Separation Logic
Part 1

Arthur Charguéraud

February 7th, 2014
Content

Separation Logic: a first example
Separation Logic triples
Definition of heap predicates
The separation algebra
Structural rules
Reasoning rules for terms
Content

Separation Logic: a first example
  Example program
  Example Separation Logic triple

Separation Logic triples
Definition of heap predicates
The separation algebra
Structural rules
Reasoning rules for terms
Example program

Assume $s$ to be an existing reference. Consider the following program:

```
let n = !s in
s := 2*n;
let r = ref (3*n) in
(4*n, r)
```

Remark: OCaml syntax for reference is used:

- `ref v` to create a reference
- `!r` to read a reference
- `r := v` to update a reference
Example specification

```plaintext
let n = !s in
s := 3*n;
let r = ref (4*n) in
(2*n, r)
```

If there exists an integer \( n \) such that the initial state is described by:

\[ (s \leftarrow n) \]

then the program returns a pair \((a, r)\) such that the final state satisfies:

\[ [a = 2n] \land (s \leftarrow 3n) \land (r \leftarrow 4n) \]

→ this description asserts that \( s \) and \( r \) are disjoint locations
→ all the rest of memory is implicitly assumed to be unchanged
Example of a Separation Logic triple

Specification of the example program:

\[ \forall n. \quad \{ s \leftarrow n \} \text{ program } \{ \lambda(a, r). [a = 2n] \ast (s \leftarrow 3n) \ast (r \leftarrow 4n) \} \]

expressed using a Separation Logic triple of the form \( \{ H \} t \{ \lambda x. H' \} \)

- \( H \) describes the initial heap
- \( t \) is the term being specified
- \( x \) is a name for the output result
- \( H' \) describes the final heap and the output result
Remark: irrelevance of names

let n = !s in
s := 3*n;
let r = ref (4*n) in
(2*n, r)

We named logical variables like in the program:

$$\forall n. \ \{s \leftarrow n\} \ \text{program} \ \{\lambda(a, r). [a = 2n] \ast (s \leftarrow 3n) \ast (r \leftarrow 4n)\}$$

but we could have picked other names:

$$\forall m. \ \{s \leftarrow m\} \ \text{program} \ \{\lambda(a, p). [a = 2m] \ast (s \leftarrow 3m) \ast (p \leftarrow 4m)\}$$

→ Note, however, that there is a confusion between $s$ as a program variable appearing in the code and $s$ as a logical variable appearing in the specification. This confusion will be resolved later in the course.
Content

Separation Logic: a first example

Separation Logic triples
  Representation of heaps
  General form of triples

Definition of heap predicates

The separation algebra

Structural rules

Reasoning rules for terms
Representation of heaps

A heap $m$, of type Heap, is a finite map from location to values.

- $\emptyset$ the empty heap
- $\langle l \mapsto v \rangle$ a singleton heap
- $m[l]$ read in a heap
- $m[l \leftarrow v]$ update of a heap
- $m_1 \oplus m_2$ merge of two heaps

Merge is only defined for heaps with disjoint domains:

$$m_1 \perp m_2 \equiv \text{dom}(m_1) \cap \text{dom}(m_2) = \emptyset$$
General form of Separation Logic triples

A heap predicate $H$ characterizes memory stores of a particular shape. A heap predicate has type: \( \text{Heap} \rightarrow \text{Prop} \), abbreviated as Hprop.

A triple has the form \( \{H\} t \{\lambda x. H'\} \) where

- $H$ has type: \( \text{Heap} \rightarrow \text{Prop} \)
- $\lambda x. H'$ has type: \( A \rightarrow \text{Heap} \rightarrow \text{Prop} \), where $A$ is the type of $t$

More generally, a triple has the form \( \{H\} t \{Q\} \) where

- the pre-condition $H$ has type: Hprop
- the post-condition $Q$ has type: \( A \rightarrow \text{Hprop} \)
Towards an interpretation of triples

Assume in this slide that triples describe the entire state.

A triple \( \{H\} \ t \ \{\lambda x. H'\} \) is interpreted in total correctness as:

\[
\forall m. \ H m \Rightarrow \exists v. \exists m'. \ t/m \Downarrow v/m' \land (\langle v/x \rangle H') m'.
\]

Let \( Q = \lambda x. H' \). We have \( Q v = \langle v/x \rangle H' \).

So, a triple \( \{H\} \ t \ \{Q\} \) is interpreted as:

\[
\forall m. \ H m \Rightarrow \exists v. \exists m'. \ t/m \Downarrow v/m' \land Qv m'.
\]
Interpretation of Separation Logic triples

In Separation Logic, a triple describes only a part $m_1$ of the heap. The rest of the heap, call it $m_2$, is assumed to remain unchanged.

A triple $\{H\} t \{Q\}$ is thus interpreted as:

$$\forall m_1 \ m_2. \left\{ \begin{array}{l} H \ m_1 \\ m_1 \perp m_2 \end{array} \Rightarrow \exists v. \exists m_1'. \left\{ \begin{array}{l} t/m_1 \circ m_2 \Downarrow v/m_1' \circ m_2 \\ Q \ v \ m_1' \\ m_1' \perp m_2 \end{array} \right. \right\}$$
Content

Separation Logic: a first example
Separation Logic triples

Definition of heap predicates
  Overview of heap predicates
  Pure facts
  Pure facts
  Singleton heaps
  Separating conjunction
  Existential quantification

The separation algebra

Structural rules

Reasoning rules for terms
Primitive heap predicates

We start by defining the basic heap predicates:

- \([\ ]\) empty heap
- \([P]\) pure fact
- \(l \leftarrow v\) singleton heap
- \(H \ast H'\) separating conjunction
- \(\exists x. H\) existential quantification

Note that these predicates all have type \(\text{Hprop}\) (that is, \(\text{Heap} \rightarrow \text{Prop}\)).
Empty heap and pure facts

Definition:

\[
\begin{align*}
  [\ ] & \equiv \lambda m. \ m = \emptyset \\
  [P] & \equiv \lambda m. \ m = \emptyset \land P
\end{align*}
\]

Example: specification of the value 3:

\[\{[\ ]\} \ 3 \ \{\lambda x. [x = 3]\}\]
Specify the recursive factorial function:

```
let rec facto n =
    if n = 0 then 1 else n * facto(n - 1)
```

by filling in the dots below, using $n!$ to denote the mathematical factorial.

\[ \forall n. \{ \ldots \} (facto n) \{ \ldots \} \]

Solution:

\[ \forall n. \{ [n \geq 0] \} (facto n) \{ \lambda x. [x = n!] \} \]
Singleton heap

Definition:

\[ l \leftarrow v \equiv \lambda m. \ m = \langle l \leftarrow v \rangle \]

Exercise. complete the specification: \{...\} (ref 3) \{...\}.
Solution:

\{[ [ ] ] (ref 3) \{ \lambda r. \ r \leftarrow 3 \} \}

Exercise: specify "incr \ r", defined as "let \ x = !r in \ r := x+1".
Remark: \ tt is the unit value and "\ \lambda tt." is like "fun() \rightarrow" in Caml.
Solution:

\[ \forall r n. \ \{ r \leftarrow n \} (\text{incr } r) \{ \lambda tt. \ r \leftarrow n + 1 \} \]
Separation Logic’s star

The heap predicate $H_1 \ast H_2$ characterizes a heap made of two disjoint parts, one that satisfies $H_1$ and one that satisfies $H_2$.

Definition:

$$H_1 \ast H_2 \equiv \lambda m. \exists m_1 m_2. \begin{cases} m_1 \perp m_2 \\ m = m_1 \oplus m_2 \\ H_1 m_1 \\ H_2 m_2 \end{cases}$$

Example: $(r \leftarrow 3) \ast (s \leftarrow 4)$ describes two disjoint memory cells.
Exercise:

Specify the term “incr r; !s”. (You may assume r \neq s.)

Solution:

\[ \forall nm. \quad \{(r \leftrightarrow n) \ast (s \leftrightarrow m)\} \]
\[(incr \; r; \; !s)\]
\[\{\lambda x. [x = m] \ast (r \leftrightarrow n + 1) \ast (s \leftrightarrow m)\} \]
Separating conjunction and aliasing

Separation Logic supports reasoning about aliased references.

```plaintext
let r = ref 3 in
let s = ref 4 in
let t = id s in
incr s;
incr t;
!s
```

where the identity function has specification: \{[ ]\} (id a) \{\lambda x. [x = a]\}.
Towards Separation Logic derivations

Specification of “incr r”:

\[
\{(r \leftarrow n)\} \text{ (incr r) } \{\lambda tt. (r \leftarrow n + 1)\} \\
\{(s \leftarrow m) \ast (r \leftarrow n)\} \text{ (incr r) } \{\lambda tt. (r \leftarrow n + 1) \ast (s \leftarrow m)\}
\]

Specification of “!s”:

\[
\{(s \leftarrow m)\} \text{ (!s) } \{\lambda x. [x = m] \ast (s \leftarrow m)\} \\
\{(r \leftarrow n + 1) \ast (s \leftarrow m)\} \text{ (!s) } \{\lambda x. [x = m] \ast (s \leftarrow m) \ast (r \leftarrow n + 1)\}
\]

Specification of the sequence “incr r; !s”:

\[
\{(s \leftarrow m) \ast (r \leftarrow n)\} \text{ (incr r; !s) } \{\lambda x. [x = m] \ast (s \leftarrow m) \ast (r \leftarrow n + 1)\}
\]
A reference into another reference

Suppose we store the location of a reference into another reference:

\[
\text{let } r = \text{ref } 3 \text{ in ref } r
\]

What is the problem with the following specification?

\[
\text{[[[]] (let } r = \text{ref } 3 \text{ in ref } r) \{\lambda s. (s \rightarrow r) \ast (r \rightarrow 3)\}}
\]

The location \( r \) needs to be quantified existentially in the post-condition:

\[
\text{[[[]] (let } r = \text{ref } 3 \text{ in ref } r) \{\lambda s. \exists r. (s \rightarrow r) \ast (r \rightarrow 3)\}}
\]
Existential quantification

Definition of the existential quantifier for heap predicates:

$$\exists x. H \equiv \lambda m. \exists x. H m$$

Existentials are introduced by allocation but also by abstraction, which is the key ingredient to the construction of modular proofs.

For example, the precise specification:

$$\{ [[]] \} (\text{ref 6}) \{ \lambda r. (r \leftrightarrow 6) \}$$

can be weakened into:

$$\{ [[]] \} (\text{ref 6}) \{ \lambda r. \exists n. (r \leftrightarrow n) \ast [\text{even } n] \}.$$
Summary

Definition of the core Separation Logic heap predicates:

\[
\begin{align*}
\emptyset & \equiv \lambda m. m = \emptyset \\
 [P] & \equiv \lambda m. m = \emptyset \land P \\
l \xleftarrow{\diamond} v & \equiv \lambda m. m = \langle l \xleftarrow{\diamond} v \rangle \\
H_1 \ast H_2 & \equiv \lambda m. \exists m_1 m_2. \begin{cases} 
  m_1 \perp m_2 \\
  m = m_1 \cup m_2 \\
  H_1 m_1 \\
  H_2 m_2 
\end{cases} \\
\exists x. H & \equiv \lambda m. \exists x. H m
\end{align*}
\]
Content

Separation Logic: a first example
Separation Logic triples
Definition of heap predicates

The separation algebra
  Properties of separating conjunction
  Heap implication
  Properties of heap implication
  Reasoning about heap implications

Structural rules
Reasoning rules for terms
The separation algebra

The operator $\ast$ has type $\text{Hprop} \to \text{Hprop} \to \text{Hprop}$. This binary operation satisfies the following properties.

- **Associativity**: $(H_1 \ast H_2) \ast H_3 = H_1 \ast (H_2 \ast H_3)$
- **Commutativity**: $H_1 \ast H_2 = H_2 \ast H_1$
- **Neutral element**: $H \ast [] = H$
- **Scope extrusion**: $(\exists x. H_1) \ast H_2 = \exists x. (H_1 \ast H_2)$ (if $x \notin H_2$)

$\rightarrow$ Remark: these proofs rely on predicate extensionality

$$(\forall x. P x \leftrightarrow Q x) \Rightarrow (P = Q)$$
Definition of heap implication

We should be able to prove that:

\[
\begin{align*}
\text{if} & \quad \{\exists n. (r \leftarrow n) \ast [\text{even } n]\} \implies \{Q\} \\
\text{then} & \quad \{(r \leftarrow 6)\} \implies \{Q\}.
\end{align*}
\]

since the new pre-condition considered is stronger than the one required.

To formalize this, we introduce the heap implication relation:

\[ H_1 \triangleright H_2 \quad \equiv \quad \forall m. \; H_1 m \Rightarrow H_2 m \]

For example, we have \((r \leftarrow 6) \triangleright \exists n. (r \leftarrow n) \ast [\text{even } n]\).
Implication between post-conditions

We should be able to prove that:

\[
\begin{align*}
\text{if } & \{ H \} t \{ \lambda x. [ x = 4 ] \ast ( r \leftrightarrow 6 ) \} \\
\text{then } & \{ H \} t \{ \lambda x. [ \text{even } x ] \ast \exists n. ( r \leftrightarrow n ) \ast [ \text{even } n ] \}.
\end{align*}
\]

To formalize this, we introduce implication between post-conditions:

\[ Q_1 \triangleright Q_2 \equiv \forall x. Q_1 x \triangleright Q_2 x \]

Remark: \( Q_1 \triangleright Q_2 \) is equivalent to: \( \forall x. \forall m. Q_1 x m \Rightarrow Q_2 x m \).

Remark: \( ( \lambda tt. H_1 ) \triangleright ( \lambda tt. H_2 ) \) is equivalent to: \( H_1 \triangleright H_2 \).
Heap implication as a partial order

The heap implication relation \( \triangleright \) has type: \( \text{Hprop} \rightarrow \text{Hprop} \rightarrow \text{Prop} \).

\[
H_1 \triangleright H_2 \iff \forall m. H_1 m \Rightarrow H_2 m
\]

→ Our goal is to never unfold this definition in the proofs

This binary relation defines a partial order on the type \( \text{Hprop} \):

- **Reflexivity:**
  \[
  H \triangleright H
  \]

- **Transitivity:**
  \[
  H_1 \triangleright H_2, H_2 \triangleright H_3 \Rightarrow H_1 \triangleright H_3
  \]

- **Antisymmetry:**
  \[
  H_1 \triangleright H_2, H_2 \triangleright H_1 \Rightarrow H_1 = H_2
  \]

Moreover, heap implication is regular w.r.t. the star operator:

\[
H_1 \triangleright H_2, H_1' \triangleright H_2' \Rightarrow H_1 \star H_1' \triangleright H_2 \star H_2'
\]
Simplifications in heap implications

Regularity can be reformulated into a cancellation rule:

\[
\frac{H_1 \triangleright H'_1}{H_1 \ast H_2 \triangleright H'_1 \ast H_2}
\]

For example, to prove:

\[
(r \leftarrow 6) \ast [] \ast (s \leftarrow 3) \triangleright \exists n. (s \leftarrow 3) \ast (r \leftarrow n) \ast [\text{even } n]
\]

using the algebra laws and the cancellation rule, it suffices to prove:

\[
(r \leftarrow 6) \triangleright \exists n. (r \leftarrow n) \ast [\text{even } n]
\]
Heap implications: true or false?

1. \((r \leftarrow 3) \ast (s \leftarrow 4) \triangleright (s \leftarrow 4) \ast (r \leftarrow 3)\) true
2. \((r \leftarrow 3) \triangleright (s \leftarrow 4) \ast (r \leftarrow 3)\) false
3. \((s \leftarrow 4) \ast (r \leftarrow 3) \triangleright (r \leftarrow 4)\) false
4. \((s \leftarrow 4) \ast (r \leftarrow 3) \triangleright (r \leftarrow 3)\) false
5. \([\text{False}] \ast (r \leftarrow 3) \triangleright (s \leftarrow 4) \ast (r \leftarrow 4)\) true
6. \((r \leftarrow 4) \ast (s \leftarrow 3) \triangleright [\text{False}]\) false
7. \((r \leftarrow 4) \ast (r \leftarrow 3) \triangleright [\text{False}]\) true
8. \((r \leftarrow 3) \ast (r \leftarrow 3) \triangleright [\text{False}]\) true
9. \(\exists n. (r \leftarrow n) \triangleright (r \leftarrow 3)\) false
10. \(\exists n. (r \leftarrow n) \triangleright (r \leftarrow n)\) false
11. \((r \leftarrow 3) \triangleright \exists n. (r \leftarrow n)\) true
12. \((r \leftarrow 3) \ast (s \leftarrow 3) \triangleright \exists n. (r \leftarrow n) \ast (s \leftarrow n)\) true
13. \(\exists n. (r \leftarrow n) \ast [n > 0] \ast [n < 0] \triangleright [\text{False}]\) true
**Instantiation in RHS**

To prove \( (r \leftrightarrow 6) \implies (\exists n. (r \leftrightarrow n) \ast [\text{even } n]) \), it suffices to exhibit an even number \( n \) such that \( (r \leftrightarrow n) \).

**Corresponding reasoning rules:**

\[
\begin{align*}
H_1 & \implies (\langle v/x \rangle H_2) \\
H_1 & \implies (\exists x. H_2) \\
(H_1 \implies H_2) & \implies (H_2 \ast [P])
\end{align*}
\]

**Example:**

\[
\begin{align*}
(r \leftrightarrow 6) & \implies (r \leftrightarrow 6) \quad \text{even 6} \\
(r \leftrightarrow 6) & \implies (r \leftrightarrow 6) \ast [\text{even 6}] \\
(r \leftrightarrow 6) & \implies \langle 6/n \rangle ((r \leftrightarrow n) \ast [\text{even } n]) \\
(r \leftrightarrow 6) & \implies \exists n. (r \leftrightarrow n) \ast [\text{even } n]
\end{align*}
\]
To prove \((\exists n. [\text{even } n] \ast (r \leftrightarrow n)) \triangleright H\), it suffices to prove that, for any even number \(n\), we have \((r \leftrightarrow n) \triangleright H\).

Corresponding reasoning rules:

\[
\Gamma, \ x : A \vdash H_1 \triangleright H_2 \\
\Gamma \vdash (\exists (x : A). \ H_1) \triangleright H_2 \\
\Gamma, \ P \vdash H_1 \triangleright H_2 \\
\Gamma \vdash ([P] \ast H_1) \triangleright H_2
\]

(x \notin H_2)

Same with implicit proof contexts:

\[
\forall x. \ (H_1 \triangleright H_2) \\
(\exists x. \ H_1) \triangleright H_2 \\
P \Rightarrow (H_1 \triangleright H_2) \\
([P] \ast H_1) \triangleright H_2
\]
Extraction and instantiation: an example

Exercise: prove the following heap implication.

\[ \exists n. [n \neq 2 \land \text{prime } n] \land (r \leftarrow n) \triangleright \exists k. (r \leftarrow (2k + 1)) \]

Solution:

\[
\begin{array}{c}
\frac{n : \text{int} \mid (r \leftarrow n) \triangleright (r \leftarrow n)}{n : \text{int} \mid (r \leftarrow n) \triangleright (r \leftarrow 2p + 1)}
\end{array}
\]

\[
\begin{array}{c}
\frac{n : \text{int}, p : \text{int}, n = 2p + 1 \mid (r \leftarrow n) \triangleright \langle p/k \rangle(r \leftarrow 2k + 1)}{n : \text{int}, p : \text{int}, n = 2p + 1 \mid (r \leftarrow n) \triangleright \exists k. (r \leftarrow 2k + 1)}
\end{array}
\]

\[
\begin{array}{c}
\frac{n : \text{int}, n \neq 2, \text{prime } n \mid (r \leftarrow n) \triangleright \exists k. (r \leftarrow 2k + 1)}{\emptyset \mid \exists n. [n \neq 2 \land \text{prime } n] \land (r \leftarrow n) \triangleright \exists k. (r \leftarrow 2k + 1)}
\end{array}
\]
Content

Separation Logic: a first example
Separation Logic triples
Definition of heap predicates
The separation algebra

Structural rules
  Frame rule
  Rule of consequence
  Garbage collection rule
  The combined rule
  Extraction rules

Reasoning rules for terms
Towards a frame rule

Recall the reasoning that applied to the specification of “\(\text{incr } r\)”:  

\[
\frac{\{r \leftarrow n\} (\text{incr } r) \{\lambda tt. \ r \leftarrow (n + 1)\}}{\{s \leftarrow m \ast r \leftarrow n\} (\text{incr } r) \{\lambda tt. \ r \leftarrow (n + 1) \ast s \leftarrow m\}}
\]

More generally, we can frame over any heap predicate \(H\):  

\[
\frac{\{r \leftarrow n\} (\text{incr } r) \{\lambda tt. \ r \leftarrow (n + 1)\}}{\{H \ast r \leftarrow n\} (\text{incr } r) \{\lambda tt. \ r \leftarrow (n + 1) \ast H\}}
\]
The frame rule

The frame rule asserts that any triple remains valid when both the pre- and the post-condition are extended with a same heap predicate.

$$\{H_1\} \ t \ \{\lambda x. H'_1\}$$

$$\{H_1 \ast H_2\} \ t \ \{\lambda x. H'_1 \ast H_2\}$$

The frame rule can be reformulated more concisely:

$$\{H_1\} \ t \ \{Q_1\}$$

$$\{H_1 \ast H_2\} \ t \ \{Q_1 \ast H_2\}$$

where $Q \ast H \equiv \lambda x. (Q \ x \ast H)$.
Interpretation of the frame rule

Reformulation of the frame rule:

\[
H = H_1 * H_2 \\
\{H_1\} t \{Q_1\} \\
\{H\} t \{Q\} \\
Q_1 * H_2 = Q
\]

The execution of \( t \) cannot affect any data that is not described by \( H_1 \). In particular, \( H_2 \) is completely hidden from the reasoning about \( t \).
Frame rule and allocation

The specification of “ref 3” ensures that the result is a fresh location.

\[ \{ [] \} \ (\text{ref 3}) \ \{ \lambda r. \ (r \leftarrow 3) \} \]

For example, applying the frame rule with \( s \leftarrow 5 \) gives:

\[ \{ s \leftarrow 5 \} \ (\text{ref 3}) \ \{ \lambda r. \ (r \leftarrow 3) \ast (s \leftarrow 5) \} \]

The post-condition ensures \( r \neq s \).
Strengthening of the pre-condition

We should be able to prove that

\[
\text{if } \{ \exists n. (r \leftrightarrow n) \ast [\text{even } n] \} t \{ Q \} \\
\text{then } \{ r \leftrightarrow 6 \} t \{ Q \}.
\]

Rule for strengthening pre-conditions:

\[
\frac{H \triangleright H'}{\{H\} t \{Q\}} \quad \text{if } \{H'\} t \{Q\}
\]
Weakening of the post-condition

We should be able to prove that

\[
\text{if } \{H\} \ t \ \{\lambda x. [x = 4] \ast (r \rightarrow 6)\} \\
\text{then } \{H\} \ t \ \{\lambda x. [\text{even } x] \ast \exists n. (r \rightarrow n) \ast [\text{even } n]\}.
\]

Rule for weakening post-conditions:

\[
\frac{\{H\} \ t \ \{Q'\} \quad Q' \triangleright Q}{\{H\} \ t \ \{Q\}}
\]

Recall that \(Q' \triangleright Q\) is defined as: \(\forall v. Q' \ u \triangleright Q \ u\).
The rule of consequence

The two rules

\[
\frac{H \triangleright H'}{\{H\} \triangleright \{Q\}} \quad \text{and} \quad \frac{\{H\} \triangleright \{Q'\}}{\{H\} \triangleright \{Q\}}
\]

can be combined into the rule of consequence:

\[
\frac{H \triangleright H' \quad \{H'\} \triangleright \{Q'\}}{\{H\} \triangleright \{Q\}}
\]

→ \(H\) and \(H'\) must cover the same set of memory cells. Same for \(Q\) and \(Q'\).
Need for garbage collection

We would like the following triple to hold:

\[
\{ [] \} \ (\text{let } r = \text{ref } 3 \ \text{in} \ !r) \ \{ \lambda x. [x = 3]\}
\]

Yet, the post-condition we have is \( \lambda x. [x = 3] \ast \exists r. (r \leftarrow 3) \).

We need to be able to discard \( \exists r. r \leftarrow 3 \) from the post-condition.

Some form of garbage collection in the program logic is needed to reflect the garbage collection performed in the programming language.
The garbage collection rules

Garbage collection for discarding a piece of the post-condition:

\[ \frac{\{ H \} \ t \ \{ Q \star H' \} \quad \text{gc-post}}{\{ H \} \ t \ \{ Q \}} \]

Garbage collection for discarding a piece of the pre-condition:

\[ \frac{\{ H \} \ t \ \{ Q \} \quad \text{gc-pre}}{\{ H \star H' \} \ t \ \{ Q \}} \]

Exercise: show that gc-pre is derivable from gc-post and frame.

\[ \frac{\{ H \} \ t \ \{ Q \} \quad \text{frame}}{\{ H \star H' \} \ t \ \{ Q \star H' \} \quad \text{gc-post}} \]

\[ \frac{\{ H \star H' \} \ t \ \{ Q \star H' \} \quad \text{gc-post}}{\{ H \star H' \} \ t \ \{ Q \}} \]
The combined rule

Consider the three structural rules:

\[
H = H_1 \ast H_2 \quad \{H_1\} \ t \ \{Q_1\} \quad Q_1 \ast H_2 = Q \quad \text{frame}
\]
\[
\{H\} \ t \ \{Q\}
\]

\[
H \succ H' \quad \{H'\} \ t \ \{Q'\} \quad Q' \bowtie Q \quad \text{conseq}
\]
\[
\{H\} \ t \ \{Q\}
\]

\[
\{H\} \ t \ \{Q\} \quad \text{gc}
\]

They can be combined into a single rule:

\[
H \succ H_1 \ast H_2 \quad \{H_1\} \ t \ \{Q_1\} \quad Q_1 \ast H_2 \bowtie Q \ast H_3
\]
\[
\{H\} \ t \ \{Q\}
\]
Extraction of existentials and propositions

Consider the triple

\[ \{ \exists n. (r \rightarrow n) \ast [\text{even } n] \} \ (\forall r) \ {Q}. \]

To prove it, we need to show that, for any even number \( n \), we have

\[ \{ r \rightarrow n \} \ (\forall r) \ {Q}. \]

Corresponding reasoning rules:

\[
\begin{align*}
\forall x. \ {H} & \vdash \ {Q} & P & \Rightarrow \ {H} \vdash \ {Q} \\
\exists x. \ {H} & \vdash \ {Q} & \exists x. \ {H} & \vdash \ {Q} \\
\{ P \ast H \} & \vdash \ {Q} & \{ [P] \ast H \} & \vdash \ {Q}
\end{align*}
\]

Same, with explicit proof contexts:

\[
\begin{align*}
\Gamma, \ x \vdash \ {H} \vdash \ {Q} & & \Gamma, P \vdash \ {H} \vdash \ {Q} \\
\Gamma \vdash \exists x. \ {H} \vdash \ {Q} & & \Gamma \vdash \{ [P] \ast H \} \vdash \ {Q}
\end{align*}
\]
Content

Separation Logic: a first example
Separation Logic triples
Definition of heap predicates
The separation algebra
Structural rules

Reasoning rules for terms
  Grammar of values and terms
  Rule for sequences
  Rule for values
  Rule for let-bindings
  Rule for conditionals
  Rule for top-level functions
  Rule for local functions
  Summary of the rules
Grammar of values and terms

\[
v := \quad | \quad x \quad | \quad tt \quad | \quad b \quad | \quad n \quad | \quad (v_1, v_2) \quad | \quad v_1 \text{ op } v_2 \quad \text{(if op total)}
\]

\[
t := \quad | \quad v
\quad | \quad t_1 ; t_2
\quad | \quad \text{let } x = t_1 \text{ in } t_2
\quad | \quad \text{if } v \text{ then } t_1 \text{ else } t_2
\quad | \quad \text{let rec } f = \lambda x. t_1 \text{ in } t_2
\quad | \quad (v_1 v_2)
\]

Remarks:

- Terms are put in A-normal form:

\[
\text{if } t_0 \text{ then } t_1 \text{ else } t_2 \quad \equiv \quad \text{let } v = t_0 \text{ in } \text{if } v \text{ then } t_1 \text{ else } t_2
\]
\[
(t_1 t_2) \quad \equiv \quad \text{let } f = t_1 \text{ in } \text{let } v = t_2 \text{ in } (f v)
\]

- “let rec \( f = \lambda x. t_1 \text{ in } t_2 \)” is another notation for “let rec \( f x = t_1 \text{ in } t_2 \)”.
Towards Separation Logic derivations

Recall the specification of the sequence “incr r; !s”:

\[
\begin{align*}
\{r \leftarrow n \ast s \leftarrow m\} & (\text{incr } r) \{\lambda tt. r \leftarrow n + 1 \ast s \leftarrow m\} \\
\{r \leftarrow n + 1 \ast s \leftarrow m\} & (\text{!s}) \{\lambda x. [x = m] \ast r \leftarrow n + 1 \ast s \leftarrow m\} \\
\{r \leftarrow n \ast s \leftarrow m\} & (\text{incr } r; \text{!s}) \{\lambda x. [x = m] \ast r \leftarrow n + 1 \ast s \leftarrow m\}
\end{align*}
\]

We need a reasoning rule to establish triples for sequences, of the form:

\[
\{\ldots} \ t_1 \ \{\ldots\} \quad \{\ldots\} \ t_2 \ \{\ldots\}
\]

\[
\{H\} \ (t_1; t_2) \ {Q}\]
Reasoning rule for sequences

First possibility:

\[
\begin{align*}
\{H\} t_1 \{\lambda tt. H'\} & \quad \{H'\} t_2 \{Q\} \\
\{H\} (t_1 ; t_2) \{Q\}
\end{align*}
\]

Second possibility:

\[
\begin{align*}
\{H\} t_1 \{Q'\} & \quad \{Q' tt\} t_2 \{Q\} \\
\{H\} (t_1 ; t_2) \{Q\}
\end{align*}
\]

Remark: \( Q = \lambda tt. H' \) is equivalent to \( Q tt = H' \).
Reasoning rule for values

Recall the specification of the term 3.

\[ \{ [ ] \} \ 3 \ \{ \lambda x. [x = 3] \} \]

The reasoning rule for values could take the form:

\[ \frac{}{\{ [ ] \} \ v \ \{ \lambda x. [x = v] \}} \]

Exercise. State a reasoning rule for values in the form:

\[ \ldots \Rightarrow \ldots \]
\[ \{ H \} \ v \ \{ Q \} \]

Solution:

\[ \frac{H \Rightarrow Q \ v}{\{ H \} \ v \ \{ Q \}} \]

Remark: \[ [ ] \Rightarrow ((\lambda x. [x = 3]) \ 3) \].
Reasoning rule for let-bindings

Exercise: complete the reasoning rule below.

\[
\frac{\{\ldots\} \; t_1 \; \{\ldots\} \; \forall x. \ (\{\ldots\} \; t_2 \; \{\ldots\})}{\{H\} \ (\text{let } x = t_1 \text{ in } t_2) \ \{Q\}}
\]

Solution:

\[
\frac{\{H\} \; t_1 \; \{Q'\} \; \forall x. \ {Q'} \; x \; t_2 \; \{Q\}}{\{H\} \ (\text{let } x = t_1 \text{ in } t_2) \ \{Q\}}
\]
Example of let-binding

Consider the triple:

\{ r \leftarrow 3 \} \ (\text{let } a = !r \text{ in } a+1) \ \{ Q \}

What should \( Q \) be? What should \( Q' \) be in the rule for let-bindings?

\[
Q' \equiv \lambda y. [y = 3] \ast (r \leftarrow 3) \\
Q \equiv \lambda x. [x = 4] \ast (r \leftarrow 3)
\]

\[
\frac{[\ ] \triangleright [\ ] \quad 4 = 4}{[\ ] \triangleright [4 = 4]} \\
\frac{a : \text{int}, \ a = 3 \vdash [\ ] \triangleright (\lambda x. [x = 4]) \ (a + 1)}{a : \text{int}, \ a = 3 \vdash \{ [\ ] \} \ (a+1) \ \{ \lambda x. [x = 4] \}} \\
\frac{a : \text{int}, \ a = 3 \vdash \{ r \leftarrow 3 \} \ (a+1) \ \{ Q \}}{a : \text{int} \vdash \{ [a = 3] \ast (r \leftarrow 3) \} \ (a+1) \ \{ Q \}} \\
\frac{\vdash \{ r \leftarrow 3 \} \ (\text{let } a = !r \text{ in } a+1) \ \{ Q \}}{\vdash \{ Q' a \} \ (a+1) \ \{ Q \}} \\
\]

\[
\vdash \{ r \leftarrow 3 \} \ (\text{let } a = !r \text{ in } a+1) \ \{ Q \}
\]
Reasoning rule for conditionals

Consider the example:

\{ r \leftarrow y \} (\text{if } x > 0 \text{ then } (r := x) \text{ else } (r := -x)) \{ \lambda t. r \leftarrow |x| \}

To establish such a triple, we need the reasoning rule for conditionals:

\[
\frac{(b = \text{true} \Rightarrow \{H\} t_1 \{Q\}) \quad (b = \text{false} \Rightarrow \{H\} t_2 \{Q\})}{\{H\} (\text{if } b \text{ then } t_1 \text{ else } t_2) \{Q\}}
\]

On the example, we prove the else-branch as follows:

\[
\begin{align*}
& x \leq 0 \vdash \{r \leftarrow y\} (r := -x) \{\lambda t. r \leftarrow (-x)\} \\
& x \leq 0 \vdash (\lambda t. r \leftarrow (-x)) \triangleright (\lambda t. r \leftarrow |x|) \\
& (x > 0) = \text{false} \vdash \{r \leftarrow y\} (r := -x) \{\lambda t. r \leftarrow |x|\}
\end{align*}
\] weaken
Reasoning rule for top-level functions

Given a top-level function $f$ defined as $\lambda x. t$, the behavior of the application $(fv)$ is the same as the behavior of the term $\langle v/x \rangle t$.

Reasoning rule for inlining of the body of the function:

\[
\begin{align*}
   f & \equiv \lambda x. t & \{H\} & \langle v/x \rangle t \{Q\} \\
   & \{H\} (fv) \{Q\}
\end{align*}
\]
Verification of a function

Consider the definition of the function `incr`:

```ocaml
let incr = fun r ->
  let a = !r in
  r := a+1
```

Let us prove the following specification:

\[
\forall r n. \{ r \leftarrow n \} (\text{incr } r) \{ \lambda tt. (r \leftarrow n + 1) \}.
\]

Fix r and n. By the reasoning rule for function, we have to prove

\[
\{ r \leftarrow n \} (\text{let } a = !r \text{ in } r := a+1) \{ \lambda tt. (r \leftarrow n + 1) \}.
\]
Functions and effects

What is the specification of \( f \) in the following program?

```ocaml
let r = ref 3 in
let f () =
    incr r in
f();
f();
!r
```

Solution:

\[
\forall n. \{ r \leftarrow n \} (f tt) \{ \lambda tt. r \leftarrow n + 1 \}
\]

Goes from \( \{ r \leftarrow 3 \} \) to \( \{ r \leftarrow 4 \} \) and \( \{ r \leftarrow 5 \} \), so the program prints 5.
Functions and aliasing

What are the two specifications of \texttt{swap}?

\begin{verbatim}
let swap r s =
    let a = !r in
    let b = !s in
    r := b;
    s := a
\end{verbatim}

Solution:

\begin{align*}
\forall rsnm. \hspace{1em} & \{(r \leftarrow n) * (s \leftarrow m)\} (\text{swap } r \text{ } s) \{\lambda tt. (r \leftarrow m) * (s \leftarrow n)\} \\
\forall rsn. \hspace{1em} & \{[r = s] * r \leftarrow n\} (\text{swap } r \text{ } s) \{\lambda tt. r \leftarrow n\}
\end{align*}
Functions with aliasing and effects

What are the two specifications of \( f \) in the following program?

```plaintext
let r = ref 3 in
let f s =
  incr r;
  incr s
```

Solution:

\[
\forall snm. \quad \{(r \leftarrow n) \ast (s \leftarrow m)\} \quad (f \quad s) \quad \{\lambda tt. \quad (r \leftarrow n + 1) \ast (s \leftarrow m + 1)\}
\]

\[
\forall sn. \quad \{[r = s] \ast r \leftarrow n\} \quad (f \quad s) \quad \{\lambda tt. \quad r \leftarrow n + 2\}
\]
Verification of a recursive function

Consider the function:

```plaintext
let rec fn =
  if n = 0
  then 0
  else let y = f(n-1) in y+2
```

Exercise: what is the specification of this function? How to prove it?

$$\forall n. \ {[n \geq 0]} \ (f\ n) \ \{\lambda r. [r = 2n]\}.$$ 

By induction on $n$. The goal is to prove $\{[n \geq 0]\} \ (f\ n) \ \{\lambda r. [r = 2n]\}$ using the induction hypothesis:

$$\{[n - 1 \geq 0]\} \ (f\ (n - 1)) \ \{\lambda r. [r = 2(n - 1)]\}.$$
Reasoning rule for local functions

The reasoning rule for local functions has the form:

\[
\forall f. (\ldots) \Rightarrow \{H\} \ t_2 \ \{Q\}
\]

\[
\{H\} \ (\text{let rec } f = \lambda x. t_1 \ \text{in } t_2) \ \{Q\}
\]

To see what should fill the dots, recall the rule for top-level functions:

\[
f \equiv \lambda x. \ t_1 \quad \{H'\} \ (\langle v/x \rangle \ t_1) \ \{Q'\}
\]

\[
\{H'\} \ (f \ v) \ \{Q'\}
\]

Therefore, we can assume about \( f \) the following:

\[
\forall v H' Q'. \quad \{H'\} \ (\langle v/x \rangle \ t_1) \ \{Q'\} \quad \Rightarrow \quad \{H'\} \ (f \ v) \ \{Q'\}
\]

or simply:

\[
\forall x H' Q'. \quad \{H'\} \ t_1 \ \{Q'\} \quad \Rightarrow \quad \{H'\} \ (f \ x) \ \{Q'\}
\]
The reasoning rule for local functions is therefore:

$$\forall f. \ (\forall x H'Q'. \ {H'} t_1 {Q'} \Rightarrow {H'} (f \ x) {Q'}) \Rightarrow {H} t_2 {Q}$$

$$\{H\} \ (\text{let rec } f = \lambda x. t_1 \text{ in } t_2) \ {Q}$$

Note that we need no specific support for recursive functions, whose specification is simply established by induction.
Verification of a local function

Suppose our goal is to prove \( \{H\} t \{Q\} \), where \( t \) is the program below:

```plaintext
let incr r =
  let a = !r in
  r := a+1 in
let s = ref 3 in
incr s;
incr s
```

By the rule for local functions, we have to prove:

\[
\forall r H' Q'. \{H'\} (let a = !r in r := a+1) \{Q'\} \Rightarrow \{H'\} (incr r) \{Q'\}
\]

\[
\vdash \{H\} (let s = ref 3 in incr s; incr s) \{Q\}
\]

From the hypothesis, we can derive and keep the specification of incr:

\[
\forall r n. \ \{r \leftarrow n\} (incr r) \{\lambda tt. (r \leftarrow n + 1)\}.
\]
Summary: rules for terms

\[ \frac{H \Rightarrow Q \nu}{\{H\} \nu \{Q\}} \quad \frac{\{H\} \ t_1 \ \{Q'\} \quad \{Q' \ tt\} \ t_2 \ \{Q\}}{\{H\} \ (t_1 \ ; \ t_2) \ \{Q\}} \]

\[ \frac{\{H\} \ t_1 \ \{Q'\} \quad \forall x. \ \{Q' \ x\} \ t_2 \ \{Q\}}{\{H\} \ (\text{let } x = t_1 \ \text{in} \ t_2) \ \{Q\}} \]

\[ b = \text{true} \Rightarrow \{H\} \ t_1 \ \{Q\} \quad b = \text{false} \Rightarrow \{H\} \ t_2 \ \{Q\} \]

\[ \{H\} \ (\text{if } b \ \text{then} \ t_1 \ \text{else} \ t_2) \ \{Q\} \]

\[ \forall f. \ (\forall xH'Q'. \ \{H'\} \ t_1 \ \{Q'\} \Rightarrow \ {H'} \ (f \ x) \ \{Q'\}) \Rightarrow \ \{H\} \ t_2 \ \{Q\} \]

\[ \{H\} \ (\text{let rec } f = \lambda x. \ t_1 \ \text{in} \ t_2) \ \{Q\} \]

Remark: there is no rule for function calls; the rule for function definitions provides hypotheses for reasoning about function calls.
Summary: structural rules and primitive functions

Combined structural rule:

\[ \begin{array}{c}
H \triangleright H_1 \ast H_2 \\
\{H_1\} \ t \ \{Q_1\} \\
Q_1 \ast H_2 \triangleright Q \ast H_3 \\
\end{array} \]
\[ \{H\} \ t \ \{Q\} \]

Extraction rules:

\[ \forall x. \ \{H\} \ t \ \{Q\} \]
\[ \{\exists x. H\} \ t \ \{Q\} \]

Specification of operations on references:

\[ \forall v. \ \{[]\} \ (\text{ref } v) \ \{\lambda r. \ (r \leftarrow v)\} \]
\[ \forall rv. \ \{r \leftarrow v\} \ (!r) \ \{\lambda x. \ [x = v] \ast (r \leftarrow v)\} \]
\[ \forall rvv'. \ \{r \leftarrow v'\} \ (r := v) \ \{\lambda tt. \ (r \leftarrow v)\} \]
Conclusion

In this course:

- interpretation of Separation Logic triples,
- properties of the star and of heap implication,
- structural rules of separation logic,
- reasoning rules for terms,
- support for aliasing and function with effects.
Exercise: proof of commutativity

Prove that the separating conjunction is commutative:

\[ H_1 \ast H_2 = H_2 \ast H_1 \]

You will need to use the definition of \( \ast \) and predicate extensionality:

\[ (\forall x. P x \iff Q x) \Rightarrow (P = Q). \]
Exercise: a heap implication

Prove the following heap implication:

$$\exists k. \ (r \leftarrow 4k) \quad \Rightarrow \quad \exists n. \ (r \leftarrow n) * [\text{even } n]$$

Remark: “even $n$” is defined as “$\exists m. n = 2m$”. 
Exercise: verification of a recursive function

Complete the verification of the function:

```plaintext
let rec fn =
  if n = 0
    then 0
  else let y = f(n-1) in y+2
```

with respect to the following specification:

\[ \forall n. \ n \geq 0 \Rightarrow \{ [] \} (f \ n) \{ \lambda r. [ r = 2n ] \}. \]
Exercise: exchange

Specify `exchange` and show that the rest of the code prints 0.

```plaintext
let a = ref 0 in
let r = ref a in

let exchange s =
    let u = !r in
    r := s;
u in

let b = ref 1 in
let a' = exchange b in
let _ = exchange a' in
!(!r)
```
Solution to exercises
Solution: proof of commutativity

Goal: prove the equality $H_1 * H_2 = H_2 * H_1$.

By predicate extensionality, it suffices to prove:

$$\forall m. (H_1 * H_2) m \iff (H_2 * H_1) m$$

By symmetry, it suffices to assume $(H_1 * H_2) m$ and prove $(H_2 * H_1) m$.

By definition of $*$, there exist $m_1$ and $m_2$ such that:

$$m_1 \perp m_2 \land m = m_1 \oplus m_2 \land H_1 m_1 \land H_2 m_2$$

The goal that we have to prove is equivalent to:

$$\exists m'_1 m'_2. \quad m'_1 \perp m'_2 \land m = m'_1 \oplus m'_2 \land H_2 m'_1 \land H_2 m'_2$$

We conclude by instantiating $m'_1$ as $m_2$ and $m'_2$ as $m_1$.
Solution: a heap implication

Exercise: prove the following heap implication.

\[ \exists k. (r \leftarrow 4k) \implies \exists n. (r \leftarrow n) \ast [\text{even } n] \]

Solution:

\[
\begin{align*}
(r \leftarrow 4k) & \implies (r \leftarrow 4k) & \text{even } (4k) \\
(r \leftarrow 4k) & \implies (r \leftarrow 4k) \ast [\text{even } 4k] \\
(r \leftarrow 4k) & \implies (4k/n) \ast ((r \leftarrow n) \ast [\text{even } n]) \\
\forall k. & \quad (r \leftarrow 4k) \implies \exists n. (r \leftarrow n) \ast [\text{even } n] \\
\exists k. & \quad (r \leftarrow 4k) \implies \exists n. (r \leftarrow n) \ast [\text{even } n]
\end{align*}
\]
Exercise: verification of a recursive function

Complete the verification of the function:

```
let rec f n =
  if n = 0
  then 0
  else let y = f(n-1) in y+2
```

with respect to the following specification:

\[
\forall n. n \geq 0 \Rightarrow \{ \square \} (f \ n) \{ \lambda r. [r = 2n] \}.
\]

By induction on \( n \). The goal is to prove \([n \geq 0]\) \((f \ n) \{\lambda r. [r = 2n]\}\) using the induction hypothesis:

\[
\{[n - 1 \geq 0]\} (f (n - 1)) \{\lambda r. [r = 2(n - 1)]\}.
\]
Solution: verification of a recursive function

Using the reasoning rule for functions, the goal becomes:

\[ \{ [] \} \ (\text{if } n = 0 \text{ then } 0 \text{ else let } y = f(n-1) \text{ in } y+2) \ \{ \lambda r. [r = 2n] \} \]

By the rule for conditionals:

- Case \( n = 0 \). The goal is: \( \{ [] \} 0 \ \{ \lambda r. [r = 2n] \} \).
  - we prove \( [] \triangleright ((\lambda r. [r = 2n]) 0) \), using the fact that \( n = 0 \).
- Case \( n \neq 0 \). Not that since \( n \geq 0 \), we have \( n > 0 \), thus \( n - 1 \geq 0 \). The goal is \( \{ [] \} \ (\text{let } y = f(n-1) \text{ in } y+2) \ \{ \lambda r. [r = 2n] \} \).
  - by the let-rule, using \( Q' \equiv \lambda r. [r = 2(n-1)] \).
  - first premise: \( \{ [] \} \ (f (n - 1)) \ \{ Q' \} \), which is exactly the IH.
  - second premise: \( \forall y. \ \{ Q' y \} \ (y + 2) \ \{ \lambda r. [r = 2n] \} \).
    - by the rule for values: \( Q' y \triangleright (\lambda r. [r = 2n]) (y + 2) \).
    - after simplification: \( [y = 2(n - 1)] \triangleright [y + 2 = 2n] \), which is true.
Solution: exchange

Program:

\[
\begin{align*}
&\text{let } a = \text{ref } 0 \text{ in} \\
&\text{let } r = \text{ref } a \text{ in} \\
&\text{let } \text{exchange } s = \text{let } u = !r \text{ in} \\
&\quad r := s; \\
&\quad u \text{ in} \\
&\text{let } b = \text{ref } 1 \text{ in} \\
&\text{let } a' = \text{exchange } b \text{ in} \\
&\text{let } b' = \text{exchange } a' \text{ in} \\
&!(!r)
\end{align*}
\]

Specification of exchange:

\[
\forall ts. \{r \leftarrow t\} (\text{exchange } s) \{\lambda x.[x = t] * r \leftarrow s\}
\]

We frame on \(a \leftarrow 0 * b \leftarrow 1\), and see the rest of the state evolve from \(r \leftarrow a\) to \(r \leftarrow b * [a' = a]\) and then to \(r \leftarrow a' * [b' = b]\).