Chapter 5

Call by Reference, Aliasing Issues

The goal of this chapter is to address the so-called *aliasing* phenomenon in programming and the issues it raises when proving a program. Section 5.1 focuses on the case of *call by reference*. Section 5.2 introduces the possibility of making side effects inside expressions, and discusses issues regarding the order of evaluation.

5.1 Call by Reference

In the previous chapter, values were passed to procedures using parameters that are similar to immutable variables. This is the so-called *call by value* semantics. The *call by reference* semantics sees parameters as mutable variables instead. If such a parameter is assigned inside the body of the procedure, then the global variable passed as an argument during the call is modified too.

The need to add call by reference to the language of the previous chapter is motivated by the need of more genericity in the programs. An example to illustrate this is as follows, where the goal is to define a *module* implementing stacks of integers.

```
type stack = list int

val s:ref stack

procedure push(x:int):
  writes s
  ensures s = Cons(x,s@Old)
  body ...
```

The procedure push puts the given value x on top of the stack s. But in the case we need to program some other procedure that needs two stacks, it would be a major burden if we need to copy the procedure, to make it operate on another stack. From the point of view of proofs, it is even worse because we should prove the correctness of this copy.

In other words, if we want to provide a module for stacks that is naturally *reusable*, we should be able to parameterize the procedure push by the stack it operates on. A syntax for that is as follows.

```
type stack = list int

procedure push(s:ref stack,x:int):
  writes s
  ensures s = Cons(x,s@Old)
```

where the stack s is now a *reference parameter* of push. A program that uses two stacks can now be easily written, e.g.

```
 val s1,s2: ref stack

 procedure test():
  ensures head(s1) = 13 ∧ head(s2) = 42
  body push(s1,13); push(s2,42)
```

Historically, call by reference is present in older programming languages like PASCAL (parameters annotated with keyword VAR), Ada (parameters annotated with out or inout), Fortran, etc. For more modern languages like C or Java, such a feature is not present since the ability to pass a pointer or an object (i.e. internally a memory address in both cases) as argument can be used to modify mutable data, and thus simulates call by reference. In functional languages like OCaml, mutable data are explicitly typed using **ref**, hence call by reference is explicitly visible in the types of parameters.

Notice that modern versions of Ada also provide pointers and objects, however when Ada is used for developing critical software, only the subset Spark is recommended, that allows out or inout parameters but no pointers. The ultimate reason, as we will see in this chapter, is that call by reference is significantly easier to handle than general pointers when one wants to prove a program.

5.1.1 Syntax

A procedure declaration now allows both value parameters and reference parameters. For the sake of simplicity, we assume that reference parameters are given first, although in practice they can come in any order. The shape of a procedure declaration is thus as follows.

procedure $p(y_1 : \text{ref } \tau_1, \dots, y_k : \text{ref } \tau_k, x_1 : \tau'_1, \dots, x_n : \tau'_n)$:
  $\dots$

where the y_i are the reference parameters and the x_j are the parameters passed by value.

It should be noted that when calling such a procedure, it is not possible to pass any expression for the effective arguments of the y_i: since y_i is intended to be assigned, the corresponding argument has to be a mutable variable itself. The general shape of a call is thus

$p(z_1, \dots, z_k, e_1, \dots, e_n)$

where each z_i must be a mutable variable.

5.1.2 Operational Semantics

Defining the operational semantics of call by reference is not a trivial matter. There is a kind of "intuitive" semantics, that expresses what we informally expect from a call by reference, which can be formalized by a syntactic substitution in the body:

$$\frac{\Pi' = \{x_i \leftarrow [\![e_i]\!]_{\Sigma,\Pi}\} \quad [\![pre]\!]_{\Sigma,\Pi'} \text{ holds} \quad Body' = Body[y_j \leftarrow z_j]}{\Sigma, \Pi, p(z_1, \dots, z_k, e_1, \dots, e_n) \rightsquigarrow \Sigma, \Pi', (\texttt{Old} : Body'; return(post, \Pi))}$$

This rule is the same rule for the procedure call in previous chapter (regarding the parameters passed by value) but we replace each occurrence of reference parameters by the corresponding reference argument in the body of the executed procedure.

This semantics captures the informal idea of calling by reference, but it is not used in practice when interpreting or compiling a program, because there is no simple way to make a "copy" of the body of a procedure each time it is called.

Historically, there have been several variants proposed to implement call by reference. One of the older techniques is the semantics called *copy/restore*. There are reserved memory locations for the reference parameters, the values of the effective arguments are copied there when the procedure is called, and then the final values are copied back to the variables given as arguments. Such a semantics is formalized by the following rules. In the rule for call below, the reference parameters are added into the current state.

$$\frac{\Sigma' = \Sigma[y_j \leftarrow \Sigma(z_j)] \qquad \Pi' = \{x_i \leftarrow [\![e_i]\!]_{\Sigma,\Pi}\} \qquad [\![pre]\!]_{\Sigma,\Pi'} \text{ holds}}{\Sigma, \Pi, p(z_1, \dots, z_k, e_1, \dots, e_n) \rightsquigarrow \Sigma, \Pi', (\texttt{Old} : \mathit{Body}; \mathit{return}(\mathit{post}, \Pi))}$$

In the rule for the dummy statement *return* below, the state is updated, to formalize the "restore" step.

$$\frac{[\![post]\!]_{\Sigma,\Pi'} \text{ holds} \qquad \Sigma' = \Sigma[z_j \leftarrow \Sigma(y_j)]}{\Sigma, \Pi', \mathit{return}(\mathit{post}, \Pi) \rightsquigarrow \Sigma', \Pi, \texttt{skip}}$$

The important point to notice here is that *it is not equivalent to the intuitive semantics above*. Differences may appear in presence of *aliasing*, that is if two different variable names indeed denote the same variable. An example is as follows.

```
val g : ref int

procedure p(x:ref int):
  body x := 1; x := g + x;

procedure test():
  body g := 0; p(g);
```

In the intuitive semantics, executing p(g) means executing the body of p where x is replaced by g, that is

g := 1; g := g + g;

and thus g = 2 at the end. With the copy/restore semantics, executing p(g) means executing x := 1; x := g + x in a state where x has value 0 (the current value of g), which results in a state where g = 0 and x = 1, and the value of x is copied back to g thus g=1.

Such a difference in the two semantics comes from the aliasing between g and x when calling p(g): g and x are different names for the same memory location.

In a context where we want to prove programs, this difference of semantics is of course an issue. On the example above, one would naturally specify the procedure p as follows:

```
procedure p(x:ref int):
  writes x
  ensures x = g+1
  body x := 1; x := g + x;
```

The postcondition is proved valid with either Hoare logic or Weakest Precondition calculus. However, for the test code g := 0; p(g), if we use the rules of the previous chapter as they are we could prove both

- g=0, because the contract of p says that g is not changed by the procedure ;
- g=1 because the postcondition of p says that g = g@old+1.

This happens for several reasons:

- The post-condition of p is proved under an implicit assumption that x and g are not alias;

- The **writes** clause says that a call to p cannot modify anything but x, but it is not enough to say that the *name* g is different from the *name* x to ensure that g and x do not denote the same variable.

This clearly shows that the rules of Hoare logic and WP cannot be use in presence of call by reference without any precautions.

The program below illustrates another aliasing issue.

```
procedure p(x:ref int, y:ref int):
  writes x y
  ensures x = 1 ∧ y = 2
  body x := 1; y := 2
```

The post-condition is natural for the given code, and indeed can be proved valid if we use the rules we know about Hoare logic and weakest preconditions. However in the following context:

```
val g : ref int
procedure test():
  body p(g,g);
```

one could derive g = 1 ∧ g = 2 which is inconsistent. This time, this is because two reference parameters are alias.

Another example is as follows.

```
val g1 : ref int
val g2 : ref int

procedure p(x:ref int):
  writes g1 x
  ensures g1 = 1 ∧ x = 2
  body g1 := 1; x := 2

procedure test():
  body
    p(g2); assert g1 = 1 ∧ g2 = 2; (* OK *)
    p(g1); assert g1 = 1 ∧ g1 = 2; (* ??? *)
```

The first call p(g2) is OK, but the second one p(g1) is not, one could derive g1 = 1 ∧ g1 = 2 which is inconsistent.

5.1.3 Typing, Alias-Freedom Condition

To prevent the unexpected behaviors presented above, we have to make sure that no alias occur. For this purpose we need to add a new reads clause in procedure contracts, analogous to the **writes** clause, to specify the references that are accessed, but not assigned.

With this extended form of contract, we can prevent unexpected reference aliasing thanks to an additional premise in the typing rule for procedure calls. For a procedure declared under the form

```
procedure $p(y_1 : \texttt{ref}\ \tau_1, \dots, y_k : \texttt{ref}\ \tau_k, x_1 : \tau'_1, \dots, x_n : \tau'_n)$:
  writes $\vec{w}$
  reads $\vec{r}$
```

The typing rule for a call to p is extended with additional premises:

$$\frac{\dots \quad \forall ij, i \neq j \rightarrow z_i \neq z_j \qquad \forall ij, z_i \neq w_j \qquad \forall ij, z_i \neq r_j}{... \vdash p(z_1, \dots, z_k, ...) : wf}$$

In other words, the effective arguments z_i must be distinct, and each effective argument z_i must neither be read nor written by p [1]

Theorem 5.1.1 (Soundness in presence of call by reference) *If a program is well-typed, with the Alias-Freedom restriction above, then*

- *Semantics by substitution and by copy/restore coincide*
- *Hoare rules of previous chapter remain correct*
- *WP rules of previous chapter remain correct*

Indeed, the rules are almost unchanged: we must take care that for procedure call, appropriate substitution of the reference parameters by effective arguments is done, so the WP rule for call is as follows.

$$\begin{array}{l}\mathrm{WLP}(p(z_1,\dots,z_k,e_1,\dots,e_n),Q) = \textit{Pre}[x_i \leftarrow e_i][y_j \leftarrow z_j] \\ \quad \wedge \forall \vec{y}, (\textit{Post}[x_i \leftarrow e_i][y_j \leftarrow z_j][\vec{w'}@\textit{Here} \leftarrow \vec{y}][\vec{w'}@\textit{Old} \leftarrow \vec{w'}@\textit{Here}] \Rightarrow Q[\vec{w'} \leftarrow \vec{y}])\end{array}$$

where $\vec{w'} = \vec{w}[y_j \leftarrow z_j]$

For example, the following program is well-typed, and can be proved correct using WP.

```
type stack = list int
procedure push(s:ref stack,x:int):
  writes s
  ensures s = Cons(x,s@Old)
  body s := Cons(x,s)

val s1,s2: ref stack
procedure test():
  ensures head(s1) = 13 ∧ head(s2) = 42
  body push(s1,13); push(s2,42)
```

The proof is as follows:

$$\begin{array}{l}\mathrm{WP}(push(s_1,13);push(s_2,42), head(s_1)=13 \wedge head(s_2)=42) \\ = \mathrm{WP}(push(s_1,13), \mathrm{WP}(push(s_2,42), head(s_1)=13 \wedge head(s_2)=42)) \\ = \mathrm{WP}(push(s_1,13), \\ \qquad \forall y, (s = Cons(x, s@Old))[x \leftarrow 42][s \leftarrow s_2][s_2 \leftarrow y][s_2@Old \leftarrow s_2] \rightarrow \\ \qquad\qquad (head(s_1)=13 \wedge head(s_2)42)[s_2 \leftarrow y]) \\ = \mathrm{WP}(push(s_1,13), \\ \qquad \forall y, y = Cons(42, s_2) \rightarrow (head(s_1)=13 \wedge head(y)=42)) \\ = \mathrm{WP}(push(s_1,13), (head(s_1)=13 \wedge head(Cons(42,s_2))=42)) \\ = \mathrm{WP}(push(s_1,13), head(s_1)=13) \\ = \forall y, (s = Cons(x, s@Old))[x \leftarrow 13][s \leftarrow s_1][s_1 \leftarrow y][s_1@Old \leftarrow s_1] \rightarrow \\ \qquad (head(s_1)=13)[s_1 \leftarrow y]) \\ = \forall y, (y = Cons(13, s_1)) \rightarrow head(y) = 13 \\ = head(Cons(13,s_1)) = 13 \\ = true\end{array}$$

In some sense the alias-freedom condition ensures that the second call to push does not have any effect on the first stack s1, and thus proving head(s1) = 13 is directly a consequence of the post-condition of push for the first call.

5.2 Expressions with Side Effects, Function Calls

The ability to call by reference provides genericity, allowing to improve the modularity of the programs we can write. In practice, it would be very handy to be able to program *functions* in the sense of subprograms that can return a value. Continuing our stack example, we would like to write a pop function

```
function pop(s:ref stack): int
```

which should both modify the stack s passed by reference (removing the head) and returning the head element.

Of course, a function call will not be the same as a procedure call: it is part of an expression, like in

```
push(s,13);
push(s,42);
let x = pop(s) − pop(s) in
assert x = 29
```

This introduces an important novelty in our programming language: an expression may now have a side-effect. Moreover, as illustrated by the example above, the order of evaluation becomes important: the assertion above is true only if evaluation of the arguments of addition is from left to right.

Another novelty is that to specify a function, we should be able to specify, in postconditions, some properties about the result. For example, a specification of pop could be

```
function pop(s:ref stack): int
  requires s ≠ Nil
  writes s
  ensures result = head(s@Old) ∧ s = tail(s@Old)
```

where **result** is a keyword denoting the returned value.

All these novelties justify important changes in the syntax.

5.2.1 Syntax

Up to now, expressions were *pure* (no side-effects) and could be used indifferently in programs and in specifications. In our new syntax, these pure expressions are called *terms*. In programs, we now proceed the same as functional languages, that is, we do not make any distinction between expressions and statements. A statement is just an expression whose type is unit, which is a new base type, inhabited only by the constant denoted () (which is indeed the same as the former skip statement).

The grammar of expressions is

$$\begin{array}{rcll} e & ::= & n \mid r \mid \mathit{true} \mid \mathit{false} \mid () & \text{constants} \\ & & \mid x \mid e\ \mathit{op}\ e \mid x := e \mid e; e & \\ & & \mid \mathtt{let}\ \mathit{id} = e\ \mathtt{in}\ e \mid \mathtt{let}\ \mathit{id} = \mathtt{ref}\ e\ \mathtt{in}\ e & \text{local binding} \\ & & \mid \mathtt{if}\ e\ \mathtt{then}\ e\ \mathtt{else}\ e \mid \mathtt{while}\ e\ \mathtt{do}\ e & \\ & & \mid \mathtt{raise}\ \mathit{exn} \mid \mathtt{try}\ e\ \mathtt{with}\ \mathit{exn} \Rightarrow e' & \\ & & \mid L : e & \text{label} \\ & & \mid f(e, \ldots e) & \text{function call} \end{array}$$

5.2.2 Typing

The typing rules are given below.

constants

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

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

variables

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

binary operators

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

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

assignment

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

conditional

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

loop

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

local bindings

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

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

Function call

$$\frac{\begin{array}{c} z_i : \texttt{ref}\ \tau'_i \in \Gamma \qquad \Gamma \vdash e_i : \tau_i \\ \forall ij, i \neq j \to z_i \neq z_j \qquad \forall i, j, z_i \neq w_j \qquad \forall i, j, z_i \neq r_j \end{array}}{\Gamma \vdash f(z_1, \ldots, z_k, e_1, \ldots, e_n) : \tau}$$

if *f* declared as

function $f(y_1 : \texttt{ref}\ \tau'_1, \ldots, y_k : \texttt{ref}\ \tau'_k, x_1 : \tau_1, \ldots, x_n : \tau_n)$: τ
  requires *Pre*
  reads $\vec{r}$
  writes $\vec{w}$
  ensures *Post*
  body *Body*

5.2.3 Operational Semantics

The relation for one-step execution now operates on expressions instead of statements. It has the form

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

Formerly, the only possible final statement was `skip` (and also `raise` Exn in case of exceptions), now the possible final expression are all the constants of the logic. These are traditionally called the *values*. To make explicit the order of evaluation, the rules below distinguish the case of values and others.

Variables

$$\frac{x \text{ is an immutable variable} \qquad x = v \in \Pi}{\Sigma, \Pi, x \rightsquigarrow \Sigma, \Pi, v}$$

$$\frac{x \text{ is a reference} \qquad (x, \mathit{Here}) = v \in \Sigma}{\Sigma, \Pi, x \rightsquigarrow \Sigma, \Pi, v}$$

Let binding, immutable variable

$$\frac{\Sigma, \Pi, e_1 \rightsquigarrow \Sigma', \Pi', e'_1}{\Sigma, \Pi, \texttt{let } x = e_1 \texttt{ in } e_2 \rightsquigarrow \Sigma', \Pi', \texttt{let } x = e'_1 \texttt{ in } e_2}$$

$$\frac{}{\Sigma, \Pi, \texttt{let } x = v_1 \texttt{ in } e_2 \rightsquigarrow \Sigma, \{x = v_1\} \cdot \Pi, e_2}$$

Let binding, mutable variable

$$\frac{\Sigma, \Pi, e_1 \rightsquigarrow \Sigma', \Pi', e'_1}{\Sigma, \Pi, \texttt{let } x = \texttt{ref } e_1 \texttt{ in } e_2 \rightsquigarrow \Sigma', \Pi', \texttt{let } x = \texttt{ref } e'_1 \texttt{ in } e_2}$$

$$\frac{}{\Sigma, \Pi, \texttt{let } x = \texttt{ref } v \texttt{ in } e_2 \rightsquigarrow \{(x, \mathit{Here}) = v\} \cdot \Sigma, \Pi, e_2}$$

Binary operators

$$\frac{\Sigma, \Pi, e_1 \rightsquigarrow \Sigma', \Pi', e'_1}{\Sigma, \Pi, e_1 \mathbin{op} e_2 \rightsquigarrow \Sigma', \Pi', e'_1 \mathbin{op} e_2}$$

$$\frac{\Sigma, \Pi, e_2 \rightsquigarrow \Sigma', \Pi', e'_2}{\Sigma, \Pi, v_1 \mathbin{op} e_2 \rightsquigarrow \Sigma', \Pi', v_1 \mathbin{op} e'_2}$$

$$\frac{v = v_1 \mathbin{op} v_2}{\Sigma, \Pi, v_1 \mathbin{op} v_2 \rightsquigarrow \Sigma, \Pi, v}$$

Assignment

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

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

Conditional

$$\frac{\Sigma, \Pi, e_1 \rightsquigarrow \Sigma', \Pi', e'_1}{\Sigma, \Pi, \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \rightsquigarrow \Sigma', \Pi', \texttt{if } e'_1 \texttt{ then } e_2 \texttt{ else } e_3}$$

$$\frac{}{\Sigma, \Pi, \texttt{if } \mathit{true} \texttt{ then } e_2 \texttt{ else } e_3 \rightsquigarrow \Sigma, \Pi, e_2}$$

$$\frac{}{\Sigma, \Pi, \texttt{if } \mathit{false} \texttt{ then } e_2 \texttt{ else } e_3 \rightsquigarrow \Sigma, \Pi, e_3}$$

Loop

$$\frac{}{\Sigma, \Pi, \texttt{while}\ e_1\ \texttt{do}\ e_2 \rightsquigarrow \Sigma, \Pi, \texttt{if}\ e_1\ \texttt{then}\ (e_2; \texttt{while}\ e_1\ \texttt{do}\ e_2)\ \texttt{else}\ ()}$$

Function call

$$\frac{\Sigma' = \Sigma[y_j \leftarrow \Sigma(z_j)] \qquad \Pi' = \{x_i \leftarrow v_i\} \qquad [\![pre]\!]_{\Sigma,\Pi'} \text{ holds}}{\begin{array}{l}\Sigma, \Pi, p(z_1, \dots, z_k, v_1, \dots, v_n) \rightsquigarrow \\ \quad \Sigma, \Pi', \texttt{let}\ v = \mathit{Body}\ \texttt{in}\ \mathit{return}(v, \mathit{Post}, \Pi)\end{array}}$$

$$\frac{[\![\texttt{let}\ \mathsf{result} = v\ \texttt{in}\ \mathit{Post}]\!]_{\Sigma,\Pi'} \text{ holds} \qquad \Sigma' = \Sigma[z_j \leftarrow \Sigma(y_j)]}{\Sigma, \Pi', \mathit{return}(v, \mathit{Post}, \Pi) \rightsquigarrow \Sigma', \Pi, v}$$

Notice that the dummy statement $return$ is now a dummy expression, taking an extra argument to denote the returned value v of the call. Notice then that the validity check of the postcondition is done when binding the reserved variable result to this value v.

5.2.4 Hoare Rules and Weakest Preconditions

Hoare logic rules and rules for computing WP must be modified in order to operate on expressions instead of statements. The modifications to perform are similar in both cases, and we focus here on the WP calculus.

Generally speaking, the operator WP now applies on an expression and a formula Q where in the latter the reserved variable name result can occur. The WP rules should allow to compute a formula $\mathrm{WP}(e, Q)$ such that if the expression e is executed in a state that satisfies $\mathrm{WP}(e, Q)$, then the final state and final value v should satisfy $Q[\mathsf{result} \leftarrow v]$. For example, we expect that $\mathrm{WP}((x := x + 13; x + 1), \mathsf{result} = 42)$ is $x = 28$.

Let's start with the rules for pure terms and for a let binding:

$$\begin{array}{rcl}\mathrm{WP}(t, Q) & = & Q[\mathsf{result} \leftarrow t] \\ \mathrm{WP}(\texttt{let}\ x = e_1\ \texttt{in}\ e_2, Q) & = & \mathrm{WP}(e_1, \mathrm{WP}(e_2, Q)[x \leftarrow \mathsf{result}])\end{array}$$

The rule for let binding allows us to provide simpler rules below, by assuming that the sub-expressions are pure terms. Indeed the rule for binary operators can be derived from the rules above by just noticing that $e_1\ \mathit{op}\ e_2$ has the same semantics as $\texttt{let}\ t_1 = e_1\ \texttt{in}\ \texttt{let}\ t_2 = e_2\ \texttt{in}\ t_1\ \mathit{op}\ t_2$ thanks to the choice of evaluation order from left to right. Hence

$$\begin{array}{rcl}\mathrm{WP}(e_1\ \mathit{op}\ e_2, Q) & = & \mathrm{WP}(\texttt{let}\ t_1 = e_1\ \texttt{in}\ \texttt{let}\ t_2 = e_2\ \texttt{in}\ t_1\ \mathit{op}\ t_2, Q) \\ & = & \mathrm{WP}(e_1, \mathrm{WP}(\texttt{let}\ t_2 = e_2\ \texttt{in}\ t_1\ \mathit{op}\ t_2, Q)[t_1 \leftarrow \mathsf{result}]) \\ & = & \mathrm{WP}(e_1, (\mathrm{WP}(e_2, \mathrm{WP}(t_1\ \mathit{op}\ t_2, Q))[t_2 \leftarrow \mathsf{result}])[t_1 \leftarrow \mathsf{result}]) \\ & = & \mathrm{WP}(e_1, (\mathrm{WP}(e_2, Q[\mathsf{result} \leftarrow t_1\ \mathit{op}\ t_2])[t_2 \leftarrow \mathsf{result}])[t_1 \leftarrow \mathsf{result}])\end{array}$$

The rule for assignment is the same as before, except for an extra substitution of the reserved variable result by $()$.

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

Similarly as for binary operator we can generalize the rule to any expression e in place of a pure term t:

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

The rule for conditional expressions is

$$\mathrm{WP}(\texttt{if}\, t \,\texttt{then}\, e_2 \,\texttt{else}\, e_3, Q) = \texttt{if}\, t \,\texttt{then}\, \mathrm{WP}(e_2, Q) \,\texttt{else}\, \mathrm{WP}(e_3, Q))$$

or more generally

$$\mathrm{WP}(\texttt{if}\, e_1 \,\texttt{then}\, e_2 \,\texttt{else}\, e_3, Q) = \mathrm{WP}(e_1, \texttt{if}\, \textsf{result} \,\texttt{then}\, \mathrm{WP}(e_2, Q) \,\texttt{else}\, \mathrm{WP}(e_3, Q))$$

The rule for the while loop is

$$\begin{array}{l} \mathrm{WP}(\texttt{while}\, e_1 \,\texttt{invariant}\, I \,\texttt{variant}\, t \,\texttt{do}\, e_2, Q) = \\ \quad I \wedge \forall \vec{y}, I \rightarrow \\ \qquad \mathrm{WP}(e_1, \texttt{if}\, \textsf{result} \,\texttt{then}\, \mathrm{WP}(e_2, I \wedge t < t@L) \,\texttt{else}\, Q)[x@L \leftarrow x@\mathit{Here}] \end{array}$$

The rule for function calls is very similar to the one before, we just need to add an extra quantification on the result of the call:

$$\begin{array}{l} \mathrm{WLP}(p(z_1, \ldots, z_k, t_1, \ldots, t_n), Q) = \mathit{Pre}[x_i \leftarrow t_i][y_j \leftarrow z_j] \\ \quad \wedge \forall \vec{y}\, \textsf{result}, (\mathit{Post}[x_i \leftarrow t_i][y_j \leftarrow z_j][\vec{w'}@\mathit{Here} \leftarrow \vec{y}][\vec{w'}@\mathit{Old} \leftarrow \vec{w'}@\mathit{Here}] \Rightarrow Q[\vec{w'} \leftarrow \vec{y}]) \end{array}$$

where $\vec{w'} = \vec{w}[y_j \leftarrow z_j]$.

Example 5.2.1 *Let's prove*

```
push(s,13); push(s,42);
let x = pop(s) − pop(s) in
assert x = 29;
```

using WP *calculus:*

$$\begin{array}{l} \mathrm{WP}(\texttt{let}\, x = \mathit{pop}(s) - \mathit{pop}(s) \,\texttt{in assert}\, x = 29, \mathit{true}) \\ = \mathrm{WP}(\mathit{pop}(s) - \mathit{pop}(s), \mathrm{WP}(\texttt{assert}\, x = 29, \mathit{true})[x \leftarrow \textsf{result}]) \\ = \mathrm{WP}(\mathit{pop}(s) - \mathit{pop}(s), (x = 29)[x \leftarrow \textsf{result}]) \\ = \mathrm{WP}(\mathit{pop}(s) - \mathit{pop}(s), \textsf{result} = 29) \\ = \mathrm{WP}(\texttt{let}\, t_1 = \mathit{pop}(s) \,\texttt{in let}\, t_2 = \mathit{pop}(s) \,\texttt{in}\, t_1 - t_2, \textsf{result} = 29) \\ = \mathrm{WP}(\mathit{pop}(s), \mathrm{WP}(\texttt{let}\, t_2 = \mathit{pop}(s) \,\texttt{in}\, t_1 - t_2, \textsf{result} = 29)[t_1 \leftarrow \textsf{result}]) \\ = \mathrm{WP}(\mathit{pop}(s), \mathrm{WP}(\mathit{pop}(s), \mathrm{WP}(t_1 - t_2, \textsf{result} = 29)[t_2 \leftarrow \textsf{result}])[t_1 \leftarrow \textsf{result}]) \\ = \mathrm{WP}(\mathit{pop}(s), \mathrm{WP}(\mathit{pop}(s), (t_1 - t_2 = 29)[t_2 \leftarrow \textsf{result}])[t_1 \leftarrow \textsf{result}]) \\ = \mathrm{WP}(\mathit{pop}(s), \mathrm{WP}(\mathit{pop}(s), t_1 - \textsf{result} = 29)[t_1 \leftarrow \textsf{result}]) \\ = \mathrm{WP}(\mathit{pop}(s), \\ \qquad (s \neq \mathit{Nil} \wedge \forall s_0\, \textsf{result}, \textsf{result} = \mathit{head}(s) \wedge s_0 = \mathit{tail}(s) \rightarrow t_1 - \textsf{result} = 29)[t_1 \leftarrow \textsf{result}]) \\ = \mathrm{WP}(\mathit{pop}(s), (s \neq \mathit{Nil} \wedge \forall s_0\, r_0, r_0 = \mathit{head}(s) \wedge s_0 = \mathit{tail}(s) \rightarrow t_1 - r_0 = 29)[t_1 \leftarrow \textsf{result}]) \\ \qquad \text{(renaming internal } \textsf{result} \text{ into } r_0 \text{ to avoid capture in the next step)} \\ = \mathrm{WP}(\mathit{pop}(s), (s \neq \mathit{Nil} \wedge \forall s_0\, r_0, r_0 = \mathit{head}(s) \wedge s_0 = \mathit{tail}(s) \rightarrow \textsf{result} - r_0 = 29)) \\ = s \neq \mathit{Nil} \wedge \forall s_1\, \textsf{result}, \textsf{result} = \mathit{head}(s) \wedge s_1 = \mathit{tail}(s) \rightarrow \\ \qquad (s_1 \neq \mathit{Nil} \wedge \forall s_0\, r_0, r_0 = \mathit{head}(s_1) \wedge s_0 = \mathit{tail}(s_1) \rightarrow \textsf{result} - r_0 = 29) \qquad (A) \end{array}$$

For the last expression (A) we must compute the WP through push(s,13); push(s,42);. *We get*

$$\begin{array}{l} \mathrm{WP}(\mathit{push}(s, 13), \mathrm{WP}(\mathit{push}(s, 42), A)) \\ = \mathrm{WP}(\mathit{push}(s, 13), \forall s_2 . s_2 = \mathit{Cons}(42, s) \rightarrow A[s \leftarrow s_2]) \\ = \forall s_3, s_3 = \mathit{Cons}(13, s) \rightarrow \forall s_2, s_2 = \mathit{Cons}(42, s) \rightarrow A[s \leftarrow s_2] \\ = A[s \leftarrow \mathit{Cons}(42, \mathit{Cons}(13, s))] \end{array}$$

which gives, after evaluation of head *and* tail *:*

$$\begin{aligned}
&Cons(42, Cons(13, s)) \neq Nil \land \forall s_1\ \mathsf{result}, \mathsf{result} = 42 \land s_1 = Cons(13, s) \to \\
&\qquad (s_1 \neq Nil \land \forall s_0\ r_0, r_0 = head(s_1) \land s_0 = tail(s_1) \to \mathsf{result} - r_0 = 29) \\
&= true \land (Cons(13, s) \neq Nil \land \forall s_0\ r_0, r_0 = 13 \land s_0 = s \to 42 - r_0 = 29) \\
&= true \land 42 - 13 = 29 \\
&= true
\end{aligned}$$

5.3 Exercises

Exercise 5.3.1 *Incrementations*

- *Specify and prove a procedure which takes a reference to a list of reals as argument, and increments by 1.0 each element of this list*
- *Specify and prove a procedure which takes a reference to an array of reals as argument, and increments by 1.0 each element of this array*

Exercise 5.3.2 *Below is a procedure that replaces the reference argument by its reverse.*

```
procedure rev_append(l : ref (list α))
  writes ?
  ensures ?
  body
    let r := ref l in
    l := Nil;
    try while true do
      invariant ?
      variant ?
      match r with
      | Nil → raise Break
      | Cons(x,y) → l := Cons(x,l); r := y
      done;
      absurd
    with Break → ()
```

1. *Fill the ? with appropriate annotations.*
2. *Prove the program using WP. Which general lemmas on lists are needed?*

Bibliography

[1] J.-C. Filliâtre. Verification of non-functional programs using interpretations in type theory. *Journal of Functional Programming*, 13(4):709–745, July 2003. URL `http://www.lri.fr/~filliatr/ftp/publis/jphd.pdf`.