course this is just a snapshot of work in progress. Our Java fragment is still quite modest, so we should soon add new features to it such as exceptions and objects; we expect this to be easy thanks to the CPS semantics. After this, threads and concurrency should also be added, and Hoare rules for these new features should also be investigated. Our goal is of course modularity, so that our Hoare rules will be applicable not just to Java, but to any other languages using some of the same features in a modular way, but this still remains an exciting goal for the future.

Acknowledgement

We cordially thank Manuel Clavel for his generous help with details of the ITP implementation. This help has been crucial for us to extend the ITP into Java+ITP.

References

[1] Ahrendt, W., T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager and P. H. Schmitt, *The KeY tool*, Software and System Modeling **4** (2005), pp. 32–54.

[2] Ahrendt, W., A. Roth and R. Sasse, *Automatic validation of transformation rules for java verification against a rewriting semantics*, in: G. Sutcliff and A. Voronkov, editors, *Proceedings, 12th International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Montego Bay, Jamaica*, LNCS **3835** (2005), pp. 412–426.

[3] Burdy, L., Y. Cheon, D. Cok, M. D. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino and E. Poll, *An overview of JML tools and applications*, Software Tools for Technology Transfer **7** (2005), pp. 212–232.

[4] Clavel, M., *The itp tool's home page* (2005), http://maude.sip.ucm.es/itp.

[5] Clavel, M., F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer and C. Talcott, "Maude Manual," (April 2005), available from http://maude.cs.uiuc.edu.

[6] Clavel, M. and J. Santa-Cruz, *ITP/ASIP: a verification tool based on algebraic semantics* (2005), to appear in Proc. *PROLE 2005*: V Jornadas Sobre Programacin y Lenguajes, Thomson.
URL http://maude.sip.ucm.es/~clavel/pubs

[7] Farzan, A., F. Chen, J. Meseguer and G. Roşu, *Formal analysis of Java programs in JavaFAN.*, in: R. Alur and D. Peled, editors, *CAV*, Lecture Notes in Computer Science **3114** (2004), pp. 501–505.

[8] Farzan, A., J. Meseguer and G. Roşu, *Formal jvm code analysis in javafan.*, in: Rattray et al. [21], pp. 132–147.

[9] Flanagan, C., K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe and R. Stata, *Extended static checking for java*, in: *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation* (2002), pp. 234–245.

[10] Goguen, J. and G. Malcolm, "Algebraic Semantics of Imperative Programs," The MIT Press, 1996.

[11] Gosling, J., B. Joy, G. Steele and G. Bracha, "The Java Language Specification Second Edition," Addison-Wesley, Boston, Mass., 2000.
URL citeseer.ist.psu.edu/gosling00java.html

[12] Hoare, C. A. R., *An axiomatic basis for computer programming*, Commun. ACM **12** (1969), pp. 576–580.

[13] Huisman, M., "Reasoning about JAVA programs in higher order logic with PVS and Isabelle," Ph.D. thesis, Katholieke Universiteit Nijmegen (2001).

[14] Jacobs, B., *Weakest precondition reasoning for Java programs with JML annotations*, Journal of Logic and Algebraic Programming **58** (2004), pp. 61–88.

[15] Jacobs, B., C. Marché and N. Rauch, *Formal verification of a commercial smart card applet with multiple tools*, in: Rattray et al. [21], pp. 241–257.

[16] Jacobs, B. and E. Poll, *Java program verification at Nijmegen: Developments and perspective*, in: K. Futatsugi, F. Mizoguchi and N. Yonezaki, editors, *Software Security – Theories and Systems*, LNCS 3233 (2004), pp. 134–153.

[17] Marché, C., C. Paulin-Mohring and X. Urbain, *The krakatoa tool for certificationof java/javacard programs annotated in jml.*, J. Log. Algebr. Program. **58** (2004), pp. 89–106.

[18] Meseguer, J. and G. Roşu, *The Rewriting Logic semantics project*, in: *Structural Operational Semantics, Proceedings of the SOS Workshop, Lisbon, Portugal, 2005*, ENTCS (2005), to appear.

[19] Meseguer, J. and G. Rosu, *Rewriting logic semantics: From language specifications to formal analysis tools.*, in: D. A. Basin and M. Rusinowitch, editors, *IJCAR*, Lecture Notes in Computer Science **3097** (2004), pp. 1–44.

[20] Meyer, J. and A. Poetzsch-Heffter, *An architecture for interactive program provers.*, in: S. Graf and M. I. Schwartzbach, editors, *TACAS*, Lecture Notes in Computer Science **1785** (2000), pp. 63–77.

[21] Rattray, C., S. Maharaj and C. Shankland, editors, "Algebraic Methodology and Software Technology, 10th International Conference, AMAST 2004, Stirling, Scotland, UK, July 12-16, 2004, Proceedings," Lecture Notes in Computer Science **3116**, Springer, 2004.

[22] Santa-Cruz, J., "ITP/ASIP: a Verification Tool for Imperative Programs based on Algebraic Semantics," Master's thesis, Facultad de Informática, Universidad Complutense de Madrid (2005).
URL http://maude.sip.ucm.es/~juansc/

[23] Sasse, R., *Java+ITP tool* (2005), http://maude.cs.uiuc.edu/tools/javaitp.

[24] Sasse, R. and J. Meseguer, *Java+ITP: A verification tool based on hoare logic and algebraic semantics*, Technical Report UIUCDCS-R-2006-2685, Department of Computer Science, University of Illinois at Urbana-Champaign, USA (2006).