Chapter 5

Aliasing

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 considers the general case of *pointer programs*.

5.1 Call by Reference

In Chapter 3, values were passed to functions 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 function, 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 chapters is motivated by the need of more genericity in 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

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

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

The function push puts the given value x on top of the stack s, whereas the pop function removes the top of the stack and returns its value.

But in the case we need to program some other function that needs two stacks, it would be a major burden if we need to copy the functions, to make them 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 functions push and pop by the stack it operates on. A syntax for that is as follows.

```
type stack = list int

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

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

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

```
  val s1,s2: ref stack

  function test(): int
    ensures head(s2) = 42 ∧ result = 13
    body push(s1,13); push(s2,42); pop(s1)
```

5.1.1 Aliasing Problems

Let's consider the following functions using stacks.

```
function test(s1,s2: ref stack) : unit
  ensures { head(s1) = 42 ∧ head(s2) = 13 }
  body push(s1,42); push(s2,13)

function wrong(s: ref stack) : int
  body test(s,s)
```

The post-condition of `test` is natural, but this means that when someone calls `test` with twice the same reference, like function `wrong` above, one could be able to prove both `head(s) = 42` and `head(s) = 13`, which is obviously a mistake. This happens because in function `test`, the post-condition can be established only under the assumption that parameters `s1` and `s2` denotes distinct mutable variables. This is an implicit *non-aliasing* hypothesis.

Indeed, aliasing is a major issue when proving programs. Deductive Verification Methods like Hoare logic or the Weakest Precondition Calculus implicitly require absence of aliasing.

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.

5.1.2 Syntax

A function 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 function declaration is thus as follows.

function $f(y_1 : \mathsf{ref}\ \tau_1, \dots, y_k : \mathsf{ref}\ \tau_k, x_1 : \tau'_1, \dots, x_n : \tau'_n)$:
  $\dots$

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 function, 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

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

where each z_i must be a mutable variable.

5.1.3 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:

$$\frac{\Pi' = \{x_i \leftarrow [\![t_i]\!]_{\Sigma,\Pi}\} \quad \Sigma, \Pi' \models \mathit{Pre} \quad \mathit{Body}' = \mathit{Body}[y_j \leftarrow z_j]}{\Sigma, \Pi, f(z_1, \dots, z_k, t_1, \dots, t_n) \rightsquigarrow \Sigma, \Pi', (\mathit{Old} : \texttt{let}\ \xi = \mathit{Body}'\ \texttt{in return}\ \xi, \mathit{Post}, \Pi)}$$

This rule is the same rule as for function calls in Chapter 3 (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 function.

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 function 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 function 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.

$$\frac{\Sigma' = \Sigma[y_j \leftarrow \Sigma(z_j)] \quad \Pi' = \{x_i \leftarrow [\![t_i]\!]_{\Sigma,\Pi}\} \quad \Sigma, \Pi' \models \mathit{Pre}}{\Sigma, \Pi, f(z_1, \dots, z_k, t_1, \dots, t_n) \rightsquigarrow \Sigma', \Pi', (\mathit{Old} : \texttt{let}\ \xi = \mathit{Body}\ \texttt{in return}\ \xi, \mathit{Post}, \Pi)}$$

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

$$\frac{\Sigma, \Pi' \models P[\mathsf{result} \leftarrow v] \quad \Sigma' = \Sigma[z_j \leftarrow \Sigma(y_j)]}{\Sigma, \Pi, (\texttt{return}\ v, P, \Pi') \rightsquigarrow \Sigma', \Pi', v}$$

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

function f(x:ref int):unit
  body x := 1; x := g+1

function test():unit
  body g:=0; f(g)
```

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

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

and thus `g = 2` at the end. With the copy/restore semantics, executing `f(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 `f(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 function `f` as follows:

```
function f(x:ref int):
  writes x
  ensures x = g+1
  body x := 1; x := g + 1;
```

The postcondition can be proved valid with Hoare Logic or with Weakest Precondition calculus. However, for the test code `g := 0; p(g)`, if we use the rules of Chapter 3 as they are we could prove both

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

This happens for several reasons:

- The post-condition of `f` is proved under an implicit assumption that `x` and `g` are not alias;
- The **`writes`** clause says that a call to `f` 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 used in presence of call by reference without any precautions.

The program below illustrates another aliasing issue.

```
function f(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
function test():
  body f(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

function f(x:ref int):
  writes g1 x
```

```
    ensures g1 = 1 ∧ x = 2
    body g1 := 1; x := 2

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

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

5.1.4 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 function 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 function calls. For a function declared under the form

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

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

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

In other words, the effective arguments z_i must be distinct, and each effective argument z_i must neither be read nor written by f [8]

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 chapters remain correct*
- *WP rules of previous chapters remain correct*

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

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

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

Example 5.1.2 *The following program is well-typed, and can be proved correct using WP.*

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

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

The proof is as follows:

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

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.*

Example 5.1.3 *The following program is correct:*

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

$$
\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),\\
\quad\quad(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}])\\
\quad\quad\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\\
\quad\quad(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*

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

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

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

5.1.5 About Creation of References

To program modules like the one for stacks, one also needs the ability to return newly created references. For example one would like to write

```
function create():ref stack
  ensures result = Nil
  body
   let ref s = Nil in s
```

To ensure that no unexpected aliasing occur, typing should require that a returned reference is always *fresh*, i.e. is distinct from any global mutable variable or reference parameters. For our language, this means it should return a local mutable variable. In the Why3 programming language which is a bit more general than ours, the control of aliasing is made using the notions of *regions* [11].

5.2 Pointer Programs

The goal of this section is to drop the hypothesis that we have until now, that is, references are not values of the language, and in particular there is nothing like a reference to a reference, and there is no way to program with data structures that can be modified *in-place*, such as records where fields can be assigned directly.

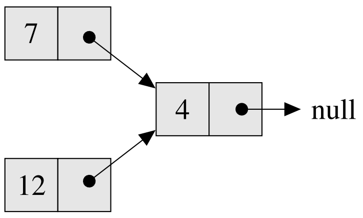
There is a fundamental difference of semantics between in-place modification of a record field and the way we modeled records in Chapter 4. The small piece of code below, in the syntax of the C programming language, is a typical example of the kind of programs we want to deal with in this chapter.

```
typedef struct List { int data; list next; } *list;

list create(int d, list n) {
  list l = (list)malloc(sizeof(struct List));
  l->data = d;
  l->next = n;
  return l;
}

void incr_list(list p) {
  while (p <> NULL) {
    p->data++; p = p->next;
    }
  }
```

Like call by reference, in-place assignment of fields is another source of aliasing issues. Consider this small test program

```
  void test() {
    list l1 = create(4,NULL);
    list l2 = create(7,l1);
    list l3 = create(12,l1);
    assert (l3->next->data == 4);
    incr_list(l2);
    assert (l3->next->data == 5);
  }
```

which builds the following structure:



the list node l_1 is shared among l_2 and l_3, hence the call to `incr_list`(l_2) modifies the second node of list l_3.

The goal of this chapter is to provide methods for specifying this kind of pointer programs, and reasoning on them in a sound way, taking possible aliasing into account.

In the next sections, we extend our language with pointers to records, and formalize the operational semantics. For the moment we do not consider allocation and deallocation. Memory allocation will be considered in Chapter 6.

5.2.1 Syntax

We use the same language of expressions with side effects as in the previous chapters, which we enrich with data types of pointers to records. The syntax of global declarations contains a new kind of constructs to declare record types, with the form:

$$\mathsf{record}\ id = \{f_1 : \tau_1; \cdots f_n : \tau_n\}$$

and where the grammar of types is extended accordingly:

$$\tau ::= \mathtt{int} \mid \mathtt{real} \mid \mathtt{bool} \mid \mathtt{unit} \mid id$$

The record declarations can be recursive: a field f_i of a record type S can itself be of type S. Construction of recursive "linked" data structures can be performed thanks to a *null* pointer which can be of any record type.

The grammar of expressions of the language is extended has follows.

$$\begin{array}{lll} e ::= & \mathtt{null} & \text{null pointer} \\ & \mid e \rightarrow f & \text{field access} \\ & \mid e \rightarrow f := e & \text{field update} \end{array}$$

We consider that a memory access is safe as soon as the considered pointer is not `null` (as in Java for instance).

Example 5.2.1 *The small example given at the beginning of this section, for incrementing a linked list, is written in our language as follows*

```
record List = { data : int ; next: List; }

function incr_list(l:List)
body
  let p = ref l in
  while p != null do
    p→data := p→data + 1;
    p := p→next
  done
```

5.2.2 Operational Semantics

To formalize the semantics of pointer programs, we first introduce a new kind of values: the *memory locations*, which concretely correspond to the addresses on a computer. This kind of values is denoted by the type `loc`. In other words, the pointer variables in a program are the variables whose values are of type `loc`. The special expression `null` is one particular value of the type `loc`. We assume that there are infinitely many values of type `loc`, in other words we assume an ideal infinite memory.

Instead of a pair (Σ, Π), a program state is now a triple $(\mathcal{H}, \Sigma, \Pi)$. Σ and Π still map variable identifiers to values, whereas $\mathcal{H}$ maps pairs $(\texttt{loc}, \text{field name})$ to values. We consider that $\mathcal{H}$ is a total map, that is, all memory locations are "allocated" (remind that allocation/deallocation is not supported).

Example 5.2.2 *A program state that corresponds to the informal example with lists from the introduction of this chapter is*

$$\begin{array}{rcl} \Pi & = & \emptyset \\ \Sigma & = & \{l_1 = loc_1; l_2 = loc_2; l_3 = loc_3\} \\ \mathcal{H} & = & \{(loc_1, data) = 4; (loc_2, data) = 7; (loc_3, data) = 12; \\ & & \quad (loc_1, next) = \texttt{null}; (loc_2, next) = loc_1; (loc_3, next) = loc_1\} \end{array}$$

where loc_1, loc_2, loc_3 are some values of type `loc`

The two additional rules we need to execute pointer programs are as follows, to respectively evaluate field access and field update.

$$\frac{v \neq \texttt{null}}{\mathcal{H}, \Sigma, \Pi, (v \to f) \rightsquigarrow \mathcal{H}, \Sigma, \Pi, \mathcal{H}(v, f)}$$

$$\frac{v_1 \neq \texttt{null}}{\mathcal{H}, \Sigma, \Pi, (v_1 \to f := v_2) \rightsquigarrow \mathcal{H}[(v_1, f) \leftarrow v_2], \Sigma, \Pi, ()}$$

5.2.3 Component-as-Array Model

The Component-as-Array model is a method that allows to encode the programs of our language extended with pointer to records into the language without pointers we had in the previous section. This technique allows to reuse the techniques for reasoning on programs using Hoare logic or WP, on pointer programs. This method is originally an old idea proposed by Burstall in 1972 [6] and was resurrected by Bornat in 2000 [5]. The idea is that we can collect all the field names declared in a given program, and view the heap $\mathcal{H}$ not as a unique map indexed by pairs (loc,field name) but as a finite collection of maps $\mathcal{H}_f$ indexed by loc, one for each field name f. If the field f is declared of type τ, the map $\mathcal{H}_f$ can be encoded into a reference to a purely applicative map from `loc` to τ, in the same way that an array is encoded by a reference to a purely applicative map from integers to τ.

The following is the common part of the component-as-array model, that can be used for any programs.

```
type loc
constant null : loc

function acc(field: ref (map loc α),l:loc) : α
  requires l ≠ null
  reads field
  ensures result = select(field,l)

function upd(field: ref (map loc α),l:loc,v:α):unit
  requires l ≠ null
  writes field
  ensures field = store(field@Old,l,v)
```

The two functions above allow us to simulate the operational semantics, that requires pointers to be non-null when accessing memory.

In other words, this *Component-as-Array* model allows to encode any given pointer program into a program without pointer.

Example 5.2.3 *Example 5.2.1 for incrementation of elements of a linked list can be encoded in our pointer-free language as follows.*

```
val data: ref (map loc int)
val next: ref (map loc loc)

function incr_list(l:loc)
  body
  let r = ref l in
  while p != null do
   upd(data,p,acc(data,p)+1);
   p := acc(next,p)
 done
```

This technique is implemented in verification tools that deal with mainstream languages like C and Java. Caduceus [9] and the Jessie plugin [14] of Frama-C [12] can generate verification conditions for C code annotated with ACSL [3]. Krakatoa [13] does the same for Java source code annotated with a variant of JML. The former tools use the component-as-array method to encode C or Java source into Why [10] or Why3 [4]. Other tools proceed in a similar way, by encoding into another intermediate language and VC generator called Boogie [2]: Spec# [1] to verify C# programs, VCC [7] for C code, etc.

5.2.4 Fundamental Example: In-place List Reversal

A classical example of a function modifying a linked list in-place is the list reversal. Indeed, Bornat [5] emphasizes that any proof method which claims to support pointer program should first demonstrate how it handles this example.

Here is a version of the code in our language with pointers.

```
function reverse(l:list) : list
body
  let p = ref l in let r = ref null in
  while p ≠ null do
    let n = p→next in
    p→next := r;
    r := p;
    p := n
  done;
  r
```

The same program, encoded in our pointer-free language using the Component-as-Array model is

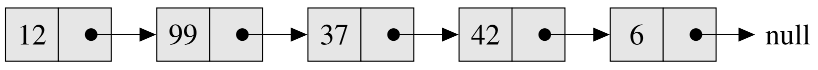
```
function reverse (l:loc) : loc
body
  let p = ref l in let r = ref null in
  while p ≠ null do
    let n = acc(next,p) in
    upd(next,p,r);
    r := p;
    p := n
  done;
  r
```

Our goal in the following is to show how one can specify the expected behavior of this code, and then how we can prove it. This example will illustrate the major issue regarding the proof of pointer programs, that is the property of *disjointness*, also called *separation*, of linked data structures.

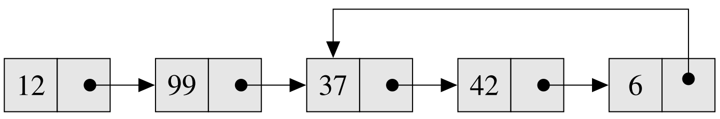
Specifying List Reversal

We want to specify the expected behavior of the reversal in the form of a contract. The first thing to specify in the precondition is that the input list should be well-formed, in the sense that it is terminated by the null pointer. In fact, thanks to our hypothesis that all memory accesses are safe, there are three possibilities for a shape of a linked list:

- the list can be null terminated, e.g.:



- the list can be cyclic, e.g.:



- the list can be infinite.

The third possibility cannot be ignored since we have no hypothesis on the finiteness of memory. However, there is no way to construct such a list with a program. But the second case is clearly possible, e.g by doing

```
p→ next := p
```

Thus, we want to specify that our input list has the first kind of the shapes above. The property of being terminated by null is typically a case where an inductive definition is useful. We can specify that a location l points to a null terminated list if and only if l is null, or l is not null and $l \rightarrow next$ points to a null-terminated list.

Specifying the shape of the list is not enough. If we want to express that the list l is reversed by the program, we need to be able to formally talk about the sequence of memory cells that occurs in the path from l to null. This idea leads to the definition of a predicate $\texttt{list\_seg}(p, next, p_M, q)$ meaning that p points to a list of nodes p_M that ends at q:

$$p = p_0 \overset{next}{\mapsto} p_1 \cdots \overset{next}{\mapsto} p_k \overset{next}{\mapsto} q$$

where

$$p_M = Cons(p_0, Cons(p_1, \cdots Cons(p_k, Nil) \cdots))$$

The pure list p_M is typically called the *model list* of p. The predicate is defined inductively by

```
inductive list_seg(loc,map loc loc,list loc,loc) =
  | list_seg_nil: forall p:loc, next:map loc loc. list_seg(p,next,Nil,p)
  | list_seg_cons: forall p q:loc, next:map loc loc, pM:list loc.
        p ≠ null ∧ list_seg(select(next,p),next,pM,q) → list_seg(p,next,Cons(p,pM),q)
```

The formal specification of list reversal can thus be given as follows.

- The precondition should state that the input list l is null-terminated, which corresponds to the formula

$$\exists l_M.\ \texttt{list\_seg}(l, next, l_M, null)$$

- The postcondition should state that the output is also null-terminated:

$$\exists r_M.\ \texttt{list\_seg}(result, next, r_M, null)$$

  but also that the model list of the result is the reverse (in the sense of pure logic lists as defined in Section 4.3.5) of the model list of the input:

$$r_M = rev(l_M)$$

The formal contract for reverse cannot be written exactly like that, since the list l_M is quantified in the precondition, and thus is not visible in the post-condition. A solution is to use the so-called *ghost* variables, which are extra variables in the program added for specification purposes. For our example of list reversal, this corresponds to passing the model list as an extra parameter, as follows.

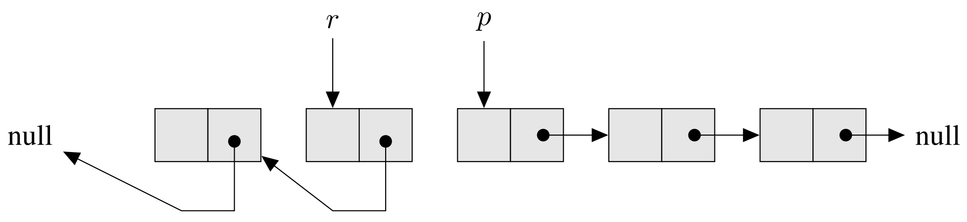
```
function reverse (l:loc,lM:list loc) : loc =
  requires list_seg(l,next,lM,null)
  writes next
  ensures list_seg(result,next,rev(lM),null)
  body ...
```

```
function reverse (l:loc, lM:list loc) : loc =
  requires list_seg(l,next,lM,null)
  writes next
  ensures list_seg(result,next,rev(lM),null)
  body
   let p = ref l in
   let pM = ref lM in
   let r = ref null in
   let rM = ref Nil in
   while (p ≠ null) do
     invariant list_seg(p,next,pM,null) ∧
               list_seg(r,next,rM,null) ∧
               append(rev(pM), rM) = rev(lM)
     let n = acc(next,p) in
     store(next,p,r);
     r := p;
     p := n;
     rM := Cons(head(!pM),!rM);
     pM := tail(!pM)
  done;
  r
```

Figure 5.1: List reversal annotated with ghost variables

In-place list reversal: loop invariant

In order to design an appropriate loop invariant, we use local ghost variables p_M, r_M that represent the model lists of p and r respectively. The proposed extended code is now as given in Figure 5.1. The first two parts of the loop invariant state that p and r point to null terminated lists. The third part `append(rev(pM), rM) = rev(lM)` formalizes the structure of the linke list and some iteration of the loop which is illustrated by the following picture.



Proving list reversal

With the annotated code of Figure 5.1, the proof cannot be performed easily, in the sense that automated provers will fail to prove the verification conditions generated by WP. More precisely, the loop invariant cannot be proved preserved by loop iterations. The reason is that it is not obvious that the operation

```
next := store(next,p,r);
```

preserves the property `list_seg(r,next,rM,null)` expressing the list structure of `r`. The formula to prove is as follows

```
list_seg(r,next,rM,null) ∧ next' = store(next,p,r) → list_seg(r,next',rM,null)
```

To prove such a preservation, we should say that the memory location that is modified, that is `p→next`, does not appear anywhere in the list `r`. A way to express that is to say that $\mathsf{p} \notin \mathsf{rM}$. This fact can be made explicit by posing the following lemma.

```
lemma list_seg_frame:
forall next1 next2:map loc loc, p q v: loc, pM:list loc.
  list_seg(p,next1,pM,null) ∧ next2 = store(next1,q,v) ∧
  ¬ mem(q,pM) → list_seg(p,next2,pM,null)
```

where the predicate `mem` is defined as

```
predicate mem (x:α,l:list α) =
  match l with
  | Nil → false
  | Cons(y,r) → x=y ∨ mem(x,r)
  end
```

Such a lemma is typically called a *frame* lemma, because it states what is the frame of the predicate `list_seg`, that is, the part of the memory on which the predicate depends. This lemma can be proved by induction on the length of the model list pM. Nevertheless, posing this lemma is not yet enough to prove the preservation of `list_seg(r,next,rM,null)`: the premise `¬ mem(q,pM)` of the lemma should be true, for the case where $q = p$ and pM is rM. Hence, we should be able to prove that p does not belong to the model list of r. To prove that, we need to strengthen the loop invariant by stating that the model lists pM and rM are always disjoint:

```
invariant list_seg(p,next,pM,null) ∧ list_seg(r,next,rM,null) ∧
              append(rev(pM), rM) = rev(lM) ∧ disjoint(pM,rM)
```

where the predicate `disjoint` is defined as

```
predicate disjoint (l1:list α,l2:list α) = forall x:α. ¬ (mem(x,l1) ∧ mem(x,l2))
```

With the strengthened loop invariant and the frame lemma, the proof that `list_seg(r,next,rM,null)` is preserved by a loop iteration can be made automatically. However, it is not enough to prove that `disjoint(pM,rM)` and `list_seg(p,next,pM,null)` are preserved. To prove this preservation, the frame lemma should be applied again, but in order to establish the premise `¬ mem(q,pM)` of this lemma, in the case when q is p and pM is the tail of the previous value of pM. That is, we should prove that the head of the model list does not appear in its tail. This property exactly corresponds to the fact that the list is not cyclic. In other words, we should make explicit another property of the list segment predicate, which is that a model list does not contain any repetition. We state this fact as another general lemma as follows.

```
lemma list_seg_no_repet:
  forall next:map loc loc, p: loc, pM:list loc.
    list_seg(p,next,pM,null) → no_repet(pM)
```

where

```
predicate no_repet (l:list α) =
  match l with
  | Nil → true
  | Cons(x,r) → ¬ (mem(x,r)) ∧ no_repet(r)
  end
```

Thanks to this second lemma. The proof of list reversal can be done using automated provers. The second lemma itself must be proved using induction.

Lessons learned from the list reversal example

The proof of this example emphasizes the major role of the property of disjointness of data structures when proving pointer programs. A lemma like the frame lemma above is needed to make explicit on which part of the memory a predicate on pointer data structures depends on. These are the motivations for introducing a dedicated logic called *Separation Logic*, detailed in the next section.

5.3 Exercises

Exercise 5.3.1 *Incrementations*

- *Specify and prove a function 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 function 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 function that replaces the reference argument by its reverse.*

```
function 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?*

Exercise 5.3.3 *The goal of this exercise is to specify and prove the function in the beginning of Section 5.2, which increments by one each element of a null-terminated linked list of integers.*

1. *Using the Component-as-Array version given in Example 5.2.1, provide a contract for this function*
2. *Find an appropriate loop invariant and prove the program using* WP

Exercise 5.3.4 *The following function appends two lists and returns a pointer to the resulting list.*

```
function append(l1:list,l2:list) : list
body
  if l1=null then l2 else
  let p = ref l1 in
  while p→next ≠ null do
    p := p→next;
  done;
  p→next := l2;
  l1
```

We assume that the two inputs are null-terminated lists, and are disjoint.

1. *Encode this program using the Component-as-Array model.*
2. *Specify it.*
3. *Prove its partial correctness using* WP*, using an appropriate loop invariant.*

Bibliography

[1] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# Programming System: An Overview. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS'04)*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer, 2004.

[2] M. Barnett, R. DeLine, B. Jacobs, B.-Y. E. Chang, and K. R. M. Leino. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *Formal Methods for Components and Objects: 4th International Symposium*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387, 2005.

[3] P. Baudin, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto. *ACSL: ANSI/ISO C Specification Language, version 1.4*, 2009. URL `http://frama-c.cea.fr/acsl.html`. `http://frama-c.cea.fr/acsl.html`.

[4] 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`.

[5] R. Bornat. Proving pointer programs in Hoare logic. In *Mathematics of Program Construction*, pages 102–126, 2000.

[6] R. Burstall. Some techniques for proving correctness of programs which alter data structures. *Machine Intelligence*, 7:23–50, 1972.

[7] M. Dahlweid, M. Moskal, T. Santen, S. Tobies, and W. Schulte. VCC: Contract-based modular verification of concurrent C. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Companion Volume*, pages 429–430. IEEE Comp. Soc. Press, 2009. ISBN 978-1-4244-3494-7.

[8] 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`.

[9] J.-C. Filliâtre and C. Marché. Multi-prover verification of C programs. In J. Davies, W. Schulte, and M. Barnett, editors, *6th International Conference on Formal Engineering Methods*, volume 3308 of *Lecture Notes in Computer Science*, pages 15–29, Seattle, WA, USA, Nov. 2004. Springer. URL `http://www.lri.fr/~filliatr/ftp/publis/caduceus.ps.gz`.

[10] J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In W. Damm and H. Hermanns, editors, *19th International Conference on Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 173–177, Berlin, Germany, July 2007. Springer. URL `http://www.lri.fr/~filliatr/ftp/publis/cav07.pdf`.

[11] J.-C. Filliâtre and A. Paskevich. Why3 — where programs meet provers. In *Proceedings of the 22nd European Symposium on Programming*, volume 7792 of *Lecture Notes in Computer Science*. Springer, Mar. 2013.

[12] Frama-C. The Frama-C platform for static analysis of C programs, 2008. `http://www.frama-c.cea.fr/`.

[13] C. Marché, C. Paulin-Mohring, and X. Urbain. The KRAKATOA tool for certification of JAVA/-JAVACARD programs annotated in JML. *Journal of Logic and Algebraic Programming*, 58(1–2): 89–106, 2004. URL `http://krakatoa.lri.fr`. `http://krakatoa.lri.fr`.

[14] Y. Moy. *Automatic Modular Static Safety Checking for C Programs*. PhD thesis, Université Paris-Sud, Jan. 2009. URL `http://www.lri.fr/~marche/moy09phd.pdf`.