Chapter 17
KeY-Hoare

Richard Bubel and Reiner Hähnle

17.1 Introduction

In contrast to the other book chapters that focus on verification of real-world Java program, here we introduce a program logic and a tool based on KeY that has been designed solely for teaching purposes (see the discussion in Section 1.3.3). It is targeted towards B.Sc. students who get in contact with formal program verification for the first time. Hence, we focus on program verification itself, while treating first-order reasoning as a black box. We aimed to keep this chapter self-contained so that it can be given to students as reading material without requiring them to read other chapters in this book. Even though most of the concepts discussed here are identical to or simplified versions of those used by the KeY system for Java, there are subtle differences such as the representation of arrays, which we point out when appropriate.

Experience gained from teaching program verification using Hoare logic [Hoare, 1969] (or its sibling the weakest precondition (wp) calculus [Dijkstra, 1976]) as part of introductory courses exposed certain short-comings when relying solely on text books with pen-and-paper exercises:

- Using the original Hoare calculus requires to "guess" certain formulas, while using the more algorithmic wp-calculus requires to reason backwards through the target program, which is experienced as nonnatural by the students.
- Doing program verification proofs by hand is tedious and error-prone even for experienced teachers. The reason is that trivial and often implicit assumptions like bounds or definedness of values are easily forgotten. Hence, pen-and-paper proofs often contain undiscovered minor bugs due to too weak or incomplete specifications.
- First-order logic reasoning is typically not a part of an introduction to program verification. But this is difficult to avoid in standard formulations of program verification where the reduction of programs to first-order proof obligations and first-order reasoning is interleaved.

© Springer International Publishing AG 2016
W. Ahrendt et al. (Eds.): Deductive Software Verification, LNCS 10001, pp. 571–589, 2016
DOI: 10.1007/978-3-319-49812-6_17

KeY-Hoare mitigates the identified short-comings as follows: The first issue is addressed by reusing the idea of symbolic state updates from JavaDL (see Chapter 3). We designed a calculus to realize a symbolic execution style of reasoning in the same spirit as the JavaDL calculus, but we stay close to the formalism used in a Hoare-style calculus that extends standard Hoare triples to Hoare quadruples including symbolic state updates. This is only a small change, which does not impede the students from using standard text books (for example, [Tennent, 2002, Huth and Ryan, 2004]). Our target programming language is a simple imperative while programming language with scalar arrays. The remaining two issues have been solved by providing an easy-to-use tool which implements our modified Hoare calculus on top of the powerful first-order reasoning engine of KeY. Specifically, the discharge of first-order proof obligations can be treated as a black box.

This chapter is an extended version of the paper [Hähnle and Bubel, 2008] adding support for arrays, *total* correctness and worst-case execution time analysis.

17.2 The Programming Language

For KeY Hoare we focus on a simple imperative programming language whose syntax is defined by the following grammar:

```
⟨Program⟩ ::= (⟨Statement⟩)?
⟨Statement⟩ ::= ⟨EmptyStatement⟩ | ⟨AssignmentStatement⟩ |
                ⟨CompoundStatement⟩ | ⟨ConditionalStatement⟩|
                ⟨LoopStatement⟩
⟨EmptyStatement⟩ ::= ';'
⟨AssignmentStatement⟩ ::= ⟨Location⟩ = ⟨Expression⟩';'
⟨CompoundStatement⟩ ::= ⟨Statement⟩⟨Statement⟩
⟨ConditionalStatement⟩ ::= if '('⟨BoolExp⟩')'
                           '{'⟨Statement⟩'}' else '{'⟨Statement⟩'}'
⟨LoopStatement⟩ ::= while'('⟨BoolExp⟩')' '{'⟨Statement⟩'}'
⟨Expression⟩ ::= ⟨BoolExp⟩ | ⟨IntExp⟩
⟨BoolExp⟩ ::= ⟨IntExp⟩ ⟨CompOp⟩ ⟨IntExp⟩ | ⟨IntExp⟩ == ⟨IntExp⟩ |
              ⟨BoolExp⟩ ⟨BoolOp⟩ ⟨BoolExp⟩ | !⟨BoolExp⟩ |
              ⟨Location⟩ | true | false
⟨IntExp⟩ ::= ⟨IntExp⟩ ⟨IntOp⟩ ⟨IntExp⟩ | ℤ | ⟨Location⟩
⟨CompOp⟩::= < | <= | >= | >                        ⟨BoolOp⟩ ::= & | | | ==
⟨IntOp⟩::= * | / | % | + | -       ⟨Location⟩ ::= IDENT | IDENT[⟨IntExp⟩]
```

The grammar used in the implementation is slightly more complex as it incorporates the usual precedence rules for operators and allows one to use parenthesized expressions. We point out some important design decisions:

1. The programming language has only two incomparable types called `boolean` and `int`, in particular, no array types exist. To simplify the semantics of arith-

metic operations, the type `int` represents the mathematical integers $\mathbb{Z}$ and not a finite integer type with overflow.
2. Program variables and arrays (referred to as $\langle$*Locations*$\rangle$ by the grammar) are declared globally outside of the program, i.e., there are no local program variables and arrays can only be manipulated, but not created at runtime. Arrays are unbounded and their elements are uniquely indexed by any integer. This avoids the need for checking boundaries when accessing array elements and to define exceptional behavior in case an array access is out of bounds.
3. *Locations* are program variables or array access expressions, e.g., `a[i]` (with `i` being an integer expression such as a program variable or number literal). Arrays themselves are scalar arrays, referenced by name. They are neither locations nor expressions. Let `a` and `b` denote two arrays of element type `int` then `a == b` is not a valid comparison expression and `a = b;` is not a valid assignment statement. On the other hand, `a[7] == b[8]` or `a[i] = b[j];` are syntactically valid. In addition, as `a` and `b` are different names, they denote different arrays. This avoids the aliasing problem for arrays (that is present in Java), e.g., `a[i]` and `b[i]` are never aliases in our language.

17.3 Background

17.3.1 First-Order Logic

Specifying a program means to express what a program is supposed to do in contrast to how to do it. To be able to machine-check whether a given implementation (which describes *how* to compute a function) adheres to its specification, we need to define a formal language in which to write the specification as we needed one (the programming language) to write the implementation.

The language of our choice is typed first-order logic as a small but well studied formal language suitable to express state-based properties about the behavior of programs. Terms and formulas are inductively defined as usual, but we also allow expressions of the programming language as terms. This does not cause any problems, as all expressions of our programming language are side effect free.

As usual the atomic formulas of our logic are either formulas of the form $P(t_1,\ldots,t_n)$ with a (user-defined) predicate symbol P of arity n and terms $t_1,\ldots,t_n$ of appropriate type or $s \doteq t$ with the reserved *equality* symbol $\doteq$ taking arbitrary terms as arguments.

Program variables are *not* modeled as first-order variables but as constants (0-ary functions). Therefore, it is not possible to quantify over program variables. This modeling is identical to the one used by JavaDL (see Chapter 3). Program variables are special constant symbols in the sense that they are *nonrigid* symbols. In contrast to *rigid* symbols which are evaluated by a classical interpretation function and variable assignment, *nonrigid* symbols (and hence program variables) are evaluated

with respect to a (program) state. As a consequence the value of rigid symbols is fixed and cannot be changed by a program, while nonrigid program variables can be evaluated differently depending on the current execution state. A common usage of rigid function symbols is to "store" the initial value of a program variable so that one can refer to that value in a later execution state. In addition, built-in symbols with a fixed semantics, such as equality $\doteq$ and arithmetic operators of the programming language, are rigid symbols.

In JavaDL the only nonrigid symbols are program variables, but for KeY-Hoare it is useful to implement as well nonrigid unary function symbols f that map `int` values to values of type `int` or `boolean`. They are used to model arrays. We permit the notation $f[i]$ instead of $f(i)$ to resemble more closely the array syntax used in programs.

Many program logics, including the original logic by Hoare [1969], model program variables with first-order (rigid) variables. The disadvantage is that this requires to introduce so called *primed* variables as in [Boute, 2006]. Each state change during symbolic execution then causes the introduction of fresh primed variables. The increase in the number of symbols required to specify and to prove a problem compromises the readability of proofs considerably. Readability, however, is a central issue where interactive usability is concerned—not only for students at the beginner level. A further point in favor of nonrigid symbols is to avoid confusion for students who may just have gotten used to the static viewpoint of first-order logic.

Some useful *conventions*: program variables are typeset in *typewriter* font, logical variables in *italic*. When we specify a program π we assume that all program variables of π are contained in the first-order signature with their correct type. The *semantics* of first-order formulas is interpreted over fixed domain models. Specifically, all Boolean terms are interpreted over $\{\mathtt{true}, \mathtt{false}\}$ and all integer terms over $\mathbb{Z}$. There are built-in function symbols for arithmetic including `+`, `-`, `*`, `/` and `%` and integer comparison operators `<=`, `<`, `>` and `>=` with their standard meaning. For the concrete formula syntax see Section 17.7.2. Apart from that, all semantic notions such as satisfiability, model, validity, etc., are completely standard and as defined in Chapter 2.

17.3.2 Hoare Calculus

Before we define our own version, we present a standard version of the Hoare [1969] calculus (without arrays) to introduce some basic notions and to identify some problematic design decisions that we avoid in our approach. As usual, the behavior of programs is specified with *Hoare triples*:

$$\{P\}\,\pi\,\{Q\} \tag{17.1}$$

Here, P and Q are closed first-order formulas and π is a program over locations $L = \{l_1, \ldots, l_m\}$. The meaning of a Hoare triple is as follows: *for each first-order*

model $\mathcal{M} = (D, I)$ *and program state* s *of* P*, if* π *is started with initial values* $i_k = val_{\mathcal{M},s,\beta}(l_k)$ $(1 \leq k \leq m)$ *and if* π *terminates with final values* f_k*, then* $\mathcal{M}, s^{f_1,\ldots,f_m}_{l_1,\ldots,l_m}, \beta$ *is a model of* Q.

We can paraphrase this in a slightly more informal, but more intuitive, manner: for a given program π over locations $\{l_1, \ldots, l_m\}$, let us call an assignment of values $l_k = v_k$ $(1 \leq k \leq m)$ a *state* s of π. What the Hoare triple then says is that if we start π in any state satisfying the *precondition* P, if π terminates, then we end up in a final state that satisfies *postcondition* Q.

The standard Hoare rules are displayed in Figure 17.1. We use the following conventions for schematic variables occurring in the rules: `e` is an expression, `b` is a Boolean expression, `x` is a program variable, `s`, `s1`, `s2` are statements. P, Q, R, I are closed first-order formulas. Here $P\{\mathtt{x}/\mathtt{e}\}$ is the formula arising from P by replacing every occurrence of the constant `x` by the expression `e`.

$$\text{assignment}\ \frac{}{\{P\{\mathtt{x}/\mathtt{e}\}\}\,\mathtt{x=e;}\,\{P\}}$$

$$\text{composition}\ \frac{\{P\}\,\mathtt{s1}\,\{R\} \qquad \{R\}\,\mathtt{s2}\,\{Q\}}{\{P\}\,\mathtt{s1\ \ s2}\,\{Q\}} \qquad \text{skip}\ \frac{}{\{P\}\,\mathtt{;}\,\{P\}}$$

$$\text{conditional}\ \frac{\{P \land \mathtt{b} \doteq \mathtt{true}\}\,\mathtt{s1}\,\{Q\} \qquad \{P \land \mathtt{b} \doteq \mathtt{false}\}\,\mathtt{s2}\,\{Q\}}{\{P\}\,\mathtt{if(b)\{s1\}else\{s2\}}\,\{Q\}}$$

$$\text{loop}\ \frac{\{I \land \mathtt{b} \doteq \mathtt{true}\}\,\mathtt{s}\,\{I\}}{\{I\}\,\mathtt{while(b)\{s\}}\,\{I \land \mathtt{b} \doteq \mathtt{false}\}}$$

$$\text{weakeningLeft}\ \frac{P \rightarrow Q \qquad \{Q\}\,\mathtt{s}\,\{R\}}{\{P\}\,\mathtt{s}\,\{R\}} \qquad \text{weakeningRight}\ \frac{\{P\}\,\mathtt{s}\,\{Q\} \qquad Q \rightarrow R}{\{P\}\,\mathtt{s}\,\{R\}}$$

$$\text{oracle}\ \frac{}{P} \quad (P \text{ any valid first-order formula})$$

Figure 17.1 Rules of standard Hoare calculus.

17.4 Hoare Logic with Updates

The standard formulation of Hoare logic in Figure 17.1 has a number of *drawbacks* in usability that are particularly problematic when used for teaching purposes:

- The assignment rule computes directly the weakest preconditions from a given postcondition. Therefore the calculus requires to reason backwards through the program. This is obviously unnatural as it requires the user to understand a reached program state by "executing" the program in opposite direction to its control flow.
- The compositional rule splits the proof into two subgoals and requires to provide a formula describing the intermediate state reached in between both statements. The standard formalization of the Hoare calculus requires this formula to be guessed instead of being algorithmically inferred.

- The rules for the conditional statement and loops require to apply an additional weakening rule as preparatory step. As weakening is a purely first-order reasoning rule, it would be preferable to defer this step until the program is completely eliminated and to hide it as part of the first-order reasoning process.

We overcome these issues by introducing an explicit notation that describes finite parts of symbolic program states. This allows us to recast Hoare logic as forward symbolic execution.

17.4.1 State Updates

State updates in our logic are almost identical to those introduced in Chapter 3. But since we use scalar arrays and an implicit heap, the updates in this chapter are closer to the original variant used in previous versions of the KeY system [Beckert et al., 2007, Rümmer, 2006].

A (state) *update* is an expression of the form $\langle \mathit{Location} \rangle := \langle \mathit{FOLTerm} \rangle$. Actually, this is only the most simple form of an update, called *elementary update*. Complex updates are defined inductively: if $\mathcal{U}$ and $\mathcal{V}$ are updates, then so are $\mathcal{U}, \mathcal{V}$ (*sequential update*), and $\mathcal{U} \parallel \mathcal{V}$ (*parallel update*) and $\mathcal{U}(\mathcal{V})$ (*update application*). Sequential updates as an explicit construct are available only in KeY-Hoare, but not in JavaDL.

The most important kind of update is the parallel update. Consider a parallel update of the form $\mathcal{U} = l_1 := t_1 \| \cdots \| l_m := t_m$. Let (D,I) be a first-order model with domain D, interpretation I and let β denote a variable assignment. Assume a given program state s. Then the update takes us into a state $s_{\mathcal{U}}$ such that:

$$s_{\mathcal{U}}(\mathtt{x}) = \begin{cases} t_k & \text{if } \mathtt{x} = l_k \text{ and } l \notin \{l_{k+1}, \ldots, l_m\} \\ s(\mathtt{x}) & \text{if } \mathtt{x} \notin \{l_1, \ldots, l_m\} \end{cases} \tag{17.2}$$

$$s_{\mathcal{U}}(\mathtt{a})(i) = \begin{cases} t_k & \text{if } k = \max\{j \in \{1 \ldots m\} | \mathtt{a}[t'] = l_j \wedge val_{D,I,s,\beta}(t') = i\} \text{ exists} \\ s(\mathtt{a})(i) & \text{otherwise} \end{cases} \tag{17.3}$$

In words: the value of the locations occurring in $\mathcal{U}$ are overwritten with the corresponding right-hand side. Equation (17.2) defines the case for program variables and (17.3) for arrays. The condition $l \notin \{l_{k+1}, \ldots, l_m\}$ in (17.2) ensures that the rightmost update in $\mathcal{U}$ "wins" if the same location occurs more than once on the left-hand side in $\mathcal{U}$. Similarly, the max operator in the array case. Apart from that, all updates are executed in parallel.

Updates are similar to a preamble or fixture as used in unit testing [Myers, 2004]: a piece of code that gets you into a certain state. There is, however, a difference between updates and code: the right-hand side of an update may contain a symbolic first-order term, not merely a program expression. This feature is often used to initialize a program with "arbitrary, but fixed" values.

The significance of parallel updates lies in the following property, formally stated in Lemma 17.1 below. Let us call two updates $\mathcal{U}$ and $\mathcal{V}$ *equivalent* if $s_{\mathcal{U}} = s_{\mathcal{V}}$ for any state s. Then for each update $\mathcal{U}$ there exists an equivalent *parallel* update $\mathcal{V}$ of the form $l_1 := t_1 \parallel \cdots \parallel l_m := t_m$.

17.4.2 Hoare Triples with Update

We extend the classical Hoare triple to a Hoare quadruple by placing an update $\mathcal{U}$ in front of any program like this: $[\mathcal{U}]\pi$. If we are in state s the meaning is that the program is started in state $s_{\mathcal{U}}$. Within Hoare logic we use updates as follows (identical to JavaDL):

$$\{P\}\,[\mathcal{U}]\,\pi\,\{Q\} \tag{17.4}$$

where, P, Q, and π are as above, and $\mathcal{U}$ is an update over the signature of P and π. We enclose updates in square brackets to increase readability. Either one of $\mathcal{U}$ and π can be empty. The meaning of this *Hoare triple with update* is as follows: if s is any state satisfying the *precondition* P and we start π in $s_{\mathcal{U}}$, then, if π terminates, we end up in a final state that satisfies *postcondition* Q. The Hoare triple with updates is equivalent to the dynamic logic formula:

$$P \rightarrow \{\mathcal{U}\}[\pi]Q$$

17.4.3 Hoare Style Calculus with Updates

In Figure 17.2 we state the rules of a Hoare calculus with updates that has some new features when compared to the Hoare calculus of Figure 17.1:

- Composition is turned into left-to-right symbolic execution. Thereby the intermediate formula R is computed by rule application and needs not to be guessed. While this is sufficient for completeness, it does not subsume the composition rule as a whole as it lacks its implicit weakening.
- Weakening is delayed until after all program rules have been applied and becomes part of first-order verification condition checking.
- We use updates for computing the result of assignments.

One advantage of weakest precondition calculation [Dijkstra, 1976] as well as backward-execution style Hoare calculus is that an assignment can be computed by simple substitution and no renaming of old variables is necessary. The price to be paid for that is the not very intuitive backward-execution of programs. The KeY program logic uses updates to achieve weakest precondition computation with *forward* symbolic execution. In our eyes, this is a major pedagogical advantage: not only follows program rule application the natural execution flow in imperative

programs, but the whole proof process is compatible with established paradigms such as symbolic debugging (see Chapter 11).

$$\mathsf{assignment}_{pv}\ \frac{\{P\}\,[\mathscr{U}, \mathtt{x} := \mathtt{e}]\,\mathtt{s}\,\{Q\}}{\{P\}\,[\mathscr{U}]\,\mathtt{x=e;s}\,\{Q\}} \qquad \mathsf{assignment}_{arr}\ \frac{\{P\}\,[\mathscr{U}, \mathtt{a[i]} := \mathtt{e}]\,\mathtt{s}\,\{Q\}}{\{P\}\,[\mathscr{U}]\,\mathtt{a[i]=e;s}\,\{Q\}}$$

$$\mathsf{exit}\ \frac{\vdash P \rightarrow \mathscr{U}(Q)}{\{P\}\,[\mathscr{U}]\,\{Q\}} \qquad \mathsf{skip}\ \frac{\{P\}\,[\mathscr{U}]\,\mathtt{s}\,\{Q\}}{\{P\}\,[\mathscr{U}]\,\mathtt{;s}\,\{Q\}}$$

$$\mathsf{conditional}\ \frac{\{P \land \mathscr{U}(\mathtt{b} \doteq \mathtt{true})\}\,[\mathscr{U}]\,\mathtt{s1;s}\,\{Q\} \qquad \{P \land \mathscr{U}(\mathtt{b} \doteq \mathtt{false})\}\,[\mathscr{U}]\,\mathtt{s2;s}\,\{Q\}}{\{P\}\,[\mathscr{U}]\,\mathtt{if(b)\{s1\}else\{s2\}s}\,\{Q\}}$$

$$\mathsf{loop}\ \frac{\vdash P \rightarrow \mathscr{U}(I) \qquad \{I \land \mathtt{b} \doteq \mathtt{true}\}\,[]\,\mathtt{s1}\,\{I\} \qquad \{I \land \mathtt{b} \doteq \mathtt{false}\}\,[]\,\mathtt{s}\,\{Q\}}{\{P\}\,[\mathscr{U}]\,\mathtt{while(b)\{s1\}s}\,\{Q\}}$$

Figure 17.2 Rules of Hoare calculus with updates.

In the KeY logic as well as in the presented version of Hoare logic the rules have a "local" flavor in the sense that each judgment (i.e., node) in the proof tree relates to an elementary symbolic state transition during program execution.

We use the same conventions for schematic variables as above, but in addition, let $\mathscr{U}$ be an update and s is either a statement or the empty string. The rules are depicted in Figure 17.2. Let us briefly discuss each of them.

The *assignment* rules become easy as assignments are directly turned into updates. We can turn the whole assignment into an update in a single step, because in our simple language expressions have no side effects. Hence, we do not need to introduce temporary variables to capture expression evaluation as in JavaDL. The same holds for guards. Because we moved composition of substitutions into updates, we can now evaluate programs left-to-right. The weakest precondition calculation is moved into the update rules (see Figure 17.3 below).

There is one new rule called exit that is applied when a program is fully symbolically executed. At this point, the update is applied which computes the weakest precondition of the symbolic program state $\mathscr{U}$ with respect to the postcondition Q. Then it is checked whether the given precondition implies the weakest precondition. The premise of the exit rule (as well as the left-most premise of the loop rule) are purely first-order verification conditions. This is indicated by a turnstile in order to make clear that we left the language of Hoare triples.

The conditional rule simply adds the guard expression as a branch condition to the precondition. Of course, we must evaluate the guard in the current state $\mathscr{U}$. As mentioned above, this requires expressions to have no side effects. It has the advantage that path conditions can easily be read off each proof node.

The loop rule is a standard invariant rule. We exploit again that expressions have no side effects, but also that we have no reference types. The chosen formulation stresses the analogies to the conditional rule. The first premise says that the precondition must be strong enough to ensure that the invariant holds after reaching the state at the beginning of the loop. In the second premise we are not allowed to use P, because P might have been affected by executing $\mathscr{U}$. In addition, we must reset the update to the empty one. In other words, started in *any* state where the loop invariant and condition hold the invariant must hold again after execution of the loop body. In practice, one

uses as a starting point for the invariant those parts of P that are unaffected by $\mathscr{U}$. In those parts that *are* modified, one typically generalizes a suitable term and adds that to the invariant. More advice on how to choose a loop invariant can be found in Section 16.3.

17.4.4 Rules for Updates

We still need rules that handle our explicit state updates. Specifically, we need to (i) turn sequential into parallel updates (Section 17.4.1) and (ii) apply updates to terms, formulas, and to other updates. For the first task we use a Lemma from [Rümmer, 2006] (in specialized form):

Lemma 17.1. *For any updates $\mathscr{U}$, $\mathtt{x} := t$ and $\mathtt{a}[t_{idx}] := t$ the updates*

- $\mathscr{U}, \mathtt{x} := t$ and $\mathscr{U} \parallel \mathtt{x} := \mathscr{U}(t)$ *are equivalent;*
- $\mathscr{U}, \mathtt{a}[t_{idx}] := t$ and $\mathscr{U} \parallel \mathtt{a}[\mathscr{U}(t_{idx})] := \mathscr{U}(t)$ *are equivalent.*

The resulting rule is depicted together with the other update application rules in Figure 17.3. These are rewrite rules that can be applied whenever they match. We use the same schematic variables as before and, in addition, t, t_{idx} are first-order terms, $\mathscr{P}$ is a parallel update of the form $l_1 := t_1 \parallel \cdots \parallel l_m := t_m$, y is a first-order variable, F is a rigid n-ary function or predicate symbol, $\Box$ is a propositional connective, and λ is a quantifier.

$$\mathscr{U}, \mathtt{x} := t \rightsquigarrow \mathscr{U} \parallel \mathtt{x} := \mathscr{U}(t) \qquad \mathscr{U}, \mathtt{a}[t_{idx}] := t \rightsquigarrow \mathscr{U} \parallel \mathtt{a}[\mathscr{U}(t_{idx})] := \mathscr{U}(t)$$

$$\mathscr{P}(\mathtt{x}) \rightsquigarrow \begin{cases} t_k & \text{if } \mathtt{x} = l_k \text{ and } l \notin \{l_{k+1}, \ldots, l_m\} \\ \mathtt{x} & \text{if } \mathtt{x} \notin \{l_1, \ldots, l_m\} \end{cases} \qquad \mathscr{U}(y) \rightsquigarrow y$$

$$\mathscr{P}(\mathtt{a}[t_{idx}]) \rightsquigarrow \text{if } \mathscr{P}(t_{idx}) \doteq t_{idx_{i_k}} \text{ then } t_{i_k} \text{ else } \ldots \text{else if } \mathscr{P}(t_{idx}) \doteq t_{idx_{i_1}} \text{ then } t_1 \text{ else } \mathtt{a}[\mathscr{P}(t_{idx})]$$
$$\text{for } \mathscr{P} \downarrow \mathtt{a} = (i_1, \ldots, i_k)$$

$$\mathscr{U}(F(t_1, \ldots, t_n)) \rightsquigarrow F(\mathscr{U}(t_1), \ldots, \mathscr{U}(t_n)) \qquad \mathscr{U}(P \Box Q) \rightsquigarrow \mathscr{U}(P) \Box \mathscr{U}(Q)$$
$$\mathscr{U}(\lambda y.\, P) \rightsquigarrow \lambda y.\, \mathscr{U}(P),\ y \notin fv(\mathscr{U})$$

Figure 17.3 Rewrite rules for update computation.

The top row in Figure 17.3 contains the rules that turn sequential updates into parallel updates. The second row contains rules for applying updates to program and to first-order variables. There is a strong similarity between the first rule and the semantics definition of an update (17.2) on p. 576, while first-order variables are rigid and can never be changed by an update.

The third row shows the rule for applying a parallel update to an array access term. This rewrite rule is more complex as we cannot syntactically decide whether two array access expressions refer to the same location and we need in addition to compare the indices for equality. This comparison manifests itself in a cascade of conditional terms that check which (if any) elementary update of the parallel update is applicable. The comparisons must be performed backwards, because of

the last-one-wins semantics of updates (corresponding to the max operator in the semantics definition of updates (17.3) on p. 576). In the rule we denote with $\mathscr{P}\downarrow\mathtt{a}$ the tuple $(i_1,\ldots,i_k)$ such that $i_z \in \mathscr{P}\downarrow\mathtt{a} \Leftrightarrow l_{i_z} = \mathtt{a}[t_{idx_{i_z}}]$ for some term $t_{idx_{i_z}}$ and such that from $z_1 < z_2$ it follows that $i_{z_1} < i_{z_2}$ (with $0 < z_1, z_2 \leq k$). Intuitively, $\mathscr{P}\downarrow\mathtt{a}$ enumerates (order-preserving) all updates to array a by the parallel update $\mathscr{P}$.

Example 17.2. We demonstrate briefly how an application of a parallel update to an array access term is rewritten. Given the parallel update

$$\mathscr{V} := \mathtt{a[i]} := 3 \parallel \mathtt{m} := 5 \parallel \mathtt{a[j]} := 4 \parallel \mathtt{b[m]} := 10$$

The term $\mathscr{V}(\mathtt{a[m]})$ becomes

$$\text{if } \mathscr{V}(\mathtt{m}) \doteq \mathtt{j} \text{ then } 4 \text{ else if } \mathscr{V}(\mathtt{m}) \doteq \mathtt{i} \text{ then } 3 \text{ else } \mathtt{a}[\mathscr{V}(\mathtt{m})]$$

with $\mathscr{V}\downarrow\mathtt{a} = (1,3), l_1 = \mathtt{a[i]}$ $(t_{idx_1} = \mathtt{i})$ and $l_3 = \mathtt{a[j]}$ $(t_{idx_3} = \mathtt{j})$. Further applications of the update rewrite rules results in

$$\text{if } 5 \doteq \mathtt{j} \text{ then } 4 \text{ else if } 5 \doteq \mathtt{i} \text{ then } 3 \text{ else } \mathtt{a[5]}$$

It can be easily seen that the resulting conditional term evaluates to 4, if for instance $\mathtt{i} \doteq \mathtt{j} \doteq \mathscr{V}(m)(=5)$ holds which is according to the last one-wins semantics.

The fourth and fifth row contain rules for complex terms and for formulas. These are merely homomorphism rules propagating the update to the subterms/-formulas. In quantified formulas, again, first-order variables cannot be affected, but as they may occur in updates one has to ensure that no name clashes occur ($fv(\mathscr{U})$ returns the set of first-order variables not bound in $\mathscr{U}$). Update application can be seen as substitution of program variables with their new values with additional aliasing checks in case of arrays.

There is no rule to apply updates to programs. Updates accumulate during the reasoning process until symbolic execution of the target program terminates. Applying the update to the postcondition Q then computes its weakest precondition with respect to the taken path condition.

17.5 Using KeY-Hoare

We illustrate how to prove correctness of a program using the KeY-Hoare tool along a small example. Consider the program searchMax

```
max = a[0];
i = 1;
while (i < len) {
        if (max < a[i]) {
                max = a[i];
        } else {}
```

```
        i = i + 1;
}
```

which retrieves the value of a maximal element of the first `len` elements of an array `a`. The determined maximal value is stored in program variable `max`. We observe that the intended functionality assumes implicitly that the value of program variable `len` is at least one as otherwise no maximal element exists. Formalizing this natural language specification yields the following initial Hoare triple with updates (where the initial update is empty):

$$\{\texttt{len} >= 1\}\,[]\,\texttt{searchMax}\,\{\backslash\text{forall } \texttt{int}\ j\ (j >= 0 \land j < \texttt{len} \rightarrow \texttt{max} >= \texttt{a}[j]);\}$$

Such an initial Hoare triple can be easily specified in a text-based format (described in Section 17.7.3) and loaded into the KeY-Hoare system as a proof obligation.

Moving the mouse pointer over the displayed Hoare triple allows the user to apply the calculus rules described in Section 17.4.3 using a simple point-to-click interface. After clicking on the highlighted program (incl. the preceding updates for some rules) a popup menu with all applicable rules is shown (see screenshot below). There is exactly one applicable rule for each program construct and the system offers exactly this rule:[1] by applying the program rules the user symbolically executes the program step by step. The only nontrivial interaction is to provide the loop invariant for the loop rule. In the example entering the invariant $\backslash\text{forall } \texttt{int}\ j\ (j >= 0 \land j < \texttt{i} \rightarrow \texttt{max} >= \texttt{a}[j]);$ suffices to close the proof.

Whenever first-order verification conditions are reached, the system offers a rule Update Simplification that applies the update rules from Figure 17.3 automatically. At this point, the user can opt to push the green Go button ▶. Then the built-in first-order theorem prover tries to establish validity automatically. For simple problems discussed in the introductory courses, such as `searchMax`, this works quite well and the reason that a proof could not be found is rarely rooted in insufficient reasoning power of the underlying theorem prover. In the majority of cases an unclosable proof points to a problem in the code or specification. It is worth to mention that the problem is at least as often a too weak or wrong specification as it is a bug in the code.

Inspecting open proof goals usually gives a good hint where to find the bug or which (implicit) assumption is missing. The system allows the student to follow the symbolic execution of the program and to concentrate on getting invariants and specification right without needing to deal with first-order reasoning which is done in the background by the system. It is possible to inspect and undo previous proof steps as well as to save and load proofs.

[1] The other rules displayed are propositional rules that can be applied anytime and they can be ignored.

17.6 Variants of the Hoare Logic with Updates

17.6.1 Total Correctness

The previous sections focused on partial correctness of programs, i.e., termination was ignored and nonterminating programs satisfied their specification trivially. In this section we explain the necessary changes to our calculus such that for the Hoare triple

$$\{P\}\,[\mathscr{U}]\,\pi\,\{Q\}$$

to be valid, the program π must not only adhere to its functional specification but also terminate when started in an initial state $s_{\mathscr{U}}$ such that s satisfies P. In other words, to establish the validity of a total correctness Hoare triple, we have to prove *total correctness* of the program π. Total correctness is supported by our tool KeY-Hoare.

The calculus rules for total correctness are identical to those presented in Section 17.4 except for the loop invariant rule. The new version of the loop invariant rule is given in Figure 17.4. To ensure that a while loop terminates one has to provide a term *dec* which decreases strictly monotonic after each execution of the loop body, but stays nonnegative. The first branch of the while rule now ensures additionally that the given term is initially greater or equal to zero. The second branch checks also that after each loop iteration *dec* is strictly smaller than before, but still nonnegative. To be able to access the old value of *dec*, the rule introduces a fresh (not yet used) rigid function *oldDec* to capture the value of *dec* at the beginning of the loop iteration.

$$\mathsf{loop_T}\ \frac{\begin{array}{l}\vdash\ P \rightarrow \mathscr{U}(I \land dec >= 0)\\ \{I \land \mathtt{b} \doteq \mathtt{true} \land oldDec \doteq dec\}\,[]\,\mathtt{s1}\,\{I \land dec >= 0 \land dec < \mathrm{oldDec}\}\\ \{I \land \mathtt{b} \doteq \mathtt{false}\}\,[]\,\mathtt{s}\,\{Q\}\end{array}}{\{P\}\,[\mathscr{U}]\,\mathtt{while(b)\{s1\}s}\,\{Q\}}$$

where *oldDec* is a new function symbol of arity $size(fv(dec))$ ($fv(dec)$ denotes the set of free first-order variables in *dec*)

Figure 17.4 Loop invariant rule for total correctness.

The total correctness proof for the maximum search example of Section 17.5 is almost identical for total correctness, except that when applying the loop invariant rule the decreasing term *dec* has to be provided in addition to the loop invariant. For the example, it suffices to instantiate *dec* with the expression `len-i`.

17.6.2 Worst-Case Execution Time

The most advanced variant of our Hoare logic with updates is concerned with reasoning about simple properties about the worst-case execution time (WCET) of a program [Harmon and Klefstad, 2007]. The calculus to reason about WCET is presented in Figure 17.5.

$$\mathsf{assignment}_{ET}\ \frac{\{P\}\,[\mathcal{U}, \mathtt{x} := \mathtt{e}, eT := eT+1]\,\mathtt{s}\,\{Q\}}{\{P\}\,[\mathcal{U}]\,\mathtt{x=e;s}\,\{Q\}}$$

$$\mathsf{skip}_{ET}\ \frac{\{P\}\,[\mathcal{U}, eT := eT+1]\,\mathtt{s}\,\{Q\}}{\{P\}\,[\mathcal{U}]\,\mathtt{;s}\,\{Q\}} \qquad \mathsf{exit}_{ET}\ \frac{\vdash\ P \to \mathcal{U}(Q)}{\{P\}\,[\mathcal{U}]\ \{Q\}}$$

$$\mathsf{conditional}_{ET}\ \frac{\begin{array}{c}\{P \wedge \mathcal{U}(\mathtt{b} \doteq \mathtt{true})\}\,[\mathcal{U}, eT := eT+1]\,\mathtt{s1;s}\,\{Q\}\\ \{P \wedge \mathcal{U}(\mathtt{b} \doteq \mathtt{false})\}\,[\mathcal{U}, eT := eT+1]\,\mathtt{s2;s}\,\{Q\}\end{array}}{\{P\}\,[\mathcal{U}]\,\mathtt{if(b)\{s1\}else\{s2\}s}\,\{Q\}}$$

$$\mathsf{loop}_{ET}\ \frac{\begin{array}{l}\vdash\ P \to \mathcal{U}(I \wedge dec >= 0)\\ \{I \wedge \mathtt{b} \doteq \mathtt{true} \wedge \mathrm{oldDec} \doteq dec\}\\ \qquad\qquad [eT := eT+1]\{\mathtt{s1}\}\{I \wedge dec >= 0 \wedge dec < \mathrm{oldDec}\}\\ \{I \wedge \mathtt{b} \doteq \mathtt{false}\}\,[eT := eT+1]\,\mathtt{s}\,\{Q\}\end{array}}{\{P\}\,[\mathcal{U}]\,\mathtt{while(b)\{s1\}s}\,\{Q\}}$$

where

- `oldDec` is a new function symbol of arity $size(fv(dec))$ as above
- eT stands for the reserved program variable `executionTime` which does not occur elsewhere

Figure 17.5 Loop invariant rule for execution time reasoning

The basic idea for the WCET-calculus is taken from [Hunt et al., 2006]. To keep track of the number of executed instructions, an implicitly declared global program variable `executionTime` is introduced. This program variable cannot be directly accessed or modified by a program, but is increased implicitly as a side effect when symbolically executing a statement. Standard statements like assignment cause the counter to be increased by one, while in case of a branching statement like a conditional or a loop the guard evaluation costs an additional unit.

Assume a program `countdown` which decreases the counter variable `timer` to zero. A Hoare triple containing a WCET specification to be proven is shown in Figure 17.6.

```
{ startVal >= 0 & executionTime = 0}

[timer := startVal]

while (timer>0) {
  timer = timer -1;
}

{ timer = 0 & executionTime = 2*startVal + 1}
```

Figure 17.6 Worst-case execution time specification

The functional part of the specification states that if the program is started in an initial state with program variable `timer` set to a nonnegative value then in its final state the program variable `timer` has the value zero. We use a rigid constant

symbol `startVal` to capture the initial value of `timer` so that we can refer to it in the postcondition.

The WCET part of the specification is added to the pre- and postcondition as a simple conjunction. The precondition states additional knowledge about the initial value of the execution time counter. In most cases one requires that the initial value of the `executionTime` counter is either equal to a fixed nonnegative but unknown value, or as in this example, zero.

The postcondition can be used to specify either the exact number of execution steps performed by the algorithm (as done here) or it can simply state an upper bound for the expected execution time. The countdown algorithm is expected to require $2 * \texttt{startVal} + 1$ time units until completion. The justification for this number is as follows: the `timer` is decreased by one in each loop iteration, hence, there are `startVal` loop iterations where each iteration costs 2 time units (evaluation of the loop guard plus decreasing the `timer` variable) plus one additional cost unit for the final evaluation of the loop guard which terminates the loop.

Again the only nontrivial interaction is the application of the loop invariant rule. Providing

```
timer>=0 & executionTime = 2*(startVal-timer)
```

as loop invariant and `timer` as decreasing term allows us to close the proof.

17.7 A Brief Reference Manual for KeY-Hoare

We conclude this chapter with a brief reference manual for KeY-Hoare to describe the installation, input format and usage of the tool in detail.

17.7.1 Installation

The tool KeY-Hoare is available at www.key-project.org/download/hoare in three versions: (i) a source code version, (ii) a bytecode version and as (iii) *Java Web start* version which allows the user to install and start KeY-Hoare with a single click. We describe here only the Web Start installation, detailed installation instructions for the other options can be found on the website.

Java Web Start is included in all recent JDK and JRE distributions of Java. It provides a simple way to run and install Java applications. If Java is installed, almost all web browsers know how to handle Java Web Start URLs out-of-the-box; if not, the file type `jnlp` needs to be associated with the application `javaws`. Otherwise a click on the Web Start link on the mentioned website loads and starts KeY-Hoare. We use a self-signed certificate, which is not trusted by a standard Java installation. You need to accept our certificate as an exception in the dialog box that pops up.

Instead of using a browser Java Web Start can also be used from the command line:

```
javaws http://www.key-project.org/download/hoare/download/webstart/KeY-Hoare.jnlp
```

After the first start no internet connection is required and KeY-Hoare can be started in offline mode by executing `javaws -viewer` and selecting the KeY-Hoare entry in the list of available applications.

17.7.2 Formula Syntax

The predicate symbols and function symbols `>`, `>=`, `<`, `<=` and `=` as well as `+`, `-`, `*`, `/` and `%` are reserved symbols for which the usual infix notation and precedence rules are in place.

These arithmetic relations and operations are supported with their canonical signature and meaning. The modulo operation is defined as $x\%y := x - (x/y) * y$. Consequently, the values of the terms $0\%y$ and $x\%0$ are undefined. As in JavaDL, undefinedness is modeled by underspecification. This means that an integer value specified as $x/0$ is a valid term/expression whose value is not specified a priori and may be assigned a different integer value by different first-order interpretations.

The concrete syntax of propositional connectives is `!`, `&`, `|`, `->`, `<->` with their obvious meaning. First-order quantified formulas are written as follows:

⟨*QuantifiedFormula*⟩ ::= ⟨*Quantifier*⟩ ⟨*Type*⟩ ⟨*LogicalVariable*⟩ ; ⟨*FOLFormula*⟩
⟨*Quantifier*⟩ ::= **`\forall`** | **`\exists`**

Example 17.3. The following formula expresses that any common divisor `x` of the integers `a` and `b` is as well a divisor of the integer `r`.

```
\forall int x; ((x > 0 & a % x = 0 & b % x = 0) -> r % x = 0))
```

17.7.3 Input File Format

Input files for KeY-Hoare must have `.key` or `.proof` as file extension. By convention `.key` files contain only the problem specification, i.e., the Hoare triple to be proven together with the necessary program variable, array and user-defined rigid function declarations. In contrast, `.proof` files include in addition (partial) proofs for the specified problem and are created when saving a proof.

The input file grammar is given in Figure 17.8. As an example the input file for the example `searchMax` is shown in Figure 17.7. An input file consists of four sections:

1. The section starting with keyword `\functions` declares all required rigid function symbols used, for example, to assign input program variables to an arbitrary but fixed value as described in Section 17.4.1. In Figure 17.7 this section is empty.

```
\functions {}

\arrays {
        int[] a;
}

\programVariables {
        int i;
        int len;
        int max;
}

\hoare {

        { len >=1 }

        \[{
                max = a[0];
                i = 1;
                while (i<len) {
                        if (max < a[i]) {
                                max = a[i];
                        } else {}
                        i = i + 1;
                }
        }\]

        { \forall int j; (j>=0 & j<len -> max >=a[j]) }

}
```

Figure 17.7 Input file for the `searchMax` example.

2. The next section starting with keyword `\arrays` declares the arrays that may be used by the program or specification. In Figure 17.7 this section declares one integer typed array `a`. In addition to integer typed arrays, Boolean typed arrays are available.
3. The section starting with keyword `\programVariables` declares *all* program variables used in the program. Local variable declarations within the program are not allowed. Multiple declarations are permitted.
4. The section starting with keyword `\hoare` contains the Hoare triple with updates to be proven valid, i.e., the program and its specification. If total correctness or worst-case execution time of a program should be proven, the keyword `\hoare` is replaced by the keyword `\hoareTotal`, respectively, `\hoareET`.

```
⟨InputFile⟩ ::= ⟨Functions⟩? ⟨ProgramVariables⟩? ⟨HoareTriple⟩?

⟨Functions⟩ ::= \functions '{'⟨FunctionDeclaration⟩*'}'
⟨FunctionDeclaration⟩ ::= ⟨Type⟩ ⟨Name⟩ ( '(' ⟨Type⟩(','⟨Type⟩)*')')?';'

⟨Arrays⟩ ::= \arrays '{'⟨ArrayDeclaration⟩*'}'
⟨ArrayDeclaration⟩ ::= ⟨Type⟩[] ⟨Name⟩(',' ⟨Name⟩)*';'

⟨ProgramVariables⟩ ::= \programVariables '{'⟨ProgramVariableDeclaration⟩*'}'
⟨ProgramVariableDeclaration⟩ ::= ⟨Type⟩ ⟨Name⟩(',' ⟨Name⟩)*';'

⟨HoareTriple⟩ ::= ( \hoare | \hoareTotal | \hoareET ) '{'
                      ⟨PreCondition⟩ ⟨Update⟩ ⟨Program⟩ ⟨PostCondition⟩
                  '}'

⟨PreCondition⟩ ::= ⟨FOLFormula⟩

⟨Update⟩ ::= '[' (⟨AssignmentPair⟩ (\|⟨AssignmentPair⟩)*)? ']'
⟨AssignmentPair⟩ ::= ⟨Name⟩ ':=' ⟨FOLTerm⟩

⟨Program⟩ ::= '\[{' ⟨WhileProgram⟩ '}\]'
⟨PostCondition⟩ ::= ⟨FOLFormula⟩

⟨Type⟩ ::= int | boolean
⟨Name⟩ ::= character sequence not starting with a number
```

Figure 17.8 Input file grammar

17.7.4 Loading and Saving Problems and Proofs

After starting KeY-Hoare (see Section 17.7.1) the prover window becomes visible (the screenshot on p. 580 is displayed in enlarged form in Figure 17.9). The prover window consists of a menu- and toolbar, a status line and a central part split into a left and a right pane. The upper left pane displays a list of all loaded problems. The lower left pane offers different tabs for proof navigation or strategy settings. The right pane displays the currently selected subgoal or an inner proof node.

Before we explain the various subpanes in more detail, the first task is to load a problem file. This can be done either by selecting `Load` in the `File` menu or by clicking on the icon in the toolbar (reloads the most recently loaded problem). In the file dialogue window that pops up the users can choose one of the examples provided (e.g., `searchMax.key`) or their own files.

After the file has been loaded the right pane of the prover window displays the Hoare triple as specified in the input file. The proof tab in the left pane should display the proof tree consisting of a single node. The first time during a KeY-Hoare session when a problem file is loaded the system loads a number of libraries which takes a few seconds.

(Partial) proofs can be saved at any time by selecting the menu item `Save` in the `File` menu and entering a file name ending with the file extension `.proof`.

17.7.5 Proving

First a few words on the various parts of the prover window. The upper part of the left pane displays all loaded problems. The lower part provides some useful tabs:

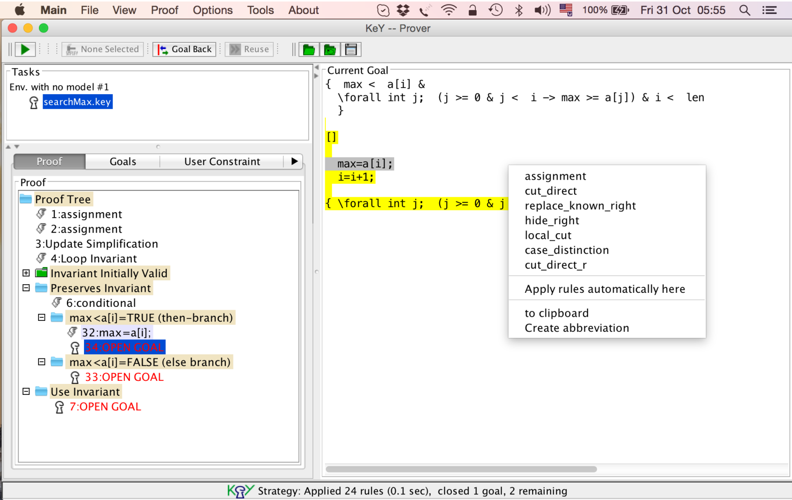

Figure 17.9 Screen shot of KeY-Hoare system

The Proof tab shows the constructed proof tree. A left click on a node updates the right pane with the node's content (a Hoare triple with updates). Using a right click offers a number of actions like pruning, searching, etc.

The Goals tab lists all open goals, i.e., the leaves of the proof tree that remain to be justified.

The Proof Search Strategy tab allows one to tune automated proof search. In case of KeY-Hoare only the maximal number of rule applications before an interactive step is required, and (de-)activation of the `autoresume` mode can be adjusted.

All other tabs are not of importance for KeY-Hoare.

The right pane displays the content of a proof node in two different modes depending on whether the node is (a) an inner node or a leaf justified by an axiom or (b) it represents an open proof goal.

(a) Inner Node View is used for inner nodes of the proof tree. It highlights the formula which had been in focus at time of rule application as well as possible necessary side formulas. The applied rule is listed on the bottom of the view.

(b) Goal View is used when an open goal is selected. This view shows the Hoare triple to be proven and allows the user to apply rules. Moving the mouse cursor

over the expressions within the node highlights the smallest enclosing term or formula at the current position. A left click creates a popup window showing all applicable rules for the currently highlighted formula or term.

17.7.6 Automation

A few remarks on automation: in our examples, the necessary interactive proof steps consisted of manual application of program rules and invocations of the strategies to simplify/prove first-order problems. To avoid having to start the automatic strategies manually one can activate the `autoresume` mode. This will invoke the strategies on all open goals after each manual rule application and simplify them as far as possible. In standard mode they will not apply program rules.

For pedagogic reasons the application of the program rules which execute a program symbolically are not performed automatically, but need to be applied interactively. Except for the loop invariant rule which requires the user to provide a loop invariant (and, if termination plays a role, a decreasing term) all other rules could be applied automatically, because they are deterministic and require no input. To enable automatic application of program rules (except loop) one can set the environment variable `TEACHER_MODE` to an arbitrary value. With both options (autoresume and teacher mode) activated the only required interaction in the examples above is the application of the loop invariant.