Chapter 6
Separation Logic

Separation logic was introduced around year 2000 by Reynolds and O’Hearn [6]. There are indeed several variants of this logic [1, 8]. It is implemented in several tools like Smallfoot [2] and Verifast [7], that use specific, partially automated provers as back-ends, or Ynot [5] or CFML [3] that uses Coq to perform the proofs.

6.1 Basics of Separation Logic

6.1.1 Syntax of Programs

We do not consider any encoding of pointers and memory heap like the Component-as-Array model of Section 5.2.3, so we directly consider the language introduced in Section 5.2.1. But, to make the formalization of memory access uniform, we remove mutable variables from that language: every mutable data must now be of the form of a pointer to a record, and only record fields are mutable. This is indeed how the references in OCaml or in Why3 are introduced: these are not built-in in the language but defined using a record with one mutable field. We detail this in Section 6.2.

For declaring such records, we keep the same syntax as in previous chapter.

\[ \text{record } \text{id} = \{ f_1 : \tau_1; \cdots; f_n : \tau_n \} \]

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 6.1.3.

The syntax of expressions is now as follows.

\[ e ::= \ldots \text{ former expression constructs} \]

\[ e \rightarrow f \text{ field access} \]

\[ e \rightarrow f := e \text{ field update} \]

\[ \text{new } S \text{ allocation} \]

\[ \text{dispose } e \text{ deallocation} \]

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

6.1.2 Operational Semantics

To support allocation and deallocation, we modify the operational semantics given in Section 5.2.2. 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:

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

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

The operational semantics is defined by the relation 
\[
    h, \Pi, e \xrightarrow{} h', \Pi', e'
\]
where \( h \) is a partial heap. The rules are as follows.

**Field access**

\[
\begin{align*}
  l, f & \in \text{dom}(h) \quad h(l, f) = v \\
  h, \Pi, (l \to f) & \xrightarrow{} h, \Pi, v
\end{align*}
\]

Implicitly, this rule means that if \( (l, f) \notin \text{dom}(h) \) 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**

\[
\begin{align*}
  (l, f) & \in \text{dom}(h) \quad h' = h \{ (l, f) \leftarrow v \} \\
  h, \Pi, (l \to f := v) & \xrightarrow{} h', \Pi, ()
\end{align*}
\]

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

**Allocation**

\[
\begin{align*}
  l & \notin \text{dom}(h) \quad h' = h \oplus \{(l, f) \leftarrow \text{default}(\tau) \mid \tau \in S\} \\
  h, \Pi, (\text{new } S) & \xrightarrow{} h', \Pi, l
\end{align*}
\]

The premise \( l \notin \text{dom}(h) \) 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. \( \text{default}(\text{int}) = 0, \text{default}(\text{bool}) = \text{false}, \text{default}(\text{real}) = 0.0, \text{default}(S) = \text{null} \).

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

**Deallocation**

\[
\begin{align*}
  \text{for all } f \text{ of } S, (l, f) & \in \text{dom}(h) \quad h' = h \backslash l \\
  h, \Pi, (\text{dispose } l) & \xrightarrow{} h', \Pi, ()
\end{align*}
\]

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 6.1.1 The small program below illustrates our operational semantics.

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

[] → let x = new List in →
[(l_0, data) = 0, (l_0, next) = null], [x = l_0]
x → next := new List; →
[(l_0, data) = 0, (l_0, next) = l_1], [l_0, data) = 0, (l_1, next) = null], [x = l_0]
dispose(x → next); →
[(l_0, data) = 0, (l_0, next) = l_1], [x = l_0]
x → next → data := 1
```

6.1.3 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 resources
- a special connective called separating conjunction

The new grammar for formulas is as follows.

```plaintext
P, Q ::= ...
  | emp                empty heap
  | t_1 \to t_2        memory chunk
  | P \ast Q           separating conjunction
```

where

- \( t_1 \) is a term of type \( S \) for some record type \( S \)
- \( f \) is a field of type \( \tau \) in \( S \)
- \( t_2 \) is a term of type \( \tau \)

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

The semantics of formulas is defined as usual by an interpretation \( \llbracket P \rrbracket_{h,\Pi} \) that now depends on a partial heap \( h \).

- Special formula \( \mathsf{emp} \):
  \[
  \llbracket \mathsf{emp} \rrbracket_{h,\Pi} \text{ valid iff } \text{dom}(h) = \emptyset
  \]

- Memory chunk: \( \llbracket t_1 \to t_2 \rrbracket_{h,\Pi} \) iff
  
  - \( \llbracket t_1 \rrbracket_{\Pi} = l \) for some location \( l \)
  - \( \text{dom}(h) = \{(l, f)\} \)
  - \( h(l, f) = \llbracket t_2 \rrbracket_{\Pi} \)

- Separating conjunction: \( \llbracket P \ast Q \rrbracket_{h,\Pi} \) is valid iff there exists \( h_1, h_2 \) such that
\[-h = h_1 \oplus h_2\]
\[-\llbracket P \rrbracket_{h_1, \Pi} \text{ is valid}\]
\[-\llbracket Q \rrbracket_{h_2, \Pi} \text{ is valid}\]

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 6.1.2** Assuming two non-null locations \(l_0\) and \(l_1\), the stack \(\Pi = [x = l_0]\) and the following partial heaps

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

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

<table>
<thead>
<tr>
<th>valid in ?</th>
<th>(h_1)</th>
<th>(h_2)</th>
<th>(h_3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x \mapsto l_1)</td>
<td>(Y)</td>
<td>(N)</td>
<td>(N)</td>
</tr>
<tr>
<td>(x \mapsto l_1 \ast x \mapsto \text{data} \mapsto 42)</td>
<td>(N)</td>
<td>(Y)</td>
<td>(N)</td>
</tr>
<tr>
<td>(x \mapsto l_1 \ast l_1 \mapsto \text{null})</td>
<td>(N)</td>
<td>(Y)</td>
<td>(Y)</td>
</tr>
<tr>
<td>(\text{emp})</td>
<td>(N)</td>
<td>(N)</td>
<td>(N)</td>
</tr>
</tbody>
</table>

**Properties of Separating Conjunction**  Separating conjunction satisfies the following identities

- \((P \ast Q) \ast R \leftrightarrow P \ast (Q \ast R)\)
- \(P \ast Q \leftrightarrow Q \ast P\)
- \(\text{emp} \ast P \leftrightarrow P\)
- if \(P, Q\) are memory-free formulas, \(P \ast Q \leftrightarrow P \land Q\)

In general \(P\) not equivalent to \(P \ast P\). In fact this identity is true only if \(P\) is \(\text{emp}\) or a memory-free formula. This is a linearity aspect of the separating conjunction.

**Symbolic Heaps** The semantics given above has some issues regarding memory-free formulas. For example, using the heaps of Example 6.1.2 above, even if the formula \(x \mapsto l_1\) is valid only in heap \(h_1\), the conjunction \(x \mapsto l_1 \ast \text{true}\) is valid also in heap \(h_1\) and \(h_2\). A variant in the semantics would be to consider that a memory-free formula is valid only in an empty heap.

Going further, a classical fragment of separation logic is the so-called *Symbolic Heaps*. Only formulas of the form

\[
\exists v_1, \ldots, v_n, P_1 \ast P_2 \ast \cdots \ast P_k \ast \phi
\]

are considered, where \(\phi\) is a memory-free formula and the \(P_i\) are memory chunks.

In the rest of this chapter, we consider this fragment only, with the semantics that memory-free formulas are valid only in an empty heap.

**Example 6.1.3** The following function resets the field \(\ell\) of some structure to zero.
record $S \{ f : \text{int} \}$

function reset_f(x:S):unit
    requires $\exists v. x \mapsto \ell v$
    ensures $x \mapsto 0$
    body $x \mapsto f := 0$

Since one cannot directly use $x \mapsto f$ as a term, an extra quantification is needed to specify that $x \mapsto f$ is allocated.

The following function increments the field $f$ by one.

function incr_f(x:S) (ghost $v$:int):unit
    requires $x \mapsto f \mapsto v$
    ensures $x \mapsto f \mapsto v + 1$
    body $x \mapsto f := x \mapsto f + 1$

A ghost variable is needed to talk about the old value of $x \mapsto f$.

Some syntactic sugar are usually used: existentially quantified variables starting with an underscore, and quantification made implicit.

Example 6.1.4 The same functions as in previous example, with syntactic sugar

function reset_f(x:S):unit
    requires $x \mapsto f \mapsto _$
    ensures $x \mapsto f \mapsto 0$
    body $x \mapsto f := 0$

function incr_f(x:S) :unit
    requires $x \mapsto f \mapsto _v$
    ensures $x \mapsto f \mapsto _v + 1$
    body $x \mapsto f := x \mapsto f + 1$

Notice how the ghost variable is made implicit.

6.1.4 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 symbolic heaps.

The following rules can easily be proved correct.

Field access

\[
\{ l \mapsto v \} l \rightarrow f \{ l \mapsto v \ast \text{result} = v \}
\]

Field update

\[
\{ l \mapsto v \} l \rightarrow f := v' \{ l \mapsto v' \ast \text{result} = () \}
\]
Allocation

\[
\{\text{emp}\} \text{new } S \{l_1 \mapsto \text{default} (\tau_1) \ast \cdots \ast l_n \mapsto \text{default} (\tau_n) \ast \text{result} = \_l \}
\]

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

Deallocation

\[
\{l_1 \mapsto v_1 \ast \cdots \ast l_n \mapsto v_n \} \text{dispose } l \{\text{emp}\}
\]

where \( l \) has type \( S \).

Frame rule

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

\[
\frac{\{P\} e \{Q\}}{\{P \ast R\} e \{Q \ast R\}}
\]

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 [1]. 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 \( P_{i-1} = Q \ast R, P_i = Q' \ast R \) and \( \{Q\} e_i \{Q'\} \) is valid for a rule above.

Example 6.1.5 Here is a simple example of Separation Logic proof in the form of symbolic execution.

\[
\{\text{emp}\} \text{let } x = \text{new List in } \{x \mapsto \text{data} := 42\; ; \; x \mapsto \text{next} := \text{new List ; } \}
\]

6.2 Case of References à la OCaml

The mutable variables that we had in previous chapters but we dropped in our new language, can be simulated using a pointer to a record with one field only. For convenience it is made polymorphic.

\[
\text{record } \text{Ref } \alpha = \{ \text{contents} : \alpha \} \\
\text{function } \text{ref} \ (x : \alpha) : \text{Ref } \alpha \\
\text{body let } r = \text{new Ref in } r \mapsto \text{contents} := x; r
\]
function (!) (r: Ref α) : α
   body r→contents

function (:=) (r: Ref α) (x:α) : unit
   body r→contents := x

   These functions can be given natural specifications in separation logic. For readability we abbreviate
   \( r \oplus t \) into \( r \leftrightarrow t \)

function ref (x: α) : Ref α
   requires emp
   ensures result\( \leftrightarrow x \)

function (!) (r: Ref α) : α
   requires r\( \leftrightarrow \_v \)
   ensures r\( \leftrightarrow \_v \star \)result\( = \_v \)

function (:=) (r: Ref α) (x:α) : unit
   requires r\( \leftrightarrow \_ \)
   ensures r\( \leftrightarrow x \)

6.3 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: \( \forall xyz : List, (x \leftrightarrow y) \star ls(y, z) \rightarrow ls(x, z) \)

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

   \( ls(x, y) \leftrightarrow x = y \lor \exists z, (x \leftrightarrow z) \star ls(z, y) \)

From such lemmas one can derives the so-called symbolic execution rules, e.g.

   \( ls(x, y) \star x \neq y \rightarrow (x \leftrightarrow _z) \star ls(_z, y) \star x \neq y \)

   In practice, systems implementing separation logic use a specific reasoning engine that applies such
rules (they do not use off-the-shelf SMT provers).

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 (we use our “references” defined in previous section).
function reverse (l:List) : List =
  requires ls(l,null)
  ensures ls(result,null)
  body
    let p = ref l in
    let r = ref null in
    while !p ≠ null do
      invariant p→_lp * ls(_lp,null) * r→_lr * ls(_lr,null)
      let n = !p→next in
      !p→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
{p↦→l * ls(l, null) * r↦→null} (symbolic execution rule)
{p↦→l * ls(l, null) * r↦→null * ls(null, null)}
while p ≠ null do
  invariant p→_lp * ls(_lp,null) * r→_lr * ls(_lr,null)
  . . .
  done;
  {p↦→lp * ls(lp,null) * r↦→lr * ls(lr,null) * lp = null}

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

while !p ≠ null do
  invariant p→_lp * ls(_lp,null) * r→_lr * ls(_lr,null)
  {p↦→l * ls(l, null) * r↦→null} * rₓ→_q * ls(_q, null) * rₓ→lr * ls(lr, null) * lp ≠ null
  let n = !p→next in
  {p↦→lp * lpₓ→next * n * ls(n, null) * rₓ→lr * ls(lr, null) * lp ≠ null}
  !p→next := !r;
  {p↦→lp * lpₓ→next * lₓ→next * n * ls(n, null) * rₓ→lr * ls(lr, null) * lp ≠ null}
  r := !p;
  {p↦→lp * lpₓ→next * lₓ→next * lₓ→next * ls(n, null) * rₓ→lp * ls(lr, null) * lp ≠ null}
  p := n;
  {p↦→n * lpₓ→next * lₓ→next * ls(n, null) * rₓ→lp * ls(lr, null) * lp ≠ null}
  {p↦→n * ls(n, null) * rₓ→lp * ls(lp, null)}

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

while !p ≠ null do
  invariant p→_lp * ls(_lp,null) * r→_lr * ls(_lr,null)
  . . .
  done
  {p↦→lp * ls(lp,null) * r→→lr * ls(lr,null) * lp = null}
6.4 Going Further 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 [4], 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. Indeed, some analogue to WP can be constructed, if one considers also a connective for separating implication, also known as magic wand\textsuperscript{1}.

6.5 Exercises

Exercise 6.5.1 Prove a complete specification of linked list reversal via Separation Logic, using appropriate ghost variables and an extended predicate \( \text{ls} \) that includes the model list.

Exercise 6.5.2 Specify and prove the function of Exercise 5.3.3 which increments by one each element of a null-terminated linked list of integers, using Separation Logic.

Exercise 6.5.3 Specify and prove the function of Exercise 5.3.4 which appends two lists, using Separation Logic.

Bibliography


\textsuperscript{1}See http://www.lsv.ens-cachan.fr/~lozes/sl-lectures.php

