Hoare Logic With State Updates - Rules

Ramón Béjar Torres

March 21, 2024

1 Rules for Proving Partial Program Correctness

This document is a very brief summary of the formal verification method introduced by Reiner Hähnle and Richard Bubel [2] that we use in the first part of this subject. Remember, this is only a summary, for the complete description of the verification method, check the mentioned reference. You have also available solved examples of verification of algorithms in this same course site.

1.1 Assignment

The basic instruction of our imperative language is the assignment. Given the current state update $\mathcal{U}$ [1], this rules creates a new state update from $\mathcal{U}$ and the assignment instruction `x = e;` we are processing. This is the rule:

$$\text{assignment}\ \frac{\{P\}\,[\mathcal{U}, \texttt{x} := \texttt{e}]\,\pi\,\{Q\}}{\{P\}\,[\mathcal{U}]\,\texttt{x = e;}\,\pi\,\{Q\}}$$

How it works:

- Turn assignment into update and append sequentially
  - Important that `e` has no side effects
  - `e` can be evaluated as FOL term
- Turn sequential into parallel update then simplify

Transforming the sequential update to a parallel one is essential, because at the end we need to obtain a final state representation that gives independent and no ambiguous information of all the variables of the program, and this is achieved when the state is a parallel one, so we do not have to resolve any sequence of assignments for getting the final value of the variables.

[1] The state update $\mathcal{U}$ summarizes the symbolic state reached **so far** by the executed part of the program. Remember that we work by performing a symbolic forward execution of the program, so that the final goal is to obtain a final symbolic state representation of the effect of the program after it is completely executed.

1.2 Exit

After a program is fully symbolically executed, we end up with a final state update. Then, we have to apply the Exit rule, for verifying whether the final state update satisfies the postcondition if the precondition is true. That is, if the program is correct, the precondition must imply the postcondition after applying to the postcondition the final state update that summarizes what the execution of the program has achieved. This is the rule:

$$\text{exit}\ \frac{\vdash\ P \mathrel{->} \mathcal{U}(Q)}{\{P\}\,[\mathcal{U}]\ \{Q\}}$$

How it works:

- Applied only when original program is fully symbolically executed
- "Precondition implies postcondition in final state of the original program, which is now summarized by $\mathcal{U}$"
- The meaning of $\mathcal{U}(Q)$ is to apply $\mathcal{U}$ to Q:
  - If $\mathtt{x} := t$ is an atomic update in $\mathcal{U}$ then replace each occurrence of $\mathtt{x}$ in Q with t
  - Assume that $\mathcal{U}$ is a parallel update
- The resulting formula $(P \mathrel{->} \mathcal{U}(Q))$, is a FOL formula, so it can be handed by an automated, or semi-automated, theorem prover for FOL formulas. That is, a program for verifying whether a FOL formula is valid. [2] The resulting formula will be valid (satisfiable by **all** possible interpretations of the formula) if and only if the original program satisfies the postcondition for **any possible** execution of the program that satisfies the precondition.

1.3 Conditional

When the symbolic execution reaches a conditional sentence, the symbolic execution must branch on two different cases, so our proof of correctness is split into two proofs, one corresponding to the conditional being TRUE and the other corresponding to the conditional being FALSE. This is the rule:

$$\text{conditional}\ \frac{\{P\ \&\ \mathcal{U}(\mathtt{b=TRUE})\}\,[\mathcal{U}]\,\pi_1\,\rho\,\{Q\} \qquad \{P\ \&\ \mathcal{U}(\mathtt{b=FALSE})\}\,[\mathcal{U}]\,\pi_2\,\rho\,\{Q\}}{\{P\}\,[\mathcal{U}]\ \mathbf{if}\,(\mathtt{b})\{\pi_1\}\mathbf{else}\{\pi_2\}\ \rho\,\{Q\}}$$

How it works:

[2] Remember that validity of formulas in FOL is a semi-decidable problem, so we do not have provers that give always a correct answer for any formula. More concretely, we have theorem provers (like the one used by `Key-Hoare`) that if the FOL formula, without fixing the meaning of any function symbols or constants, is valid, it will find a proof given enough time. But if the formula is not valid, it may run forever trying to find a proof without being able to discover that such proof does not exist.

- Case distinction necessary, because value of `b` symbolic
  - In general, Hoare calculus proofs are trees
- Important that `b` has no side effects
  - Can treat `b` as FOL Boolean term
- In premisses `b` must be evaluated in state $\mathcal{U}$
- $\mathcal{U}$(`b=TRUE`) and $\mathcal{U}$(`b=FALSE`) extend existing path condition P creating two new execution paths to explore (and to prove that they satisfy their specification).
  - Can simplify `b=TRUE` to `b` and `b=FALSE` to `!b`

1.4 Loops

When the symbolic execution reaches a loop, we must prove using the loop rule that after the loop finishes, **any possible state in which the loop can finish** will be a good starting point for executing the rest of the program and finally satisfying the postcondition of the program.

The specification of this special set of possible resulting states after the loop finishes will be captured by the use of two formulas: the loop guard formula (the boolean expression that becomes FALSE when the loop must finish its iterations) and a formula that will be provided by the user: the invariant of the loop (Inv). The invariant of the loop will capture all the possible states in which any iteration of the loop can be found. Thus, the set of possible resulting states after the loop finishes will be captured by the conjunction of the invariant formula and the negation of the loop guard formula.

This is the rule:

$$\text{loop}\ \frac{\begin{array}{cc} \vdash\ P \Rightarrow \mathcal{U}(Inv) & \text{(initially valid)} \\ \{Inv\ \&\ \texttt{b}\}\,[]\,\pi\,\{Inv\} & \text{(preserved)} \\ \{Inv\ \&\ \texttt{!b}\}\,[]\,\rho\,\{Q\} & \text{(use case)} \end{array}}{\{P\}\,[\mathcal{U}]\ \textbf{while}\ \texttt{(b)}\ \{\pi\}\,\rho\,\{Q\}}$$

How it works:

- The formula Inv is initially valid just before starting the loop, so the precondition of the program must imply the invariant formula after applyting to it the current state update.
- The validity of Inv is preserved by loop guard and body (if Inv is TRUE just before executing a loop iteration then it is also TRUE after the iteration finishes. Consequence: if Inv was true at start state of the loop, then it still holds after arbitrarily many loop iterations [3]. So, if the loop terminates at all, then Inv holds afterwards.

[3] Observe that this proof is actually a kind of induction proof for the property Inv considered as a function of the integer variables modified by the loop

- Thus, after the loop terminates, the set of possible resulting states satisfies the formula $\{Inv \;\&\; \texttt{!b}\}$. This formula will be considered as the new precondition for the rest of the program (ρ).

- Make sure to include the desired postcondition after loop into Inv (If your chosen invariant **is not strong enough**, you will not be able to prove the correctness of your program). In other words, your chosen invariant Inv must capture the relevant properties of the resulting states after loop finishes, such that

$$\{Inv \;\&\; \texttt{!b}\} \,[]\, \rho \,\{Q\}$$

  can be proven, thus proving that Inv & `!b` is a strong enough precondition to prove that the rest of the program ρ achieves what we want in the final postcondition Q.

2 Rules for Proving Total Program Correctness

2.1 Total Correctness of Loops

The only, but important, difference between the rules for proving partial or total correctness, for our simple imperative language, is the loop rule. We have to use a version of the loop rule that is able to prove not only that **when** the loop finishes the output is as expected, but also that it **always finishes** starting from any state from where the loop can start (from the states that satisfy $\{Inv \;\&\; \texttt{b}\}$).

So, in addition to preserve the Inv in each iteration of the loop, we need to prove that each iteration *shortens* the distance to the exit states (states where the loop guard condition b is false). This is done through the introduction of an integer function that will measure such distance:

1. Given a loop, the user has to provide a function $T(vars) \rightarrow \mathbb{Z}^+$ that is a measure of *how far* the loop is from an exit state
2. The function must be an expression that uses variables that are modified in the loop body.
3. If the measure T **is good** for the loop, and the loop finishes for every input, it allows to automatically prove that the loop finishes.

$$Loop_T \;\frac{\begin{array}{rc} \vdash\; P \dashrightarrow \mathcal{U}(Inv \;\&\; T >= 0) & (1) \\ \{Inv \;\&\; \texttt{b} \;\&\; oldT = T\}\,[]\,\pi\,\{Inv \;\&\; T < oldT \;\&\; T >= 0\} & (2) \\ \{Inv \;\&\; \texttt{!b}\}\,[]\,\rho\,\{Q\} & (3) \end{array}}{\{P\}\,[\mathcal{U}]\;\textbf{while}\;\texttt{(b)}\;\{\pi\}\,\rho\,\{Q\}}$$

How it works:

1. T will be a function expression that uses variables used in the loop body. This has to be provided by the user.
2. initial T value (before entering loop) non negative.
3. T decreases for each iteration performed, but never becomes negative. We define variable $oldT$ to be equal to the value of function T before performing a loop iteration to be able to specify that after one loop iteration the value of T has decreased.
4. So, at some iteration the loop guard will become FALSE, because otherwise it would not be able to preserve T always positive if it decreases at each iteration

3 Rules Summary

$$\text{assignment}\ \frac{\{P\}\,[\mathcal{U},\ \texttt{x} := \texttt{e}]\,\pi\,\{Q\}}{\{P\}\,[\mathcal{U}]\,\texttt{x = e;}\,\pi\,\{Q\}} \qquad \text{exit}\ \frac{\vdash\ P \mathrel{-\!\!>} \mathcal{U}(Q)}{\{P\}\,[\mathcal{U}]\ \{Q\}}$$

$$\text{conditional}\ \frac{\{P\ \&\ \mathcal{U}(\texttt{b=TRUE})\}\,[\mathcal{U}]\,\pi_1\,\rho\,\{Q\} \qquad \{P\ \&\ \mathcal{U}(\texttt{b=FALSE})\}\,[\mathcal{U}]\,\pi_2\,\rho\,\{Q\}}{\{P\}\,[\mathcal{U}]\ \textbf{if}\,\texttt{(b)}\{\pi_1\}\textbf{else}\{\pi_2\}\ \rho\,\{Q\}}$$

$$\text{loop}\ \frac{\vdash\ P \mathrel{-\!\!>} \mathcal{U}(\mathit{Inv}) \qquad \{\mathit{Inv}\ \&\ \ \texttt{b}\}\,[]\,\pi\,\{\mathit{Inv}\} \qquad \{\mathit{Inv}\ \&\ \texttt{!b}\}\,[]\,\rho\,\{Q\}}{\{P\}\,[\mathcal{U}]\,\textbf{while}\ \ \texttt{(b)}\ \ \{\pi\}\,\rho\,\{Q\}}$$

$$\mathit{Loop}_T\ \frac{\begin{array}{c}\vdash\ P \mathrel{-\!\!>} \mathcal{U}(\mathit{Inv}\ \&\ T >= 0) \\ \{\mathit{Inv}\ \&\ \ \texttt{b}\ \&\ \ \mathit{oldT} = T\}\,[]\,\pi\,\{\mathit{Inv}\ \&\ T < \mathit{oldT}\ \&\ T >= 0\} \\ \{\mathit{Inv}\ \&\ \texttt{!b}\}\,[]\,\rho\,\{Q\}\end{array}}{\{P\}\,[\mathcal{U}]\,\textbf{while}\ \ \texttt{(b)}\ \ \{\pi\}\,\rho\,\{Q\}}$$

4 Proving the Validity of FOL Formulas with KeY-Hoare

Once KeY-Hoare finally gets a FOL formula to be proved valid obtained with the Exit Rule and the application of the final state update to the postcondition, we get a FOL formula that we need to prove that is valid, that is, true in all the models of the formula (so meaning that any possible execution that satisfies the precondition also satisfies the postcondition).

KeY-Hoare does not perform any transformation to normal forms to use a simple calculus based on rules such resolution, but instead it works with the original formula and a set of inference rules based on so called **sequents**. A sequent is a formula of this form:

$$\phi_1, \ldots, \phi_m \Rightarrow \Phi_1, \ldots, \Phi_n$$

where the intuitive meaning of the formula is the following:

> Whenever *all* the ϕ_i of the antecedent are true then *at least* one of the Φ_j is true

Then, KeY-Hoare uses inference rules called sequent rules, that have this form:

$$\frac{\Gamma_1 \Rightarrow \Delta_1}{\Gamma_2 \Rightarrow \Delta_2}$$

where $\Gamma_1 \Rightarrow \Delta_1$ and $\Gamma_2 \Rightarrow \Delta_2$ are sequent formulas. Such rule denotes the following inference: To prove the sequent $\Gamma_2 \Rightarrow \Delta_2$, it is enough to prove the sequent $\Gamma_1 \Rightarrow \Delta_1$ [4] . For proving validity of formulas, any sequent rules used by KeY-Hoare must hold for all the models.

However, KeY-Hoare cannot simply use a set of classical FOL inference rules equivalent to the ones we know from our courses about computational logic, because there are some symbols and functions that have an intended fixed meaning in our formulas when obtained from the verification of programs of the WHILE language: the symbols and functions related to **integer arithmetic expressions**. So, we are not really interested in all the possible models, but in those models where the meaning of such symbols and functions is fixed to the one intended for integer arithmetic.

But can these special models, that capture the valid formulas of integer arithmetic, be captured with some special set of FOL inference rules ? The answer is NO. There is no finite set, or even a recursively enumerable (r.e.) set, of FOL inference rules that is able to prove any valid formula of integer arithmetic (even if they are formulas written in the language of FOL !!). This astonishing result is from Kurt Gödel [1], and it was a major breakthrough when it first appeared in a time where mathematicians like David Hilbert and philosophers

[4] The idea is that sequent $\Gamma_1 \Rightarrow \Delta_1$ must be, in some sense, a more simple sequent to be proved valid than $\Gamma_2 \Rightarrow \Delta_2$

like Bertrand Russell were trying to understand if complete axiomatizations of mathematical theories, like integer number theory, were really possible.

But although the result of Kurt Gödel is bad news for the automation of theorem proof search with computers, the fact is that we have very good axiomatizations of integer number theory (and many other mathematical theories) that although they are not complete (they cannot prove everything that is true in the theory), they are sound (any statement you can prove with your axiomatization is a theorem of the theory) and they can prove a huge amount of basic and not so basic theorems.

For example, consider the following small axiomatization for number theory, that is very similar to a small subset of the axioms that `KeY-Hoare` uses for proving the validity of formulas of integer number theory:

$$\forall x\ \neg(0 = x + 1) \quad (A1)$$

$$\forall x\ 0 + x = x \quad (A2)$$

$$\forall x\ x + 0 = 0 + x \quad (A3)$$

$$\forall x\ x + 1 = 1 + x \quad (A4)$$

$$\forall x \forall y\ (x + 1) + y = (x + y) + 1 \quad (A5)$$

$$\forall x \forall y \forall z\ (x + y) + z = x + (y + z) \quad (A6)$$

$$\forall x\ 0 * x = 0 \quad (A7)$$

$$\forall x \forall y\ (x + 1) * y = (x * y) + y \quad (A8)$$

For any FO formula $\phi(y)$ about integer numbers with y as a free variable:

$$\phi(0), (\forall y\ (\phi(y) \Rightarrow \phi(y + 1)) \Rightarrow \forall y\ \phi(y) \quad (A9)$$

From this set of axioms, observe that actually the last one is really an inference rule for proving properties (formulas) about natural numbers using induction on natural numbers [5]. It turns out that induction is necessary for proving fundamental theorems of integer number theory. For example, in the above axiomatization we have added the axioms:

$$\forall x\ x + 0 = 0 + x \quad (A3)$$

$$\forall x\ x + 1 = 1 + x \quad (A4)$$

But what about the general theorem about commutativity of addition ? That is, the formula

$$\forall y, \forall x\ x + y = y + x$$

[5] Observe that this axiom is not a single axiom, but an infinite set of axioms (one axiom for every possible FOL formula about integer numbers with a free variable y). We also say that it is an axiom scheme. But observe that this infinite set of axioms is recursively enumerable.

We can prove that this formula is true using the above axiomatization. That is, **derive that it is true assuming the above axioms are true**. This is a possible proof that uses the induction rule axiom (A9) on the integer number y:

1. The base case (when $y = 0$) is true by one of the axioms we have in our axiomatization:
$$\forall x \; x + 0 = 0 + x$$

2. Next, we prove the induction step (assuming $\forall y, \forall x \; x + y = y + x$ is valid we prove that $\forall y, \forall x \; x + (y + 1) = (y + 1) + x$ is also valid):
$$\begin{aligned} (y+1)+x &= (y+x)+1 \; (\textit{by axiom } A5) && (1)\\ &= (x+y)+1 \; (\textit{by induction hypothesis}) && (2)\\ &= x+(y+1) \; (\textit{by axiom } A6) && (3) \end{aligned}$$

So, by the induction rule axiom (A9) we have that the formula is true for any y.

`Key-Hoare` uses a much bigger set of axioms with many inference rules for being able to prove theorems about integer numbers with probably less steps than with the axiomatization we show here. In particular, commutativity of addition is indeed included as an axiom, so for that formula `KeY-Hoare` will answer that it is true without performing any inference steps.

Even if very powerful, the number of inference steps needed to prove even simple formulas can be quite large. For example, consider the following valid formula:

$$(p(a) \;\&\; na = a + 1 \;\&\; \forall x; \forall y; (p(x) \;\&\; y = x + 1 \; -> \; n(y))) -> n(na)$$

For proving the validity of this formula, `KeY-Hoare` performs the application of 26 inference steps in a single branch (a single sequence of inference steps). If you to want explore all the steps performed by `KeY-Hoare` to prove such formula, write it following the syntax of `Key-Hoare` in a file with extension `.key`:

```
\functions { int a;
             boolean p(int);
             boolean n(int); }

\programVariables { int na; }

\hoare {
{ p(a) = TRUE & (na = a+1) &
  \forall int x ; \forall int y ;
    (( p(x) = TRUE & (y=x+1) ) -> (n(y) = TRUE))
}
```

```
  \[{  ;   }\]

{ n(na) = TRUE }
}
```

Observe that in this example we are giving to `KeY-Hoare` an specification $\{P\}$ $\{\}$ $\{Q\}$ with an **empty** program, so we are really asking to prove that:

$$P -> Q$$

is a valid formula. This a way of using directly the capabilities of `KeY-Hoare` for FOL theorem proving for proving that a FOL formula is valid.

But keep in mind that no matter how big is any axiomatization of integer theory (written in First Order Logic) we use, Kurt Gödel showed that there will be always theorems that cannot be proven true. However, for practical purposes, current existing tools for formal program verification use inference systems that even if they are incomplete, for real programs almost any property that is valid in the program has a proof in the proof system. For example, the system `KeY` for verification of Java programs, uses a proof system that is able to prove many complex properties of programs with recursive data structures, and its set of inference rules is very similar to the one of its *small brother*, `KeY-Hoare`, although it is extended for being able to prove many specific situations that arise in Java programs, like for example the existence of execution paths that can produce exceptions.

References

[1] Kurt Godel. Uber formal unentscheidbare satze der principia mathematica und verwandter systeme, i. *Monatshefte fur Mathematik und Physik*, 38:173–198, 1931.

[2] Reiner Hähnle and Richard Bubel. A hoare-style calculus with explicit state updatese. `https://formal.kastel.kit.edu/key/download/hoare/students.pdf`, 2023.