Chapter 6

Pointer Programs, Separation Logic

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.

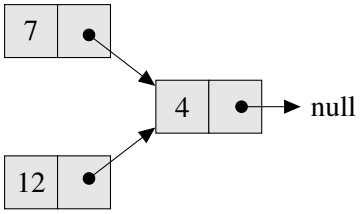
There is a fundamental difference of semantics between in-place modification of a record field and the way we modeled records in Chapter 4. The small piece of code below, in the syntax of the C programming language, is a typical example of the kind of programs we want to deal with in this chapter.

```
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.

6.1 Pointer Programs, Operational Semantics

In this first section, we extend our language with pointers to records, and formalize the operational semantics. For the moment we do not consider allocation and deallocation. We indeed completely ignore issues related to safety of memory access, so we assume that a field access and field update can always be done without error. Memory safety and allocation will be considered in Section 6.3.

6.1.1 Syntax

We use the same language of expressions with side effects as in the previous chapter, 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:

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

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.

e	$::=$	`null`	null pointer
		$\mid e \to f$	field access
		$\mid e \to f := e$	field update

Example 6.1.1 *The small example given at the beginning of this chapter, for incrementing a linked list, is written in our language as follows*

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

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

6.1.2 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" (this is our assumption that all memory accesses are safe) and that allocation/deallocation is not supported.

Example 6.1.2 *A program state that corresponds to the informal example with lists from the introduction of this chapter is*

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

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.

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

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

6.2 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 chapter. 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 [8] and was resurrected by Bornat in 2000 [7]. 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.

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

Example 6.2.1 *Example 6.1.1 for incrementation of elements of a linked list can be encoded in our pointer-free language as follows.*

type loc
function null : loc
val data: **ref** (map loc int)
val next: **ref** (map loc loc)

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

This technique is implemented in verification tools that deal with mainstream languages like C and Java. Caduceus [11] and the Jessie plugin [15] of Frama-C [13] can generate verification conditions for C code annotated with ACSL [3]. Krakatoa [14] 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 [12] or Why3 [6]. 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 [9] for C code, etc.

6.2.1 In-place List Reversal

A classical example of a procedure modifying a linked list in-place is the list reversal. Indeed, Bornat [7] 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

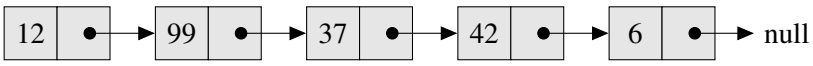
```
function reverse (l:loc) : loc
body
 let p = ref l in let r = ref null in
 while p ≠ null do
  let n = select(next,p) in next := store(next,p,r); r := p;  p := n
 done;
 r
```

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.

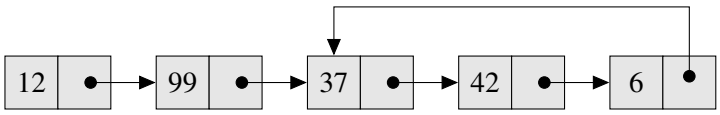
Specifying List Reversal

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:

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

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:

$$r_M = rev(l_M)$$

```
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 = select(next,p) in
    next := store(next,p,r);
    r := p;
    p := n;
    rM := Cons(head(!pM),!rM);
    pM := tail(!pM)
 done;
 r
```

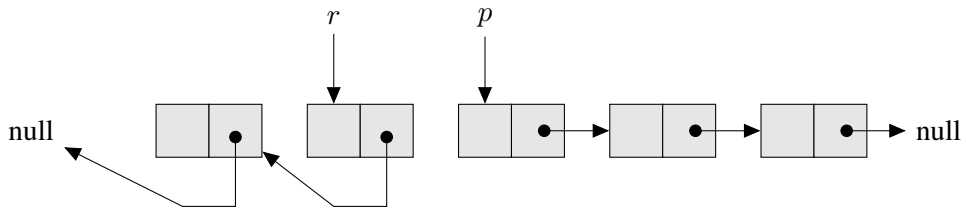
Figure 6.1: List reversal annotated with ghost variables

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 ...
```

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 6.1. 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 6.1, 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) $\wedge$ next' = store(next,p,r) $\rightarrow$ list_seg(r,next',rM,null)

To prove such a preservation, we should say that the memory location that is modified, that is p$\rightarrow$next, does not appear anywhere in the list r. A way to express that is to say that p $\notin$ 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) $\wedge$ next2 = store(next1,q,v) $\wedge$
$\neg$ mem(q,pM) $\rightarrow$ list_seg(p,next2,pM,null)

where the predicate mem is defined as

predicate mem (x:α,l:list α) =
match l **with**
| Nil $\rightarrow$ **false**
| Cons(y,r) $\rightarrow$ x=y $\vee$ 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 $\neg$ 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) $\wedge$ list_seg(r,next,rM,null) $\wedge$
append(rev(pM), rM) = rev(lM) $\wedge$ disjoint(pM,rM)

where the predicate disjoint is defined as

predicate disjoint (l1:list α,l2:list α) = **forall** x:α. $\neg$ (mem(x,l1) $\wedge$ 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 $\neg$ 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) $\rightarrow$ 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.

6.3 Separation Logic

Separation logic was introduced around year 2000 by Reynolds and O'Hearn [17]. There are indeed several variants of this logic [4, 19]. It is implemented in several tools like Smallfoot [5] and Verifast [18], that use specific, partially automated provers as back-ends, or Ynot [16] that uses Coq to perform the proofs.

6.3.1 Syntax of Programs

We do not consider any encoding of pointers and memory heap like the Component-as-Array model, so we directly consider the language introduced in Section 6.1.1. We also generalize this language in order to deal with memory allocation and deallocation. The important point is how to extend the language of formulas in specifications, which is detailed in Section 6.3.3

The syntax of expressions is now as follows.

e	::=	...	former expression constructs
		$\mid e \to f$	field access
		$\mid e \to f := e$	field update
		$\mid \texttt{new}\ S$	allocation
		$\mid \texttt{dispose}\ e$	deallocation

The expression `new` S allocates a new record of type S, and returns a pointer to it. The expression `dispose` e deallocates the pointer denoted by expression e, and returns $()$.

6.3.2 Operational Semantics

To support allocation and deallocation, we modify the operational semantics given in Section 6.1.2. Instead of considering the memory heap as a *total* map from pairs (loc,field name) to values, we consider that a heap is any *partial* map from (loc,field name) to values. We need to introduce a few notations: if h, h_1, h_2 are such partial maps:

- $\mathsf{dom}(h)$ denotes the *domain* of h, i.e. the set of pairs (loc,field name) where it is defined ;
- we write $h = h_1 \oplus h_2$ to mean that h is the disjoint union of h_1 and h_2, i.e.

- $\mathsf{dom}(h_1) \cap \mathsf{dom}(h_2) = \emptyset$
- $\mathsf{dom}(h) = \mathsf{dom}(h_1) \cup \mathsf{dom}(h_2)$
- $h(l, f) = h_1(l, f)$ if $(l, f) \in \mathsf{dom}(h_1)$
- $h(l, f) = h_2(l, f)$ if $(l, f) \in \mathsf{dom}(h_2)$

Note that in the following, the domain of partial heaps are always finite.

The operational semantics is defined by the relation

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

where h is a partial heap. The rules are as follows.

Field access

$$\frac{l, f \in \mathsf{dom}(h) \qquad h(l, f) = v}{h, \Sigma, \Pi, (l \to f) \rightsquigarrow h, \Sigma, \Pi, v}$$

Implicitly, this rule means that if $(l, f) \notin \mathsf{dom}(h)$ then the execution cannot be done, it "blocks". Typically in practice this corresponds to an invalid memory access, which usually stops the program execution with a message like "segmentation fault" or "memory fault"

Field update

$$\frac{(l, f) \in \mathsf{dom}(h) \qquad h' = h\{(l, f) \leftarrow v\}}{h, \Sigma, \Pi, (l \to f := v) \rightsquigarrow h', \Sigma, \Pi, ()}$$

Again, if $(l, f) \notin \mathsf{dom}(h)$ then the execution blocks.

Allocation

$$\frac{l \notin \mathsf{dom}(h) \qquad h' = h \oplus \{(l, f) \leftarrow \mathit{default}(\tau) \mid f : \tau \in S\}}{h, \Sigma, \Pi, (\texttt{new}\ S) \rightsquigarrow h', \Sigma, \Pi, l}$$

The premise $l \notin \mathsf{dom}(h)$ means that l is "fresh", i.e. it is not yet allocated in h. The other premise expresses the initialization of the allocated fields depending on their types, i.e. $\mathit{default}(\mathit{int}) = 0$, $\mathit{default}(\mathit{bool}) = \mathit{false}$, $\mathit{default}(\mathit{real}) = 0.0$, $\mathit{default}(S) = \mathit{null}$.

Notice that from this definition, the allocation never blocks, our formalization does not consider memory overflow issues.

Deallocation

$$\frac{\text{for all } f \text{ of } S,\ (l, f) \in \mathsf{dom}(h) \qquad h' = h \backslash l}{h, \Sigma, \Pi, (\texttt{dispose}\ l) \rightsquigarrow h', \Sigma, \Pi, ()}$$

where the notation $h' = h \backslash l$ means $h'(l', f') = h(l', f')$ if $l' \neq l$, undefined otherwise. Notice that deallocation blocks if one attempts to deallocate a memory location that is not allocated.

Example 6.3.1 *The small program below illustrates our operational semantics.*

record List = { data : int, next: List }

$[], [], []$
let x = new List **in** $\rightsquigarrow$
$[(l_0, \mathit{data}) = 0, (l_0, \mathit{next}) = \mathit{null}], [], [x = l_0]$

x→next := new List; $\rightsquigarrow$
$[(l_0, data) = 0, (l_0, next) = l_1, (l_1, data) = 0,$
$\quad (l_1, next) = null], [], [x = l_0]$
dispose(x→next); $\rightsquigarrow$
$[(l_0, data) = 0, (l_0, next) = l_1], [], [x = l_0]$
x→next→data := 1
execution blocks

6.3.3 Separation Logic Formulas

A important idea in Separation Logic is that the language of terms remains *unchanged*, in particular the expression $e \to f$ *is not a term*. The language of formulas is extended with

- special atoms specifying *available memory ressources*
- a special connective called *separating* conjunction

The new grammar for formulas is as follows.

$$\begin{array}{rcll} P, Q & ::= & \ldots & \text{former formulas} \\ & & \mid \mathsf{emp} & \text{empty heap} \\ & & \mid t_1 \xrightarrow{f} t_2 & \text{memory chunk} \\ & & \mid P * Q & \text{separating conjunction} \end{array}$$

where

- t_1 is a term of type S for some record type S
- f is a field of type τ in S
- t_2 is a term of type τ

These three new constructs allow to describe *finite portions of the memory heap*

The semantics of formulas is defined as usual by an interpretation $[\![P]\!]_{h,\Sigma,\Pi}$ that now depends on a partial heap h.

- "standard" formula: same semantics as before

$$[\![P]\!]_{h,\Sigma,\Pi} = [\![P]\!]_{\Sigma,\Pi}$$

- Special formula emp:

$$[\![\mathsf{emp}]\!]_{h,\Sigma,\Pi} \text{ valid iff } \mathsf{dom}(h) = \emptyset$$

- Memory chunk: $[\![t_1 \xrightarrow{f} t_2]\!]_{h,\Sigma,\Pi}$ iff
  - $[\![t_1]\!]_{\Sigma,\Pi} = l$ for some location l
  - $\mathsf{dom}(h) = \{(l, f)\}$
  - $h(l, f) = [\![t_2]\!]_{\Sigma,\Pi}$
- Separating conjunction: $[\![P * Q]\!]_{h,\Sigma,\Pi}$ is valid iff
  there exists h_1, h_2 such that

– $h = h_1 \oplus h_2$
– $[\![P]\!]_{h_1,\Sigma,\Pi}$ is valid
– $[\![Q]\!]_{h_2,\Sigma,\Pi}$ is valid

The semantics of the separating conjunction explicitly requires that the two conjuncts are valid on disjoint heaps. This means that predicates like our former predicates disjoint and no_repet are in some sense *internalized* in the logic.

Example 6.3.2 *Assuming two non-null locations l_0 and l_1, the stack $\Pi = [x = l_0]$ and the following partial heaps*

$$\begin{aligned} h_1 &= [(l_0, next) = l_1] \\ h_2 &= [(l_0, next) = l_1, (l_0, data) = 42] \\ h_3 &= [(l_0, next) = l_1, (l_1, next) = null] \end{aligned}$$

the table below gives the validity of a few formulas (Y for valid, N otherwise).

valid in ?	h_1	h_2	h_3
$x \overset{next}{\to} l_1$	Y	N	N
$x \overset{next}{\to} l_1 * x \overset{data}{\to} 42$	N	Y	N
$x \overset{next}{\to} l_1 * l_1 \overset{next}{\to} null$	N	N	Y
emp	N	N	N
$l_0 \neq \texttt{null}$	Y	Y	Y
$x \overset{next}{\to} l_1 * true$	Y	Y	Y

Properties of Separating Conjunction Separating conjunction satisfies the following identities

- $(P * Q) * R \leftrightarrow P * (Q * R)$
- $P * Q \leftrightarrow Q * P$
- $\mathsf{emp} * P \leftrightarrow P$
- if P, Q are memory-free formulas, $P * Q \leftrightarrow P \wedge Q$

In general P not equivalent to $P * P$. In fact this identity is true only if P is emp or a memory-free formula. This is a *linearity* aspect of the separating conjunction.

6.3.4 Separation Logic Reasoning Rules

We can define rules for reasoning on programs in the similar style as Hoare rules, on triples $\{P\}e\{Q\}$ where P and Q are separation logic formulas.

The following rules can easily be proved correct.

Field access

$$\frac{}{\{l \overset{f}{\to} v\}l \to f\{l \overset{f}{\to} v * \mathsf{result} = v\}}$$

Field update

$$\frac{}{\{l \overset{f}{\to} v\}l \to f := v'\{l \overset{f}{\to} v' * \mathsf{result} = ()\}}$$

Allocation

$$\frac{}{\{\mathsf{emp}\}\texttt{new}\ S\{\exists l, l \overset{f_1}{\to} default(\tau_1) * \cdots * l \overset{f_n}{\to} default(\tau_n)\}}$$

where S is declared as $\{f_1 : \tau_1; \cdots f_n : \tau_n\}$

Deallocation

$$\frac{}{\{l \overset{f_1}{\to} v_1 * \cdots * l \overset{f_n}{\to} v_n\}\texttt{dispose}\ l\{\mathsf{emp}\}}$$

where l has type S.

Frame rule A very important rule that can be stated is the frame rule, which allows to reason *locally*:

$$\frac{\{P\}e\{Q\}}{\{P * R\}e\{Q * R\}}$$

in particular it states that whenever $\{P\}e\{Q\}$ is proven valid, its effects are confined in the partial heaps that are described by P and Q, and consequently it remains valid in any context R which is disjoint from P and Q.

Separation Logic and Symbolic Execution It has been noticed that reasoning with Separation Logic rules is a kind of *symbolic execution* [4]. Thanks to the frame rule, a proof of a program made of a sequence $e_1; \cdots; e_n$ can be presented under the form

$$\{P_0\}e_1\{P_1\}e_2 \cdots e_n\{P_n\}$$

where at each step i we have $P_{i-1} = Q * R$, $P_i = Q' * R$ and $\{Q\}e_i\{Q'\}$ is valid for a rule above.

Example 6.3.3 *Here is a simple example of Separation Logic proof in the form of symbolic execution.*

$\{emp\}$
x := new List ;
$\{\exists l, (x = l) * (l \overset{data}{\to} 0) * (l \overset{next}{\to} null)\}$
$\{(x \overset{data}{\to} 0) * (x \overset{next}{\to} null)\}$ *(consequence)*
x→data := 42 ;
$\{(x \overset{data}{\to} 42) * (x \overset{next}{\to} null)\}$ *(frame)*
x→next := new List ;
$\{\exists l, (x \overset{data}{\to} 42) * (x \overset{next}{\to} l) * (l \overset{data}{\to} 0) * (l \overset{next}{\to} null)\}$

6.3.5 Case of Linked Lists

Similarly as the case of Component-as-Array model, we can define linked data structures using inductive predicates. For linked list, we can define

inductive $ls(List, List)$ =
| ls_nil: $\forall x : List, ls(x, x)$
| ls_cons: $\forall xyz : List, (x \overset{next}{\to} y) * ls(y, z) \to ls(x, z)$

The important novelty is the use of a separating conjunction in the second case, which makes explicit that the tail of the list is disjoint from its head, and thus it has no repetition.

During a Separation Logic proof, one should apply purely logic reasoning steps using general lemmas like

$$ls(x, y) \leftrightarrow x = y \vee \exists z, (x \overset{next}{\to} z) * ls(z, y)$$

Example: in-place list reversal

We consider again the example of in-place linked list reversal. We simplicity, we do not specify that the result list is the reverse of the input, but only that the output is a null-terminated list if the input is such a list. This avoids the need for ghost variables. The specified program we want to prove is thus the following

function reverse (l:loc) : loc =
 requires ls(l,null)
 writes next
 ensures ls(**result**,null)
 body
  let p = **ref** l **in**
  let r = **ref** null **in**
  while p $\neq$ null **do**
   invariant ls(p,null) $*$ ls(r,null)
   let n = p$\rightarrow$next **in**
   p$\rightarrow$next := r;
   r := p;
   p := n;
 done;
 r

Notice that the loop invariant tells that lists p and r are null-terminated and disjoint.

The first part of the proof is to show by symbolic execution that the loop invariant is initially true. This is as follows.

$\{ls(l, null)\}$
let p = **ref** l **in**
$\{(p = l) * ls(l, null)\}$
let r = **ref** null **in**
$\{(r = null) * (p = l) * ls(l, null)\}$
$\{ls(r, null) * ls(p, null)\}$ (consequence)
while p $\neq$ null **do**
 invariant ls(p,null) $*$ ls(r,null)

The second part is to show the loop invariant is preserved by a loop iteration.

while p $\neq$ null **do**
 invariant ls(p,null) $*$ ls(r,null)
 $\{(p \neq null) * ls(r, null) * ls(p, null)\}$
 $\{\exists q, ls(r, null) * (p \overset{next}{\rightarrow} q) * ls(q, null)\}$
 let n = p$\rightarrow$next **in**
 $\{ls(r, null) * (p \overset{next}{\rightarrow} n) * ls(n, null)\}$
 p$\rightarrow$next := r;
 $\{ls(r, null) * (p \overset{next}{\rightarrow} r) * ls(n, null)\}$ (frame rule)
 $\{ls(p, null) * ls(n, null)\}$
 r := p;
 $\{(r = p) * ls(p, null) * ls(n, null)\}$
 $\{ls(r, null) * ls(n, null)\}$
 p := n;
 $\{(p = n) * ls(r, null) * ls(n, null)\}$

$\{ls(r, null) * ls(p, null)\}$

Finally, we prove that when we exit the loop, the post-condition is established.

while p $\neq$ null **do**
  invariant ls(r,null) $*$ ls (p,null)
  ...
done
$\{(p = null) * ls(r, null) * ls(p, null)\}$
$\{ls(r, null)\}$

Final Remarks on Separation Logic

The main motivation of Separation Logic is that it internalizes disjointness and frame properties. It makes reasoning on pointer data structures much easier than a model like the Component-as-Array, where disjointness and frame properties must be stated explicitly.

Separation Logic has several applications and extensions. An important application is to the specification of *data invariants*, that are properties that should remain true along the execution of a program (such as "this list should always contain non-negative integers", "this tree should always be balanced"). Separation Logic provides for free that a data invariant is preserved when memory is modify outside its frame. Separation Logic can also deals with *concurrent programs* [10], for example two threads executing in parallel on disjoint part of the memory can be proved easily.

A major drawback in Separation Logic techniques nowadays is their low level of automation. There no simple equivalent of weakest precondition calculus, and thus SMT solvers cannot be used directly, one has to design provers that can understand the separating conjunction.

6.4 Exercises

Exercise 6.4.1 *Prove a complete specification of linked list reversal via Separation Logic, using appropriate ghost variables and an extended predicate* ls *that includes the model list.*

Exercise 6.4.2 *The goal of this exercise is to specify and prove the procedure in the introduction of this chapter, which increments by one each element of a null-terminated linked list of integers.*

1. *Using the Component-as-Array version given in Example 6.1.1, provide a contract and a loop invariant, then prove the program using* WLP

2. *Using the original version given in introduction, provide a contract and a loop invariant using separation logic, then prove it using symbolic execution.*

Exercise 6.4.3 *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, specify it, and then prove its partial correctness using* WLP.

2. *Specify this program using Separation Logic and prove it.*

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] J. Berdine, C. Calcagno, and P. W. O'Hearn. Symbolic execution with separation logic. In K. Yi, editor, *APLAS*, volume 3780 of *Lecture Notes in Computer Science*, pages 52–68. Springer, 2005.

[5] J. Berdine, C. Calcagno, and P. W. O'hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *In International Symposium on Formal Methods for Components and Objects*, pages 115–137. Springer, 2005.

[6] 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*, Wrocław, Poland, August 2011. URL `http://proval.lri.fr/submissions/boogie11.pdf`.

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

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

[9] 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.

[10] X. Feng, R. Ferreira, and Z. Shao. On the relationship between concurrent separation logic and assume-guarantee reasoning. In R. D. Nicola, editor, *ESOP*, volume 4421 of *Lecture Notes in Computer Science*, pages 173–188. Springer, 2007.

[11] 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`.

[12] 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`.

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

[14] 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`.

[15] 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`.

[16] A. Nanevski, G. Morrisett, and L. Birkedal. Polymorphism and separation in Hoare type theory. In J. H. Reppy and J. L. Lawall, editors, *11th ACM SIGPLAN International Conference on Functional Programming, ICFP 2006*, pages 62–73, Portland, Oregon, USA, 2006. ACM Press. ISBN 1-59593-309-3.

[17] J. C. Reynolds. Separation logic: a logic for shared mutable data structures. In *17h Annual IEEE Symposium on Logic in Computer Science*. IEEE Comp. Soc. Press, 2002.

[18] J. Smans, B. Jacobs, F. Piessens, and W. Schulte. An automatic verifier for java-like programs based on dynamic frames. In *Fundamental Approaches to Software Engineering (FASE'08)*, Budapest, Hungary, Apr. 2008.

[19] H. Tuch, G. Klein, and M. Norrish. Types, bytes, and separation logic. In M. Hofmann and M. Felleisen, editors, *Proc. 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'07)*, pages 97–108, Nice, France, Jan. 2007.