Chapter 3

Types, Local Variables and Procedures

The goal of this chapter is to extend the basic IMP language of the previous chapter towards more structured programming:

- base types other than integers,
- local declarations of immutable variables in expressions,
- local declarations of variables in statements, both immutable and mutable,
- procedures.

With such extensions, we need to guarantee that programs are well-formed: we add to our language a type system that defines the class of well-typed formulas and well-typed programs, in a classical way.

3.1 New base data types, let bindings

Typically, we assume that the language of expressions does not involve only integers, but also booleans and real numbers.

The type of booleans typically comes with the constants true and false, the classical connectives and, or, not.

From now we also consider that the comparison operators on integers return a boolean. Similarly, we consider that in statements, the conditions of the if and the while are booleans instead of integers.

The type of real numbers comes with real constants and the standard operators $+,-,\times$ and the comparisons. Division is not considered for the same reason as for integers: it is not always defined. For convenience, we may also consider classical operations as such min, max, abs, but also sin, cos, etc.

We augment the grammar of expressions with the if-expression and the let binding:

$$e ::= \texttt{if}\ b\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \mid \texttt{let}\ x = e_1\ \texttt{in}\ e_2$$

The if-expression should not be confused with the if statement.

Example 3.1.1 *The following program computes an approximation of the cosine function*

```
val x : ref real
val res : ref real

res := let y = x*x in 1.0 − 0.5 ∗ y + 0.04166666 ∗ y ∗ y
```

3.1.1 Typing rules

The well-typedness of an expression is defined with respect to a typing environment Γ that maps identifiers to types. In this chapter, the grammar of types τ is

$$\begin{array}{rcl} \tau & ::= & b \mid \texttt{ref}\ b \\ b & ::= & \texttt{int} \mid \texttt{real} \mid \texttt{bool} \end{array}$$

`ref` denotes types of mutable variables, or *references*.

A typing judgment for expressions is denoted $\Gamma \vdash e : \tau$, meaning that e has type τ in environment Γ.

Important note: we forbid reference to reference. In other words, a reference is not itself a value. The type of an expression is always a base type, not a reference.

Typing expressions

The typing rules for constants are as follows,

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

$$\frac{}{\Gamma \vdash \mathit{true} : \texttt{bool}} \qquad \frac{}{\Gamma \vdash \mathit{false} : \texttt{bool}}$$

The typing rules for variables are

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

Notice that the OCaml programming language and the Why3 language require to explicitly write $!x$ to get the value of a reference x. For our language, since the reference x itself is not a value, the dereferencing is mandatory, and thus we do not require to write the bang symbol.

The typing rule for built-in binary operators is

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ \mathit{op}\ e_2 : \tau}$$

for each operator *op* expecting arguments of respective types τ_1 and τ_2 and returning a value of type τ.

The rules for the if-expression and the let binding are

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

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

Typing propositions

Propositions should not be considered as boolean expressions. Remember that in our language they cannot be used in place of any expression, e.g. they cannot be used in an assignment statement: we cannot write something like

x := forall y ...

To be able to type propositions, we thus consider a typing judgment of a special form $\Gamma \vdash p : \mathit{prop}$. The rules are as follows. The first one allows to consider a boolean expression as a proposition.

$$\frac{\Gamma \vdash e : \texttt{bool}}{\Gamma \vdash e : \mathit{prop}}$$

The typing of binary connectives is given by

$$\frac{\Gamma \vdash p_1 : \mathit{prop} \qquad \Gamma \vdash p_2 : \mathit{prop}}{\Gamma \vdash p_1 \mathit{\ op\ } p_2 : \mathit{prop}}$$

and finally typing quantified formulas is given by a rule similar to the let binding:

$$\frac{\{x : \tau\} \cdot \Gamma \vdash p : \mathit{prop}}{\Gamma \vdash \forall x : \tau, p : \mathit{prop}}$$

where τ is a base type.

3.1.2 Operational semantics

Since we now have local immutable variables, the operational semantics of expressions and statements are defined in a context that is a pair Σ, Π where Σ remains as in Chapter 2, that is, it maps pairs (global identifier, label) to values, whereas Π maps local identifiers to values.

The semantics of expressions is defined on well-typed expressions by

$$\begin{aligned}
[\![n]\!]_{\Sigma,\Pi} &= n \\
[\![x]\!]_{\Sigma,\Pi} &= \Sigma(x, \mathit{Here}) \\
&\quad \text{if } x : \texttt{ref}\ \tau \\
[\![x@L]\!]_{\Sigma,\Pi} &= \Sigma(x, L) \\
&\quad \text{if } x : \texttt{ref}\ \tau \\
[\![x]\!]_{\Sigma,\Pi} &= \Pi(x) \\
&\quad \text{if } x \text{ not a reference} \\
[\![e_1\ \mathit{op}\ e_2]\!]_{\Sigma,\Pi} &= [\![e_1]\!]_{\Sigma,\Pi}\ [\![\mathit{op}]\!]\ [\![e_2]\!]_{\Sigma,\Pi} \\
[\![\texttt{if}\ c\ \texttt{then}\ e_1\ \texttt{else}\ e_2]\!]_{\Sigma,\Pi} &= [\![e_1]\!]_{\Sigma,\Pi} \\
&\quad \text{if } [\![c]\!]_{\Sigma,\Pi} = \text{true} \\
[\![\texttt{if}\ c\ \texttt{then}\ e_1\ \texttt{else}\ e_2]\!]_{\Sigma,\Pi} &= [\![e_2]\!]_{\Sigma,\Pi} \\
&\quad \text{if } [\![c]\!]_{\Sigma,\Pi} = \text{false} \\
[\![\texttt{let}\ x = e_1\ \texttt{in}\ e_2]\!]_{\Sigma,\Pi} &= [\![e_2]\!]_{\Sigma,\Pi'} \\
&\quad \text{where } \Pi' = \{x = [\![e_1]\!]_{\Sigma,\Pi}\} \cdot \Pi
\end{aligned}$$

The semantics of formulas is also modified accordingly.

3.1.3 Hoare Logic and Weakest Preconditions

Since only the language of expressions is changed, rules for Hoare logic and WP remain the same. Of course, we apply these rules only on well-typed formulas and well-typed programs.

In practice, since the new base types, the new operators, the if-expression, and the let bindings, will appear in generated proof obligations, one should take care that theorem provers support these extensions. The Why3 system transforms formulas (e.g. transformation of if-expressions and/or let-expressions into equivalent formulas) before sending them to provers.

The let binding in expressions allows to propose an alternative presentation for the assignment rule:

$$\overline{\{\texttt{let}\ \xi = e\ \texttt{in}\ P[x \leftarrow \xi]\}x := e\{P\}}$$

Notice that we avoid the formulation $\{\texttt{let}\ x = e\ \texttt{in}\ P\}x := e\{P\}$ which would not be rigorous enough: the identifier x would change from the name of a global variable into the name of a local, immutable, variable. In other words, the precondition and postcondition cannot be typed in the same typing environment.

3.2 Local variables in statements

We extend the syntax of statements with

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

The first construct introduces an immutable variable, whereas the second introduces a mutable variable, which can be assigned as global variables can.

Example 3.2.1 *The following is the ISQRT example reformulated using a local variable for sum.*

```
val x, res : ref int

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

3.2.1 Typing rules for statements

The typing judgment for statements is denoted $\Gamma \vdash s : wf$ standing for "s is well-formed in environment Γ".

$$\frac{x : \texttt{ref}\ b \in \Gamma \qquad \Gamma \vdash e : b}{\Gamma \vdash x := e : wf}$$

$$\frac{\Gamma \vdash e : b \qquad \{x : b\} \cdot \Gamma \vdash s : wf}{\Gamma \vdash \texttt{let}\ x = e\ \texttt{in}\ s : wf}$$

$$\frac{\Gamma \vdash e : b \qquad \{x : \texttt{ref}\ b\} \cdot \Gamma \vdash s : wf}{\Gamma \vdash \texttt{let}\ x = \texttt{ref}\ e\ \texttt{in}\ s : wf}$$

3.2.2 Operational semantics

One-step execution is now a relation of the form

$$\Sigma, \Pi, s \rightsquigarrow \Sigma', \Pi', s'$$

The rules of Chapter 2 are unchanged, except that the additional Π must be added on each side of each reduction. The additional rules are:

$$\Sigma, \Pi, \texttt{let}\ x = e\ \texttt{in}\ s \rightsquigarrow \Sigma, \{x = [\![e]\!]_{\Sigma,\Pi}\} \cdot \Pi, s$$

$$\Sigma, \Pi, \texttt{let}\ x = \texttt{ref}\ e\ \texttt{in}\ s \rightsquigarrow \{(x, \textit{Here}) = [\![e]\!]_{\Sigma,\Pi}\} \cdot \Sigma, \Pi, s$$

3.2.3 Hoare logic rules

The rules are very close to those of the assigment:

$$\frac{\{P\}s\{Q\}}{\{P[x \leftarrow e]\}\texttt{let } x = e \texttt{ in } s\{Q\}}$$

$$\frac{\{P\}s\{Q\}}{\{P[x \leftarrow e]\}\texttt{let } x = \texttt{ref } e \texttt{ in } s\{Q\}}$$

As for assignment, there are alternative formulations using let binding in formulas:

$$\frac{\{P\}s\{Q\}}{\{\texttt{let } x = e \texttt{ in } P\}\texttt{let } x = e \texttt{ in } s\{Q\}}$$

$$\frac{\{P\}s\{Q\}}{\{\texttt{let } \xi = e \texttt{ in } P[x \leftarrow \xi]\}\texttt{let } x = \texttt{ref } e \texttt{ in } s\{Q\}}$$

3.2.4 WP rules

The rules for the weakest liberal precondition are

$$\mathrm{WLP}(\texttt{let } x = e \texttt{ in } s, Q) = \mathrm{WLP}(s, Q)[x \leftarrow e]$$

$$\mathrm{WLP}(\texttt{let } x = \texttt{ref } e \texttt{ in } s, Q) = \mathrm{WLP}(s, Q)[x \leftarrow e]$$

and alternatives are

$$\mathrm{WLP}(\texttt{let } x = e \texttt{ in } s, Q) = \texttt{let } x = e \texttt{ in } \mathrm{WLP}(s, Q)$$

$$\mathrm{WLP}(\texttt{let } x = \texttt{ref } e \texttt{ in } s, Q) = \texttt{let } \xi = e \texttt{ in } \mathrm{WLP}(s, Q)[x \leftarrow \xi]$$

The rules for WP are exactly the same.

3.3 Procedures, Modular Verification

Non-trivial programs are structured into sub-programs (functions, procedures, etc.) and at a higher level into modules. Visibility rules (local variables, local procedures, etc.) allow the programmer to hide implementation details from the callers of a sub-program.

When proving such a structured program, one naturally expects a *modular* proof, that follows the modular structure of the program. Thus, reasoning on a sub-program call should consider an *abstract* view of the behavior of that sub-program: a specification of what it does, without telling how it does it. The notation of Hoare triples is a good candidate for an abstraction of a given sub-program. In the literature, it is known as the *subprogram contract*, a term that was first used in the context of the Eiffel language [2]

We now add to the IMP language the notion of procedures and procedure calls. These procedures have parameters which are seen as immutable local variables. Also, using the term “procedure” here means that those are not returning any value, and are a new kind a statement, but are not usable inside expressions.

3.3.1 Syntax

The syntax of programs is given as follows. A program is a sequence of declarations. Each declaration is either the declaration of a global variable, whose type is a reference, or the declaration of a procedure. A procedure is declared with a possibly empty list of parameters, whose types are base type (not reference), a contract, and a body. The contract is made of the precondition, the list of variables possibly modified, and the postcondition.

$$
\begin{array}{rcl}
prog & ::= & decl^* \\
decl & ::= & vardecl \mid procdecl \\
vardecl & ::= & \textsf{var } id : \texttt{ref } basetype \\
procdecl & ::= & \textsf{procedure } id(params)\textsf{:}contract \textsf{ body } s \\
params & ::= & \varepsilon \mid param(,param)^* \\
param & ::= & id : basetype \\
contract & ::= & \textsf{requires } e \textsf{ writes } id(,id)^* \textsf{ ensures } e
\end{array}
$$

The syntax of statements is augmented with procedure call:

$$s ::= id(e(,e)^*)$$

Notice that arguments to procedure calls are expressions, and as such they do not have any side-effects.

Example 3.3.1 *The following is a program where our ISQRT example is now a procedure that takes its argument as a parameter, and stores its result in the global variable res. It also contains a simple test procedure.*

```
val res: ref int

procedure isqrt(x:int):
  requires x ≥ 0
  writes res
  ensures res ≥ 0 ∧ sqr(res) ≤ x < sqr(res+1)
body
  res := 0;
  let sum = ref 1 in
  while sum ≤ x do
    res := res + 1; sum := sum + 2 ∗ res + 1
  done

procedure test():
  requires true
  writes res
  ensures res = 6
body
  isqrt(42)
```

3.3.2 Typing rules for procedures

Let's assume that a procedure p is declared under the form

procedure $p(x_1 : \tau_1, \ldots, x_n : \tau_n)$:
  requires *Pre*
  writes $\vec{w}$
  ensures *Post*
  body *Body*

The rule expressing that such a declaration d is well-formed is

$$\frac{\Gamma' = \{x_i : \tau_i | 1 \leq i \leq n\} \cdot \Gamma \qquad \Gamma' \vdash \textit{Pre} : \textit{formula} \qquad \vec{w} \subseteq \Gamma \qquad \Gamma' \vdash \textit{Post} : \textit{formula} \qquad \Gamma' \vdash \textit{Body} : \textit{wf}}{\Gamma \vdash d : \textit{wf}}$$

where Γ contains the declaration of global references

The typing rule for a call to p is

$$\frac{\Gamma \vdash e_i : \tau_i}{\Gamma \vdash p(e_1, \ldots, e_n) : \textit{wf}}$$

Notice that we allow recursive calls. We even allow mutually recursive procedures.

3.3.3 Operational Semantics

The small-step semantics of a procedure call is defined as follows.

$$\frac{\Pi' = \{x_i \leftarrow [\![e_i]\!]_{\Sigma,\Pi}\} \qquad [\![\textit{Pre}]\!]_{\Sigma,\Pi'} \text{ holds}}{\Sigma, \Pi, p(e_1, \ldots, e_n) \rightsquigarrow \Sigma, \Pi', (\textit{Body}; \mathsf{return}(\textit{Post}, \Pi))}$$

where p is declared as in the previous section. return is a dummy statement, whose semantics is given by

$$\frac{[\![\textit{Post}]\!]_{\Sigma,\Pi'} \text{ holds}}{\Sigma, \Pi', \mathsf{return}(\textit{Post}, \Pi) \rightsquigarrow \Sigma, \Pi, \texttt{skip}}$$

Notice how the rules express that the body of the procedure is executed in a local environment that is disjoint from the procedure. Notice how the dummy return statement restores the former local environment.

Notice also that, as in the previous chapter, we define a blocking semantics for annotations (the pre- and the postcondition): the execution blocks if any annotation does not hold.

3.4 Hoare Logic for Procedures

A simple idea to prove a procedure call in Hoare logic would be to replace the call by the body of the procedure. However this is not a good idea: in some sense, this would mean we would reprove the body of a procedure each time it is called. What we want to do is to prove once for all that the body of a procedure satisfies its contract, and then use only the contract when reasoning on procedure calls. This is a *modular* approach.

The Hoare rule (for partial correctness) that expresses this idea is the following.

$$\frac{}{\{\textit{Pre}[x_i \leftarrow e_i]\} p(e_1, \ldots, e_n) \{\textit{Post}[x_i \leftarrow e_i]\}}$$

It simply means that any procedure called in a state satisfying the precondition (with the appropriate instance for the parameters) returns (if it terminates) a state that satisfies the postcondition. It just expresses formally the idea of a procedure contract.

3.4.1 Soundness

The soundness theorem must be expressed in a slightly different way as in the previous chapter: we need to express the idea of modularity of proofs. This is done by stating the following global hypothesis: for each procedure declared as

procedure $p(x_1 : \tau_1, \ldots, x_n : \tau_n)$:
  requires *Pre*
  writes $\vec{w}$
  ensures *Post*
  body *Body*

we assume that

1. the variables assigned in *Body* belong to $\vec{w}$,
2. the triple $\{Pre\}Body\{Post\}$ is derivable in Hoare logic.

Theorem 3.4.1 *Assuming the global hypothesis above holds, then any derivable triple $\{P\}s\{Q\}$ is valid for partial correctness.*

Proof.

Let's consider an execution of s of the form $\Sigma, \Pi, s \rightsquigarrow^* \Sigma', \Pi', \texttt{skip}$ where $[\![P]\!]_{\Sigma,\Pi}$ holds. We have to prove that $[\![Q]\!]_{\Sigma',\Pi'}$ holds.

We proceed by induction on the length of this execution, by case analysis on the form of s. If s is an assignment, a sequence, an if or a while, the proof proceeds as in the previous chapter. If it is some procedure call $p(e_1, \ldots, e_n)$, and since $\{P\}s\{Q\}$ is derivable, the only rules that apply are the consequence rule and the rule for the call, thus the derivation has the form

$$\frac{\models P \Rightarrow Pre[x_i \leftarrow e_i] \qquad \{Pre[x_i \leftarrow e_i]\}s\{Post[x_i \leftarrow e_i]\} \qquad \models Post[x_i \leftarrow e_i] \Rightarrow Q}{\{P\}s\{Q\}}$$

The formula $Post[x_i \leftarrow e_i] \Rightarrow Q$ thus holds in state Σ', Π', so to prove that Q holds we can prove that $Post[x_i \leftarrow e_i]$ holds.

The execution has the form

$$\Sigma, \Pi, s \rightsquigarrow \Sigma, \Pi_1, (Body; \mathsf{return}(Post, \Pi)) \rightsquigarrow^* \Sigma', \Pi', \texttt{skip}$$

where $\Pi_1 = \{x_i \leftarrow [\![e_i]\!]_{\Sigma,\Pi}\}$. By Lemma 2.1.2 on sequence execution, we have

$$\begin{aligned} \Sigma, \Pi_1, Body &\rightsquigarrow^* \Sigma_2, \Pi_2, \texttt{skip} \\ \Sigma_2, \Pi_2, \mathsf{return}(Post, \Pi)) &\rightsquigarrow^* \Sigma', \Pi', \texttt{skip} \end{aligned}$$

where Π_2 is Π_1 augmented with the local variables in *Body*. By definition of the operational semantics of the pseudo-statement return, we have $\Sigma' = \Sigma_2$ and $\Pi' = \Pi$.

From the operational semantics we know that $[\![Pre_{\Sigma,\Pi_1}]\!]$ holds. As the length of the execution of *Body* is smaller that those of the execution of s, and as the triple $\{Pre\}Body\{Post\}$ is assumed derivable by the global hypothesis, by induction we know that the triple is valid, that is, $[\![Post_{\Sigma_2,\Pi_2}]\!]$ holds. Since no local variable of *Body* may occur in the formula *Post*, indeed $[\![Post_{\Sigma_2,\Pi_1}]\!]$ holds, that is $[\![Post[x_i \leftarrow e_1]_{\Sigma_2,\emptyset}]\!]$, hence $[\![Post[x_i \leftarrow e_1]_{\Sigma_2,\Pi'}]\!]$, and we are done since $\Sigma_2 = \Sigma'$.

3.5 Weakest Preconditions

The rule for the procedure call is

$$\mathrm{WLP}(p(e_1,\ldots,e_n),Q) = \mathit{Pre}[x_i \leftarrow e_i] \wedge \forall \vec{y}, (\mathit{Post}[x_i \leftarrow e_i][\vec{w} \leftarrow \vec{y}] \Rightarrow Q[\vec{w} \leftarrow \vec{y}])$$

It states that the instance (for the given arguments) of the precondition should hold, and that the instance of the postcondition of p should imply Q, for any possible resulting values y of the assigned variables $\vec{w}$.

Definition 3.5.1 (Verification conditions for a program) *Assuming that each procedure p of a program is equipped with a contract, and that all assigned variables in the body are in $\vec{w}$. The set of verification conditions for a program is the set of formulas Pre $\Rightarrow$ WLP(Body, Post) for each procedure p of the program.*

Theorem 3.5.2 *If the verification conditions are valid logic formulas, then for all statement s and formula Q, the triple $\{\mathrm{WLP}(s,Q)\}s\{Q\}$ is valid for partial correctness.*

3.6 The Old Label

When stating the postcondition of a procedure, it is frequent that one wants to talk about the values the written variables had when entering the procedure. A special label Old is available for this purpose

Example 3.6.1 *Here is a toy procedure which increments the global variable res by a given amount.*

```
val res: ref int

procedure incr(x:int):
  requires true
  writes res
  ensures res = res@Old + x
body
  res := res + x
```

With this extension, the WLP rule should be modified as follows.

$$\begin{aligned}\mathrm{WLP}(p(e_1,\ldots,e_n),Q) = \mathit{Pre}[x_i \leftarrow e_i] \wedge \\ \forall \vec{y}, (\mathit{Post}[x_i \leftarrow e_i][\vec{w} \leftarrow \vec{y}][\vec{w}@\mathit{Old} \leftarrow \vec{w}@\mathit{Here}] \Rightarrow Q[\vec{w} \leftarrow \vec{y}])\end{aligned}$$

3.7 Procedures Throwing Exceptions

If a procedure may raise an exception, then this should be mentioned in its contract. A generalized contract has the form

requires *Pre*
writes $\vec{w}$
raises $E_1 \cdots E_k$
ensures *Post* | $E_1 \rightarrow \mathit{Post}_1$ | $\cdots$ | $E_k \rightarrow \mathit{Post}_k$

It states that the exceptions $E_1 \cdots E_k$ may be raised by the procedure, and no others, and: if it terminates normally then the formula *Post* holds, if it terminates in exception E_i then the formula Post_i holds.

The rule for WLP is extended to exceptional cases in a natural way:

$$\begin{array}{l}\mathrm{WLP}(p(e_1,\ldots,e_n), Q \mid E_j \to Q_j) = \mathit{Pre}[x_i \leftarrow e_i] \wedge \\ \quad \forall \vec{y}, \\ \qquad (\mathit{Post}[x_i \leftarrow e_i][\vec{w} \leftarrow \vec{y}] \Rightarrow Q[\vec{w} \leftarrow \vec{y}]) \\ \qquad (\mathit{Post}_j[x_i \leftarrow e_i][\vec{w} \leftarrow \vec{y}] \Rightarrow Q_j[\vec{w} \leftarrow \vec{y}])\end{array}$$

Example 3.7.1 *The following program is a "defensive" variant of our ISQRT example. Instead of requiring a non-negative argument, and returning a result rounded down, it raises an exception when the argument is negative or it is not a perfect square.*

```
val res: ref int
exception NotSquare

procedure isqrt(x:int):
  requires: true
  writes: res
  ensures: res ≥ 0 ∧ sqr(res) = x
       | NotSquare → forall n:int. sqr(n) ≠ x
body:
  if x < 0 then raise NotSquare;
  res := 0;
  let sum = ref 1 in
  while sum ≤ x do
    res := res + 1; sum := sum + 2 ∗ res + 1
  done;
  if res ∗ res ≠ x then raise NotSquare
```

3.8 Recursive Procedures and Termination

For recursive procedures, in a similar way as for loops, we need to add a variant to exhibit a measure that decreases between recursive calls. We allow to add a *variant clause* in the contracts, of the form

variant v for $\prec$

where v is an expression of some type τ and $\prec$ is a well-founded relation on values of type τ.

The formula for the weakest precondition of a recursive call is

$$\mathrm{WP}(p(e_1,\ldots,e_n), Q \mid E_i \to Q_i) = \mathit{Pre}[x_i \leftarrow e_i] \wedge v[x_i \leftarrow e_i] \prec v@\mathit{Init} \wedge \forall \vec{y}, \ldots$$

where Init is a label placed at the beginning of the body of the procedure. It thus states that the expression v, instantiated with the arguments of the call, is smaller than it was at the entrance to the procedure.

Example 3.8.1 *The following is a naive implementation of the factorial function returning its result in the global variable* res*. For simplicity we do not give any functional property in postcondition.*

```
val res: ref int

procedure fact(x:int):
  requires x ≥ 0
  variant x
```

```
  writes res
  ensures true
body
  if x = 0 then res := 1 else (fact(x−1) ; res := res ∗ x)
```

We do not precise the ordering since we use the default one introduced in Chapter 2. The WP at the recursive call is

$$(x \geq 0)[x \leftarrow x-1] \wedge x[x \leftarrow x-1] \prec x@Init$$

which reduces to

$$x-1 \geq 0 \wedge x-1 < x \wedge x \geq 0$$

which is valid under the premises $x \geq 0$ (the precondition) and $x \neq 0$ (the negation of the condition of the if).

3.8.1 Mutually Recursive Procedures

The same kind of WP rule is able to handle the case of mutually recursive procedures. If two procedures $p(\vec{x})$ and $q(\vec{y})$ may call each other, then each of them should be given its own variant v_p (resp. v_q) in their contract, but with the same well-founded ordering $\prec$. Then, when p calls $q(\vec{e})$ the WP should include

$$v_q[\vec{y} \leftarrow \vec{e}] \prec v_p@Init.$$

and symmetrically when q calls p

It generalizes naturally to any number of procedures that can call each other recursively.

3.9 Exercises

Exercise 3.9.1 *Prove the test procedure of Example 2.1.1 using Hoare logic and then using WLP.*

Exercise 3.9.2 *Using the incr procedure of Example 3.6.1, prove the test*

```
procedure test():
  requires res = 36
  writes res
  ensures res = 42
body
  incr(6)
```

using Hoare logic and then using WLP.

Exercise 3.9.3 *The McCarthy's 91 function [1] is defined on non-negative integers by the recursive equation*

$$f91(n) = \texttt{if}\ n \leq 100\ \texttt{then}\ f91(f91(n+11))\ \texttt{else}\ n-10$$

The following is a canevas for a procedure that computes $f91(n)$ into the global variable res.

```
  val res : ref int

  procedure f91 (n:int) :
  requires n ≥ 0
  variant ?
  writes ?
```

```
  ensures ?
body
  if n ≤ 100 then f91 (n + 11); f91 !res
  else res := n − 10
```

1. *Fill-in the contract in order to prove the total correctness of this procedure.*

2. *Would it be possible to prove the total correctness using only true as postcondition?*

Bibliography

[1] Z. Manna and J. McCarthy. Properties of programs and partial function logic. In *Machine Intelligence*, volume 5, pages 79–98, 1970.

[2] B. Meyer. *Eiffel: The Language*. Prentice Hall, Hemel Hempstead, 1992.