Chapter 2

Modern Program Logics: Blocking Semantics

The purpose of this chapter is to present an alternative to the classical presentation given in the previous chapter. The main ideas are as follows.

- The programming language is a language of expressions in the style of ML, instead of a simple language of instructions à la PASCAL. The language is typed.
- The specifications are now made of various kinds of annotations in the programming language: loop invariants but also assertions, loop variants, etc.
- An alternative proof of soundness is presented, based on a proof of preservation of weakest precondition by reduction, and a property of progress, using the same guidelines as the classical proof of type soundness.
- The notion of labels is introduced, in order to specify properties relating different states of execution.
- This alternative presentation will allow a simpler treatment of function calls, which will be introduced in the next chapter.

2.1 An ML-like Programming Language

2.1.1 Syntax

In the language considered now, as in ML, we make no difference between "expressions" and "statements". The expressions considered in the previous chapter, which were *pure* (i.e. they have no side-effects), are now called *terms*. Our language contains a few basic data types: integers, booleans, reals, and also the unit type to denote the type of expressions that return no value.

In other words, we basically consider a purely functional language (ML-like), in which we add only a very specific kind of side-effect: modification of mutable variables. This restricted way of modifying program states is essential to make verification of programs easier.

This language is close to the one used in the Why3 system [1] (`http://why3.lri.fr/`). In fact, it is a subset of it.

Base Data Types, Operators, Terms, and Formulas

The grammar of terms is

$$
\begin{array}{rcll}
t & ::= & val & \text{(values, i.e. constants)} \\
 & | & v & \text{(logic variables)} \\
 & | & x & \text{(program variables)} \\
 & | & t\ op\ t & \text{(binary operations)} \\
 & | & \texttt{if}\ t\ \texttt{then}\ t\ \texttt{else}\ t & \text{(if-expression)} \\
 & | & \texttt{let}\ v = t\ \texttt{in}\ t & \text{(local binding)}
\end{array}
$$

- The unit type is denoted as `unit`, and there is only one constant of this type, denoted $()$.
- The type of Booleans is denoted as `bool`, with the two constants $True$ and $False$. The Boolean operators are denoted as and, or, not.
- The type of integers is denoted as `int`, with operators $+$, $-$, and $*$.
- The type of reals is denoted as `real`, with operators $+$, $-$, and $*$.
- We have the usual comparison operators, both on integers and on reals, returning a Boolean.

The syntax of terms is also extended with the "if-expression" which is written as `if` b `then` t_1 `else` t_2, and the local binding of logic variables.

Remark that we still have no division operator, as in the previous chapter. So we still do not have any kind of "run-time error". Also, notice that in the theorem provers available in practice (e.g. those available within Why3: Alt-Ergo, CVC3, Z3, etc.), these types are typically supported. They may also support if-expressions and let bindings. Otherwise Why3 transforms terms and formulas when needed (e.g. transformation of if-expressions and/or let-expressions into equivalent formulas [1]).

Example 2.1.1 *Here are a few toy examples of terms that one can write.*

```
(* approximated cosine *)
let cos_x =
  let y = x*x in
  1.0 - 0.5 * y + 0.04166666 * y * y
in
...

(* numbers of solutions of ax2 + bx + c = 0 *)
let delta = b * b - 4.0 * a * c in
let n = if delta < 0.0 then 0 else if delta = 0.0 then 1 else 2
in
...
```

The syntax of formulas is the same as in previous chapter, with the addition of local binding, where a variable is bound to a term in a formula. Also, the quantifications are typed.

$$
\begin{array}{rcll}
p & ::= & t & \text{(Boolean term)} \\
 & | & p \wedge p \mid p \vee p \mid \neg p \mid p \Rightarrow p & \text{(connectives)} \\
 & | & \forall v : \tau,\ p \mid \exists v : \tau,\ p & \text{(quantification)} \\
 & | & \texttt{let}\ v = t\ \texttt{in}\ p & \text{(local binding)}
\end{array}
$$

Expressions

Our programming language is made of expressions that may have side-effects. The grammar is

$$\begin{array}{rcll} e & ::= & t & \text{(pure term)} \\ & | & e\ op\ e & \text{(binary operation)} \\ & | & x := e & \text{(assignment)} \\ & | & \texttt{let}\ v = e\ \texttt{in}\ e & \text{(local binding)} \\ & | & \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e & \text{(conditional)} \\ & | & \texttt{while}\ e\ \texttt{do}\ e & \text{(loop)} \end{array}$$

The former notion of statement of the previous chapter now corresponds to expressions of type `unit`, in particular the former statement `skip` is now identified with $()$. Similarly, the former notion of sequence is generalized into the local binding: the *notation* $e_1; e_2$ is syntactic sugar for $\texttt{let}\ v = e_1\ \texttt{in}\ e_2$ when e_1 has type `unit` and v is not used in e_2.

Example 2.1.2 *Here are a few toy examples of expressions in our language*

```
z := if x ≥ y then x else y

let v = r in (r := v + 42; v)

while (x := x - 1; x > 0) do ()
(* x should be 0 at exit *)

while (let v = x in x := x - 1; v > 0) do ()
(* x should be -1 at exit *)
```

2.1.2 Typing

Since our language has expressions of different types, we need to introduce a notion of typing. Proof techniques introduced later will always assume that the given program is well typed.

The grammar of types is

$$\tau \quad ::= \quad \texttt{int} \mid \texttt{real} \mid \texttt{bool} \mid \texttt{unit}$$

A *typing judgment* is a notation of the form $\Gamma \vdash t : \tau$ meaning that t has type τ in *environment* Γ. For our language, we decide that the environment Γ maps each identifier to its type. It also tells whether it corresponds to a mutable variable. We denote

- either $v : \tau \in \Gamma$ when v is a logic variable (immutable) of type τ,
- or $x : \texttt{ref}\ \tau \in \Gamma$ when x is a program variable (mutable) of type τ.

It is important to notice that a reference is not a value in our language. In particular, there is nothing like a reference on a reference. This case will be treated in the chapter on *aliasing*. The typing rules for terms are as follows.

The rules for constants (i.e. values) simply explicit their respective types:

$$\frac{}{\Gamma \vdash n : \texttt{int}} \qquad \frac{}{\Gamma \vdash r : \texttt{real}} \qquad \frac{}{\Gamma \vdash () : \texttt{unit}}$$

$$\frac{}{\Gamma \vdash \mathit{True} : \texttt{bool}} \qquad \frac{}{\Gamma \vdash \mathit{False} : \texttt{bool}}$$

The rules for variables are:

$$\frac{v : \tau \in \Gamma}{\Gamma \vdash v : \tau} \qquad \frac{x : \texttt{ref}\ \tau \in \Gamma}{\Gamma \vdash x : \tau}$$

The rule for the conditional term imposes that the condition is Boolean and the two branches have the same type:

$$\frac{\Gamma \vdash t_1 : \texttt{bool} \qquad \Gamma \vdash t_2 : \tau \qquad \Gamma \vdash t_3 : \tau}{\Gamma \vdash \texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3 : \tau}$$

The rule for local binding is

$$\frac{\Gamma \vdash t_1 : \tau_1 \qquad \{v : \tau_1\} \cdot \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash \texttt{let}\ v = t_1\ \texttt{in}\ t_2 : \tau_2}$$

Again, one should notice that the type of a term is always a base type, never a reference.

The typing for formulas is expressed using judgment of the form $\Gamma \vdash f$ that just says that f is a well-typed formula in environment Γ.

$$\frac{\Gamma \vdash t : \texttt{bool}}{\Gamma \vdash t}$$

$$\frac{\Gamma \vdash f_1 \qquad \Gamma \vdash f_2}{\Gamma \vdash f_1 \vee f_2} \qquad \frac{\Gamma \vdash f_1 \qquad \Gamma \vdash f_2}{\Gamma \vdash f_1 \wedge f_2}$$

$$\frac{\Gamma \vdash f_1 \qquad \Gamma \vdash f_2}{\Gamma \vdash f_1 \Rightarrow f_2} \qquad \frac{\Gamma \vdash f}{\Gamma \vdash \neg f}$$

$$\frac{\{v : \tau\} \cdot \Gamma \vdash f}{\Gamma \vdash \forall v : \tau, f} \qquad \frac{\{v : \tau\} \cdot \Gamma \vdash f}{\Gamma \vdash \exists v : \tau, f}$$

$$\frac{\Gamma \vdash t : \tau \qquad \{v : \tau\} \cdot \Gamma \vdash f}{\Gamma \vdash \texttt{let}\ v = t\ \texttt{in}\ f}$$

Typing expressions

The typing rules for expressions are as follows.

The rule for assignment ensures that the variable assigned is mutable, and that the expression has the same type as the variables:

$$\frac{x : \texttt{ref}\ \tau \in \Gamma \qquad \Gamma \vdash e : \tau}{\Gamma \vdash x := e : \texttt{unit}}$$

The rules for local binding and for conditional are similar to those for terms:

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \{v : \tau_1\} \cdot \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let}\ v = e_1\ \texttt{in}\ e_2 : \tau_2}$$

$$\frac{\Gamma \vdash c : \texttt{bool} \qquad \Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{if}\ c\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \tau}$$

Finally, the rule for while loop ensures that the condition is Boolean and the body does not return a value.

$$\frac{\Gamma \vdash c : \texttt{bool} \qquad \Gamma \vdash e : \texttt{unit}}{\Gamma \vdash \texttt{while}\ c\ \texttt{do}\ e : \texttt{unit}}$$

2.1.3 Operational Semantics

From now, we only consider well-typed programs. In particular, we define the operational semantics only on well-typed expressions. Moreover, the typing process checks that any variable occurring in an expression or a term is appropriately declared in the scope. Indeed, the resolution of variables scope allows us to assume from now that variable names are unique, so that we can ignore any risk of capture of variables names in the semantics. Moreover, we can even assume that any variable is decorated with its type.

To take into account local variables, we augment program states with a second component, for these local variables:

- Σ: maps program variables to values
- Π: maps (local) logic variables to values

Semantics of Terms and Formulas

The semantics is given by the following function which is *a priori* partial.

$$\begin{array}{rcll} [\![val]\!]_{\Sigma,\Pi} & = & val & \text{(values)} \\ [\![x]\!]_{\Sigma,\Pi} & = & \Sigma(x) & \text{if } x : \texttt{ref}\ \tau \\ [\![v]\!]_{\Sigma,\Pi} & = & \Pi(v) & \text{if } v : \tau \\ [\![t_1\ op\ t_2]\!]_{\Sigma,\Pi} & = & [\![t_1]\!]_{\Sigma,\Pi}\ [\![op]\!]\ [\![t_2]\!]_{\Sigma,\Pi} & \\ [\![\texttt{let}\ v = t_1\ \texttt{in}\ t_2]\!]_{\Sigma,\Pi} & = & [\![t_2]\!]_{\Sigma,(\{v=[\![t_1]\!]_{\Sigma,\Pi}\}\cdot\Pi)} & \end{array}$$

Notice how we use the "type decoration" of a variable to decide if we access to Σ of Π.

This semantics is *a priori* only a partial function because it is guaranteed to be defined only on well-typed terms. Indeed, our logic language satisfies the following standard property of *type soundness* of purely functional languages.

Theorem 2.1.3 (Type soundness) *Every well-typed terms and well-typed formulas have a semantics.*

Proof. Straightforward using an induction on the derivation tree of well-typing.

Operational Semantics of Expressions

The fact that our language is based on expressions introduces two novelties. First we need *context rules* to evaluate sub-expressions. Second we need to precise the order of evaluation; we choose here to evaluate from left to right.

The relation for one-step execution has now the form

$$\Sigma, \Pi, e \rightsquigarrow \Sigma', \Pi', e'$$

Naturally, values do not reduce. The following rules define how other expressions can be reduced.

There are two rules for assignment:

$$\frac{\Sigma, \Pi, e \rightsquigarrow \Sigma', \Pi', e'}{\Sigma, \Pi, x := e \rightsquigarrow \Sigma', \Pi', e'}$$

$$\frac{}{\Sigma, \Pi, x := val \rightsquigarrow \Sigma[x \leftarrow val], \Pi, ()}$$

The first rule is a context rule: it tells that one should evaluate the right part to a value, before performing the actual assignment using the second rule.

The rules for local binding are:

$$\frac{\Sigma, \Pi, e_1 \rightsquigarrow \Sigma', \Pi', e_1'}{\Sigma, \Pi, \texttt{let}\ v = e_1\ \texttt{in}\ e_2 \rightsquigarrow \Sigma', \Pi', \texttt{let}\ v = e_1'\ \texttt{in}\ e_2}$$

$$\frac{}{\Sigma, \Pi, \texttt{let}\ v = val\ \texttt{in}\ e \rightsquigarrow \Sigma, \Pi[v \leftarrow val], e}$$

The binary operations evaluate as follows.

$$\frac{\Sigma, \Pi, e_1 \rightsquigarrow \Sigma', \Pi', e_1'}{\Sigma, \Pi, e_1 + e_2 \rightsquigarrow \Sigma', \Pi', e_1' + e_2}$$

$$\frac{\Sigma, \Pi, e_2 \rightsquigarrow \Sigma', \Pi', e_2'}{\Sigma, \Pi, val_1 + e_2 \rightsquigarrow \Sigma', \Pi', val_1 + e_2'}$$

$$\frac{val = val_1 + val_2}{\Sigma, \Pi, val_1 + val_2 \rightsquigarrow \Sigma, \Pi, val}$$

The first two rules, again context rules, explicit the order of evaluation, from left to right. Notice that for context rules, we could indifferently use Π or Π' in the right part of the reduction, because we do not care about the value of local variables introduced in sub-expressions, which are not used afterward (however, the situation will be different when we will introduce local mutable variables in the next chapter).

The rules for the conditional expression are

$$\frac{\Sigma, \Pi, c \rightsquigarrow \Sigma', \Pi', c'}{\Sigma, \Pi, \texttt{if}\ c\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \rightsquigarrow \Sigma', \Pi', \texttt{if}\ c'\ \texttt{then}\ e_1\ \texttt{else}\ e_2}$$

$$\frac{}{\Sigma, \Pi, \texttt{if}\ \mathit{True}\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \rightsquigarrow \Sigma, \Pi, e_1}$$

$$\frac{}{\Sigma, \Pi, \texttt{if}\ \mathit{False}\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \rightsquigarrow \Sigma, \Pi, e_2}$$

Finally, there is a unique rule for the while loop:

$$\frac{}{\begin{array}{l}\Sigma, \Pi, \texttt{while}\ c\ \texttt{do}\ e \rightsquigarrow \\ \qquad \Sigma, \Pi, \texttt{if}\ c\ \texttt{then}\ (e; \texttt{while}\ c\ \texttt{do}\ e)\ \texttt{else}\ ()\end{array}}$$

Context Rules versus Let Binding

An important and useful remark is to notice that most of the context rules can be avoided, by introducing extra local bindings. For example, the binary operations can be evaluated as

$$\frac{v_1, v_2 \text{ fresh}}{\Sigma, \Pi, e_1 + e_2 \rightsquigarrow \Sigma, \Pi, \texttt{let}\ v_1 = e_1\ \texttt{in}\ \texttt{let}\ v_2 = e_2\ \texttt{in}\ v_1 + v_2}$$

Using such a transformation, only the context rule for the local binding is really needed, so we will not introduce any new context rules.

Type Soundness Property

Theorem 2.1.4 (Type soundness) *Every well-typed expression evaluates to a value or executes infinitely.*

Proof. The classical method for performing such a proof is made in two parts:

1. One proves that the type of an expression e is preserved by reduction; this is typically done using an induction on the structure of e.

2. One shows that well-typed expressions that are not values can *progress*, i.e. there is at least one rule for one-step reduction that can be applied.

A straightforward induction on steps of execution finishes the proof.

2.2 Blocking Operational Semantics

We are now going to slightly modify our language in the same spirit as when we introduced explicit loop invariants in the previous chapter. We introduce annotations in the text of programs, that must hold during execution.

The syntax of expressions is thus modified:

- a new kind of expression is introduced: an *assertion*;
- explicit loop invariants are added to while loops.

The notion of assertion will be a general way of dealing with *run-time errors*.

The grammar for expressions is thus modified with these rules:

$$\begin{array}{rcll} e & ::= & \texttt{assert}\ p & \text{(assertion)} \\ & | & \texttt{while}\ e\ \texttt{invariant}\ I\ \texttt{do}\ e & \text{(annotated loop)} \end{array}$$

Example 2.2.1 *We can add assertions and loop invariants in the toy examples given in Example 2.1.1, as follows.*

```
z := if x ≥ y then x else y;
assert (z ≥ x ∧ z ≥ y)

while (x := x - 1; x > 0) invariant x ≥ 0 do ();
assert (x = 0)

while (let v = x in x := x - 1; v > 0) invariant x ≥ -1 do ();
assert (x < 0)
```

Typing

Our new constructions are typed as follows.

$$\frac{\Gamma \vdash P}{\Gamma \vdash \texttt{assert}\ P : \texttt{unit}}$$

$$\frac{\Gamma \vdash c : \texttt{bool} \qquad \Gamma \vdash I \qquad \Gamma \vdash e : \texttt{unit}}{\Gamma \vdash \texttt{while}\ c\ \texttt{invariant}\ I\ \texttt{do}\ e : \texttt{unit}}$$

2.2.1 Blocking Semantics and Soundness

We now modify the operational semantics to make explicit the fact that assertions and loop invariants should hold during execution. In other words, the execution will block if any such annotation is falsified [2, 4, 3].

$$\frac{[\![P]\!]_{\Sigma,\Pi}\ \text{holds}}{\Sigma, \Pi, \texttt{assert}\ P \rightsquigarrow \Sigma, \Pi, ()}$$

$$\frac{[\![I]\!]_{\Sigma,\Pi}\ \text{holds}}{\begin{array}{l}\Sigma, \Pi, \texttt{while}\ c\ \texttt{invariant}\ I\ \texttt{do}\ e \rightsquigarrow \\ \qquad \Sigma, \Pi, \texttt{if}\ c\ \texttt{then}\ (e; \texttt{while}\ c\ \texttt{invariant}\ I\ \texttt{do}\ e)\ \texttt{else}\ ()\end{array}}$$

We emphasize again the main point of this definition: execution blocks as soon as an invalid annotation is met.

Soundness of a program

We first define the notion of safe execution.

Definition 2.2.2 *We say that the execution of an expression in a given state is* safe *if it does not block: either it terminates on a value or it runs infinitely.*

In other words, safe execution is the converse of blocking. An equivalent definition of safety is thus: whenever

$$\Sigma, \Pi, e \rightsquigarrow \Sigma_1, \Pi_1, e_1 \rightsquigarrow \Sigma_2, \Pi_2, e_2 \rightsquigarrow \cdots \rightsquigarrow \Sigma_n, \Pi_n, e_n$$

and e_n does not reduce anymore, then e_n is a value.

The definition of the *validity* of some triple is modified with respect to the previous chapter, by using the notion of safety above.

Definition 2.2.3 *We say that a triple $\{P\}e\{Q\}$ is valid if for any state Σ, Π satisfying P, e* executes safely *in Σ, Π, and if it terminates, the final state satisfies Q*

Notice that, unlike in the previous chapter, the validity of a triple is meaningful even in the case when the expression does not terminate. Indeed, it still ensures that assertions and loop invariants met during execution are valid.

Example 2.2.4 *The triple*

```
{ x = 0 }
while true invariant x ≥ 0
   do (x := x + 2; assert x > 0);
{ false }
```

is valid: on one hand, the loop invariant and the assertion hold at each iteration, on the other hand the post-condition is never reached.

The result keyword

To complete the notion of triple, we introduce a way of talking about the result of an expression in a post-condition. This is done by adding a new specific keyword result in the logic. It can be used only in post-conditions, and it denotes the resulting value of the expression executed.

Example 2.2.5 *In the following annotated code, we specify that the result is greater or equal to both x and y:*

```
{ true }
if x ≥ y then x else y
{ result ≥ x ∧ result ≥ y }
```

The following is a modified version of the ISQRT example of previous chapter, where the square root is returned as a result.

```
{ x ≥ 0 }
c := 0; sum := 1;
while sum ≤ x invariant c ≥ 0 ∧ c*c ≤ x ∧ sum = (c+1)*(c+1)
  do (c := c + 1; sum := sum + 2 * c + 1);
c
{ result ≥ 0 ∧ result * result ≤ x < (result+1)*(result+1) }
```

2.3 Weakest Preconditions Revisited

We now construct a new variant of the weakest precondition calculus $\mathrm{WP}(e, Q)$. We expect that this calculus allow us to prove safety of execution in the sense defined above, that is, in any state satisfying $\mathrm{WP}(e, Q)$, e is guaranteed to execute safely. Remark that this statement may seem strange at first since Q is not needed in the conclusion of it. In other words, stating this for $Q = \mathit{true}$ would be enough to ensure safety. However, we will see that we need to state this property for an arbitrary Q to prove soundness, by induction.

The rules for this new calculus are very similar to those of the previous chapter, but adapted to a language of expressions.

Pure terms:

$$\mathrm{WP}(t, Q) = Q[\mathsf{result} \leftarrow t]$$

Local binding:

$$\mathrm{WP}(\texttt{let}\ x = e_1\ \texttt{in}\ e_2, Q) = \mathrm{WP}(e_1, \mathrm{WP}(e_2, Q)[x \leftarrow \mathsf{result}])$$

Assignment:

$$\mathrm{WP}(x := e, Q) = \mathrm{WP}(e, Q[\mathsf{result} \leftarrow (); x \leftarrow \mathsf{result}])$$

Using our remark about the introduction of extra local bindings, some alternative rules for assignment can be given as:

$$\begin{array}{rcl} \mathrm{WP}(x := e, Q) & = & \mathrm{WP}(\texttt{let}\ v = e\ \texttt{in}\ x := v, Q) \\ \mathrm{WP}(x := t, Q) & = & Q[\mathsf{result} \leftarrow (); x \leftarrow t]) \end{array}$$

In particular, the second rule can be used as soon as the expression on the right of the assignment has no side effect.

Conditional:

$$\begin{array}{l} \mathrm{WP}(\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3, Q) = \\ \quad \mathrm{WP}(e_1, \texttt{if}\ \mathsf{result}\ \texttt{then}\ \mathrm{WP}(e_2, Q)\ \texttt{else}\ \mathrm{WP}(e_3, Q)) \end{array}$$

Assertion:

$$\begin{array}{rcl} \mathrm{WP}(\texttt{assert}\ P, Q) & = & P \wedge Q \\ & = & P \wedge (P \Rightarrow Q) \end{array}$$

That second version is useful in practice, since it puts P as an hypothesis for proving Q.

While loop:

$$\begin{array}{l} \mathrm{WP}(\texttt{while}\ c\ \texttt{invariant}\ I\ \texttt{do}\ e, Q) = \\ \quad I \wedge \\ \quad \forall \vec{v}, (I \Rightarrow \mathrm{WP}(c, \texttt{if}\ \mathsf{result}\ \texttt{then}\ \mathrm{WP}(e, I)\ \texttt{else}\ Q))[w_i \leftarrow v_i] \end{array}$$

where $w_1, \ldots, w_k$ is the set of assigned variables in expressions c and e and $v_1, \ldots, v_k$ are fresh logic variables

2.3.1 General Properties of WP

Lemma 2.3.1 (Monotonicity) *If* $\models P \Rightarrow Q$ *then* $\models \mathrm{WP}(e, P) \Rightarrow \mathrm{WP}(e, Q)$

First remark that we must quantify on *all states* on the hypothesis of this lemma to make it true. There are simple counter-examples otherwise, e.g. in a state Σ such that $\Sigma(x) = 42$, the implication $\mathit{true} \Rightarrow x = 42$ holds, but $\mathrm{WP}(x := 7; \mathit{true}) \equiv \mathit{true}$ does not imply $\mathrm{WP}(x := 7; x = 42) \equiv (7 = 42)$.

Proof. This is done by structural induction on e.

In all cases, we must prove $\models \mathrm{WP}(e, P) \Rightarrow \mathrm{WP}(e, Q)$, that is, for any Σ, Π such that $\Sigma, \Pi \models \mathrm{WP}(e, P)$, we must show that $\Sigma, \Pi \models \mathrm{WP}(e, Q)$.

Case of assignment: we know that $\mathrm{WP}(x := t, P) = P[x \leftarrow t]$ holds in Σ, Π, that is, P holds in $\Sigma[x \leftarrow t], \Pi$, but then Q holds in the same state, and thus $\mathrm{WP}(x := t, Q) = Q[x \leftarrow t]$ holds in Σ, Π.

Case of local binding: we have $\mathrm{WP}(\texttt{let}\ v = e_1\ \texttt{in}\ e_2, Q) = \mathrm{WP}(e_1, \mathrm{WP}(e_2, Q)[v \leftarrow \mathsf{result}])$. By induction hypothesis on e_2, we know that $\mathrm{WP}(e_2, P) \Rightarrow \mathrm{WP}(e_2, Q)$ holds in any state, in particular if we substitute v by result, that is $\mathrm{WP}(e_2, P)[v \leftarrow \mathsf{result}] \Rightarrow \mathrm{WP}(e_2, Q)[v \leftarrow \mathsf{result}]$ holds in any state, and we conclude again by induction hypothesis, this time on e_1.

Case of the while loop: we assume that

$$\begin{array}{l}\mathrm{WP}(\texttt{while}\ c\ \texttt{invariant}\ I\ \texttt{do}\ e, P) = \\ \quad I \wedge \\ \quad \forall \vec{v}, (I \Rightarrow \mathrm{WP}(c, \texttt{if}\ \mathsf{result}\ \texttt{then}\ \mathrm{WP}(e, I)\ \texttt{else}\ P))[w_i \leftarrow v_i]\end{array}$$

holds in some state Σ, Π, and we want to show that the same formula but with Q instead of P holds in the same state. This reduces to show

$$\Sigma, \Pi \models \forall \vec{v}, (I \Rightarrow \mathrm{WP}(c, \texttt{if}\ \mathsf{result}\ \texttt{then}\ \mathrm{WP}(e, I)\ \texttt{else}\ Q))[w_i \leftarrow v_i]$$

This amounts to showing that $\forall \vec{v}, \ldots$ holds in any state Σ', Π where in Σ' and Σ coincides on all variables except $w_1, \ldots, w_k$. We assumed that $\models P \Rightarrow Q$ thus

$$\models (\texttt{if}\ \mathsf{result}\ \texttt{then}\ \mathrm{WP}(e, I)\ \texttt{else}\ P) \Rightarrow (\texttt{if}\ \mathsf{result}\ \texttt{then}\ \mathrm{WP}(e, I)\ \texttt{else}\ Q)$$

By induction hypothesis on c, we have

$$\models \mathrm{WP}(c, (\texttt{if}\ \mathsf{result}\ \texttt{then}\ \mathrm{WP}(e, I)\ \texttt{else}\ P)) \Rightarrow \mathrm{WP}(c, (\texttt{if}\ \mathsf{result}\ \texttt{then}\ \mathrm{WP}(e, I)\ \texttt{else}\ Q))$$

since this holds in any state, it holds in particular in Σ', Π.

Other cases are straightforward.

Lemma 2.3.2 (Conjunction Property) *If* $\Sigma, \Pi \models \mathrm{WP}(e, P)$ *and* $\Sigma, \Pi \models \mathrm{WP}(e, Q)$ *then* $\Sigma, \Pi \models \mathrm{WP}(e, P \wedge Q)$

Proof. This is done by structural induction on e.

Case of the local binding: we have $\mathrm{WP}(\texttt{let}\ v = e_1\ \texttt{in}\ e_2, P \wedge Q) = \mathrm{WP}(e_1, \mathrm{WP}(e_2, P \wedge Q)[v \leftarrow \mathsf{result}])$

By induction hypothesis on e_2 we have

$$\models \mathrm{WP}(e_2, P) \wedge \mathrm{WP}(e_2, Q) \Rightarrow \mathrm{WP}(e_2, P \wedge Q)$$

Thus

$$\models \mathrm{WP}(e_2, P)[v \leftarrow \mathsf{result}] \wedge \mathrm{WP}(e_2, Q)[v \leftarrow \mathsf{result}] \Rightarrow \mathrm{WP}(e_2, P \wedge Q)[v \leftarrow \mathsf{result}]$$

By the monotonicity lemma, we get

$$\models \mathrm{WP}(e_1, \mathrm{WP}(e_2, P)[v \leftarrow \mathsf{result}] \wedge \mathrm{WP}(e_2, Q)[v \leftarrow \mathsf{result}])) \Rightarrow \mathrm{WP}(e_1, \mathrm{WP}(e_2, P \wedge Q)[v \leftarrow \mathsf{result}])$$

We finish the proof by the induction hypothesis on e_1, in the left part of this implication.

Other cases are straightforward.

Lemma 2.3.3 (Generalized Monotonicity) *If $w_1, \ldots, w_k$ is the set of assigned variables in expression e and $v_1, \ldots, v_k$ are fresh logic variables, and*

$$\Sigma, \Pi \models \forall \vec{v}, (P \Rightarrow Q)[w_i \leftarrow v_i]$$

and

$$\Sigma, \Pi \models \mathrm{WP}(e, P)$$

then

$$\Sigma, \Pi \models \mathrm{WP}(e, Q)$$

Proof.

We notice first that the hypothesis

$$\Sigma, \Pi \models \forall \vec{v}, (P \Rightarrow Q)[w_i \leftarrow v_i]$$

is equivalent to saying that

$$\Sigma', \Pi \models P \Rightarrow Q$$

for any Σ', Π such that Σ' and Σ coincide on each variable that is not modified by e.

The proof is again done by structural induction on e.

Case of assignment: we know that $\mathrm{WP}(x := t, P) = P[x \leftarrow t]$ holds in Σ, Π, that is, P holds in $\Sigma[x \leftarrow t], \Pi$. But this state coincides with Σ on variables not modified, so Q holds in the same state, and thus $\mathrm{WP}(x := t, Q) = Q[x \leftarrow t]$ holds in Σ, Π

Other cases are similar to the proof of the monotonicity lemma.

2.3.2 Soundness of WP

Lemma 2.3.4 (Preservation by Reduction) *If $\Sigma, \Pi \models \mathrm{WP}(e, Q)$ and $\Sigma, \Pi, e \rightsquigarrow \Sigma', \Pi', e'$ then $\Sigma', \Pi' \models \mathrm{WP}(e', Q)$.*

Proof. This is done by predicate induction of $\rightsquigarrow$: we consider all the rules in turn, and prove the lemma for the reduction in the conclusion of the rule, assuming that the lemma holds for the reductions in hypothesis of the same rule.

Case of the rule for one-step reduction of a while loop:

$$\frac{[\![I]\!]_{\Sigma,\Pi} \text{ holds}}{\Sigma, \Pi, \texttt{while}\ c\ \texttt{invariant}\ I\ \texttt{do}\ e \rightsquigarrow \Sigma, \Pi, \texttt{if}\ c\ \texttt{then}\ (e; \texttt{while}\ c\ \texttt{invariant}\ I\ \texttt{do}\ e)\ \texttt{else}\ ()}$$

The WP of the result of this reduction is (assuming for simplicity that the condition c is pure)

$$\begin{array}{l}\mathrm{WP}(\texttt{if}\ c\ \texttt{then}\ e; \texttt{while}\ c\ \texttt{invariant}\ I\ \texttt{do}\ e\ \texttt{else}\ (), Q) = \\ \texttt{if}\ c\ \texttt{then}\ \mathrm{WP}(e; \texttt{while}\ c\ \texttt{invariant}\ I\ \texttt{do}\ e, Q)\ \texttt{else}\ \mathrm{WP}((), Q) = \\ \texttt{if}\ c\ \texttt{then}\ \mathrm{WP}(e, \mathrm{WP}(\texttt{while}\ c\ \texttt{invariant}\ I\ \texttt{do}\ e, Q))\ \texttt{else}\ Q\end{array}$$

We need to show that this formula is valid in state Σ, Π. If the condition c is false in this state, then we must show that Q holds. But since the WP of that loop is true initially, then $I \Rightarrow Q$ holds, and since I holds we are done. If the condition c is true, then we need to prove the validity of

$$\begin{array}{l}\mathrm{WP}(e, \mathrm{WP}(\texttt{while}\ c\ \texttt{invariant}\ I\ \texttt{do}\ e, Q)) = \\ \mathrm{WP}(e, I \wedge \forall \vec{v} \ldots)\end{array}$$

Thanks to the conjunction property, it suffices to prove separately that $\mathrm{WP}(e, I)$ and $\mathrm{WP}(e, \forall \vec{v}, (I \Rightarrow \mathrm{WP}(c, \texttt{if result then WP}(e, I) \texttt{ else } P))[w_i \leftarrow v_i])$ hold. The first formula holds thanks to the hypothesis that the initial WP holds. Since the WP of the loop is assumed to hold initially, the formula

$$\forall \vec{v}, (I \Rightarrow \mathrm{WP}(c, \texttt{if result then WP}(e, I) \texttt{ else } P))[w_i \leftarrow v_i]$$

holds, and since $\mathrm{WP}(e, I)$ holds, the generalized monotonicity property gives exactly the formula we need to prove.

The other cases are straightforward.

Lemma 2.3.5 (Progress) *If $\Sigma, \Pi \models \mathrm{WP}(e, Q)$ and e is not a value then there exists Σ', Π', e' such that $\Sigma, \Pi, e \rightsquigarrow \Sigma', \Pi', e'$.*

Proof. This is done by structural induction on e.

For all constructions of expressions that are not values, there is a reduction rule that applies, except maybe the assertion and the loop, since the blocking semantics requires us to prove that the corresponding annotation (resp. the assertion or the loop invariant) holds. In both cases, the hypothesis that the WP holds implies trivially that this annotation holds.

Theorem 2.3.6 (Soundness) *If $\Sigma, \Pi \models \mathrm{WP}(e, Q)$ then e executes safely in Σ, Π. Moreover, if e terminates with value v, $Q[\mathsf{result} \leftarrow v]$ holds in the final state.*

Proof. We assume that $\Sigma, \Pi \models \mathrm{WP}(e, Q)$. If the execution of e is infinite, it is neither blocking nor terminating, so the theorem trivially holds. So we assume the contrary:

$$\Sigma, \Pi, e \rightsquigarrow \Sigma_1, \Pi_1, e_1 \rightsquigarrow \Sigma_2, \Pi_2, e_2 \rightsquigarrow \cdots \rightsquigarrow \Sigma_n, \Pi_n, e_n$$

where e_n does not reduce. We have to prove that e_n is a value and that $\Sigma_n, \Pi_n \models Q[\mathsf{result} \leftarrow e_n]$ holds. This is done by induction on n

When $n = 0$, we have $\Sigma_n = \Sigma$, $\Pi_n = \Pi$ and $e_n = e$, thus $\Sigma_n, \Pi_n \models \mathrm{WP}(e_n, Q)$, hence the progress lemma tells us that e_n must be a value. Moreover, $\mathrm{WP}(e_n, Q)$ is then $Q[\mathsf{result} \leftarrow e_n]$, which thus holds in Σ_n, Π_n.

When $n > 0$, we have $\Sigma_1, \Pi_1 \models \mathrm{WP}(e_1, Q)$ thanks to the preservation lemma. Since

$$\Sigma_1, \Pi_1, e_1 \rightsquigarrow \Sigma_2, \Pi_2, e_2 \rightsquigarrow \cdots \rightsquigarrow \Sigma_n, \Pi_n, e_n$$

is made of $n - 1$ steps, by induction we know that the property holds.

2.4 Programs with Labels

On top of the language defined previously, we add the notion of *labels*. Labels can be inserted in the code to give names to some program points. In the specifications, one can then refer to the value of a mutable variables as it was the program state associated to that label. Example are given below.

2.4.1 Syntax and Typing

We add in the syntax of *expressions* the following construction for declaring labels:

$$e \quad ::= \quad L : e \quad \text{(labeled expression)}$$

We then add in the syntax of *terms* the following construction to denote the variable at some label:

$$t \quad ::= \quad x@L \quad \text{(labeled variable access)}$$

Example 2.4.1 *In the toy code below*

```
L: x := x + 42;
assert { x > x@L }
```

the assertion means that the value of variable `x` *is now larger that the value it had when the label* `L` *was met, that is before the assignment.*

Implicit labels

To write specifications more easily, it is handy to assume some pre-existing labels:

- The label `Here`, available in every formula, denotes the current program point. In other words, writing `x` is equivalent to writing `x@Here`.
- The label `Old`, available in post-conditions of triples only, denotes the beginning of the corresponding expression. In other words, any triple $\{P\}e\{Q\}$ can be seen as $\{P\}Old : e\{Q\}$

Example 2.4.2 *Pre-incrementation:*

```
{ true }
let v = r in (r := v + 42; v)
{ r = r@Old + 42 ∧ result = r@Old }
```

Swapping two references:

```
{ true }
let tmp = x in x := y; y := tmp
{ x = y@Old ∧ y = x@Old }
```

Finally, we introduce some syntactic sugar: any term t can be labeled as $t@L$, meaning that label L is attached to any variable of t that does not have an explicit label yet. For example

$$(x + y@K + 2)@L + x$$

is the same as

$$x@L + y@K + 2 + x@\mathit{Here}$$

Typing

The new typing rules should ensures that only mutable variables can be accessed through a label, and labels must be declared before use. To achieve this, we need to augment the typing judgment with the set of already declared labels in the context, under the notations $\mathcal{L}, \Gamma \vdash t : \tau$. The new typing rules are

$$\frac{(\{L\} \cdot \mathcal{L}), \Gamma \vdash e : \tau}{L, \Gamma \vdash L : e : \tau}$$

$$\frac{L \in \mathcal{L} \qquad x : \texttt{ref}\ \tau \in \Gamma}{L, \Gamma \vdash x@L : \tau}$$

The other rules are essentially unchanged, the label environment $\mathcal{L}$ being just added recursively.

2.4.2 Labels: Operational Semantics

To define the operational semantics, we need to change the notion of program state, so that one can retrieve the value of a variable in a former state. This is achieved by considering that the program state is not only a map indexed by variables, but a collection of maps indexed by labels. Given such a collection, still denoted as Σ, the value of some variable x at some label L is now denoted $\Sigma(x, L)$.

The definition of the semantics of variables in terms is modifying accordingly:

$$\begin{array}{rcl} [\![x]\!]_{\Sigma,\Pi} & = & \Sigma(x, \mathit{Here}) \\ [\![x@L]\!]_{\Sigma,\Pi} & = & \Sigma(x, L) \end{array}$$

Notice again that x alone can be considered as just an abbreviation for $x@\mathit{Here}$.

For the definition of the operational semantics of expressions, we introduce a notation $\Sigma[L_1 \leftarrow L_2]$ to denote that all the values of variables at label L_2 are copied under the label L_1:

$$\Sigma[L_1 \leftarrow L_2] := \Sigma\{(x, L_1) \leftarrow \Sigma(x, L_2) \mid x \text{ any variable}\}$$

The rules for operational semantics of expressions are then modified as follows:

$$\begin{array}{rcl} \Sigma, \Pi, x := val & \rightsquigarrow & \Sigma\{(x, \mathit{Here}) \leftarrow val\}, \Pi, () \\ \Sigma, \Pi, L : e & \rightsquigarrow & \Sigma[L \leftarrow \mathit{Here}], \Pi, e \end{array}$$

2.4.3 New Rules for Weakest Preconditions

We pose now the following new rules for computing WP:

$$\begin{array}{rcl} \mathrm{WP}(x := t, Q) & = & Q[x@\mathit{Here} \leftarrow t] \\ \mathrm{WP}(L : e, Q) & = & \mathrm{WP}(e, Q)[L \leftarrow \mathit{Here}] \end{array}$$

Again $[L \leftarrow \mathit{Here}]$ is an abbreviation for the substitution of all $x@L$ by $x@\mathit{Here}$, for any variable x.

Theorem 2.4.3 (Soundness of WP**)** *Soundness (as stated in Theorem 2.3.6) still holds on our language extended with labels.*

Proof. As before, it amounts to prove the preservation of WP by reduction and the progress, for the new constructions. Each of these new cases is straightforward.

2.5 Proving Termination

As in Section 1.4 of previous chapter, we extend the syntax of annotations in order to prove termination of a program. With our simple language, it amounts to show that loops are never infinite, using *loop variants*.

2.5.1 Syntax

As in previous chapter, we extend the syntax of loop by allowing a variant to be given:

$$e \ ::= \ \texttt{while}\ e\ \texttt{invariant}\ I\ \texttt{variant}\ t, \prec\ \texttt{do}\ e$$

The loop variant t can be any term of some type τ, and $\prec$ can be any well-founded ordering on τ.

Example 2.5.1 *The following annotated program computes the sum of* `x` *and* `y` *and stores the result in* `x`*.*

```
{ y ≥ 0 }
L:
while y > 0
  invariant x + y = x@L + y@L
  variant y
  do x := x + 1; y := y - 1
{ x = x@Old + y@Old ∧ y = 0 }
```

The loop variant allows to prove its termination, with the standard ordering on integers defined in the previous chapter.

2.5.2 Operational Semantics

Following our idea of blocking semantics, we modify the rule for reducing a loop, by adding a check that the variant decreases. This is done naturally by introducing an extra assertion as follows:

$$\frac{[\![I]\!]_{\Sigma,\Pi} \text{ holds}}{\begin{array}{l}\Sigma, \Pi, \texttt{while}\ c\ \texttt{invariant}\ I\ \texttt{variant}\ t, \prec\ \texttt{do}\ e \rightsquigarrow \\ \quad \Sigma, \Pi, \texttt{if}\ c \\ \qquad \texttt{then}\ (e; \texttt{assert}\ t \prec [\![t]\!]_{\Sigma,\Pi}; \texttt{while}\ c\ \texttt{invariant}\ I\ \texttt{variant}\ t, \prec\ \texttt{do}\ e) \\ \qquad \texttt{else}\ ()\end{array}}$$

There is an equivalent rule that makes use of a label:

$$\frac{[\![I]\!]_{\Sigma,\Pi} \text{ holds}}{\begin{array}{l}\Sigma, \Pi, \texttt{while}\ c\ \texttt{invariant}\ I\ \texttt{variant}\ t, \prec\ \texttt{do}\ e \rightsquigarrow \\ \quad \Sigma, \Pi, L : \texttt{if}\ c \\ \qquad \texttt{then}\ (e; \texttt{assert}\ t \prec t@L; \texttt{while}\ c\ \texttt{invariant}\ I\ \texttt{variant}\ t, \prec\ \texttt{do}\ e) \\ \qquad \texttt{else}\ ()\end{array}}$$

Naturally, label L above is assumed to be *fresh*, that is, not already used.

2.5.3 Weakest Precondition

Notice that, unlike in the previous chapter, we do not make any distinction between liberal and strict weakest precondition. Indeed, one can specify individually for each loop whether termination should be proven, by annotating it with a variant or not.

The rule for computing the WP of a loop with a variant is quite naturally derived from those without a variant by adding the verification of the assertion as introduced in the operational semantics:

$$\begin{array}{l}\mathrm{WP}(\texttt{while}\ c\ \texttt{invariant}\ I\ \texttt{variant}\ t, \prec\ \texttt{do}\ e, Q) = \\ \quad I \wedge \forall \vec{v}, (I \rightarrow \mathrm{WP}(L : c, \texttt{if}\ result\ \texttt{then}\ \mathrm{WP}(e, I \wedge t \prec t@L)\ \texttt{else}\ Q))[w_i \leftarrow v_i]\end{array}$$

Again, the introduced label L must be fresh.

2.6 Local Mutable Variables

The last thing we add to our language in this chapter is the ability to have *local mutable variables*. The syntax of expressions is extended with

$$e ::= \texttt{let ref}\ id = e\ \texttt{in}\ e$$

Example 2.6.1 *Our running example ISQRT can be modified in order to make the* `sum` *and* `res` *variables local, as follows. The other variable* `x` *remains global.*

```
let ref res = 0 in
let ref sum = 1 in
while sum ≤ x do
  res := res + 1; sum := sum + 2 * res + 1
done;
res
```

The operational semantics $\Sigma, \Pi, e \rightsquigarrow \Sigma', \Pi', e'$ is just slight extended by saying that Π no longer contains just immutable variables, but also mutable ones. Like for Σ, Π is now also a collection of maps indexed by labels.

The extra rules are then

$$\frac{\Sigma, \Pi, e_1 \rightsquigarrow \Sigma', \Pi', e_1'}{\Sigma, \Pi, \texttt{let ref}\ x = e_1\ \texttt{in}\ e_2 \rightsquigarrow \texttt{let ref}\ x = e_1'\ \texttt{in}\ e_2}$$

$$\frac{}{\Sigma, \Pi, \texttt{let ref}\ x = v\ \texttt{in}\ e \rightsquigarrow \Sigma, \Pi\{(x, \mathit{Here}) \mapsto v\}, e}$$

$$\frac{x \text{ local variable}}{\Sigma, \Pi, x := v \rightsquigarrow \Sigma, \Pi\{(x, \mathit{Here}) \mapsto v\}, e}$$

The new rule for computing WP is

$$\mathrm{WP}(\texttt{let ref}\ x = e_1\ \texttt{in}\ e_2, Q) = \mathrm{WP}(e_1, \mathrm{WP}(e_2, Q)[x \leftarrow \mathsf{result}])$$

which is indeed exactly the same as the one for immutable variables.

2.7 Exercises

Exercise 2.7.1 *Compute the following weakest preconditions by hand, and check that they are valid.*

$$WP(\texttt{let}\ v = x\ \texttt{in}\ (x := x + 1; v), x > result)$$

$$\mathrm{WP}(L : x := x + 42, x@\mathit{Here} > x@L)$$

Exercise 2.7.2 *Consider the variant of our ISQRT example given in Example 2.6.1, where local variabes are used, and the result is returned.*

- *Propose suitable pre- and post-condition to specify the expected behavior.*
- *Propose suitable loop invariant and variant to prove the program, including its termination.*

Exercise 2.7.3 *Exponentiation, revisited.*

```
let ref r = 1.0 in
let ref p = x in
while n > 0 do
  if mod n 2 = 1 then r := r *. p;
  p := p *. p;
  n := div n 2
done;
r
```

- *Propose pre- and post-condition.*
- *Propose suitable loop invariant and variant.*
- *Add lemmas and assertions as hints for the proof.*

Bibliography

[1] F. Bobot, J.-C. Filliâtre, C. Marché, and A. Paskevich. Why3: Shepherd your herd of provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, pages 53–64, Wrocław, Poland, August 2011. URL `http://proval.lri.fr/publications/boogie11final.pdf`.

[2] P. Herms, C. Marché, and B. Monate. A certified multi-prover verification condition generator. In R. Joshi, P. Müller, and A. Podelski, editors, *Verified Software: Theories, Tools, Experiments (4th International Conference VSTTE)*, volume 7152 of *Lecture Notes in Computer Science*, pages 2–17, Philadelphia, USA, Jan. 2012. Springer. URL `http://proval.lri.fr/publications/herms12vstte.pdf`.

[3] C. Marché and A. Tafat. Weakest precondition calculus, revisited using Why3. Research Report RR-8185, INRIA, Dec. 2012.

[4] C. Marché and A. Tafat. Calcul de plus faible précondition, revisité en Why3. In *Vingt-quatrièmes Journées Francophones des Langages Applicatifs*, Aussois, France, Feb. 2013.