Separation Logic

Part 1

Arthur Charguéraud

February 2015

Representation of heaps

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

$\varnothing$	the empty heap
$\langle l \mapsto v \rangle$	a singleton heap
$m\langle l \rangle$	read in a heap
$m\langle l \mapsto v \rangle$	update of a heap
$m_1 \uplus m_2$	merge of two heaps

Merge is only defined for heaps with disjoint domains:

$$m_1 \perp m_2 \quad \equiv \quad \mathrm{dom}(m_1) \cap \mathrm{dom}(m_2) = \varnothing$$

Heap predicates

A heap predicate H characterizes memory stores of a particular shape.
A heap predicate has type "Heap → Prop", i.e. "$H\,m$" is a proposition.

We next define the primitive heap predicates:

$[\,]$	empty heap
$[P]$	pure fact
$l \mapsto v$	singleton heap
$l \hookrightarrow v$	singleton reference heap
$H * H'$	separating conjunction
$\exists x.\, H$	existential quantification

Empty heap and pure facts

Definition:

$$[\,] \quad \equiv \quad \lambda m.\ m = \varnothing$$
$$[P] \quad \equiv \quad \lambda m.\ m = \varnothing \ \wedge \ P$$

Example: specification of the program "`let a = 3 and b = a+1`".

Before: $[\,]$
After: $[a = 3 \ \wedge \ b = 4]$

Singleton reference heap

Recall that, in Caml, "`ref v`" is equivalent to "`{contents = v}`".
Definition:

$$l \hookrightarrow v \quad \equiv \quad \lambda m.\ m = \langle l \mapsto \{\!|\text{contents}{=}v|\!\}\rangle$$

Exercise: specify the program "`let r = ref 3`".

Before: $[\,]$
After: $r \hookrightarrow 3$

Exercise: specify the program "`incr s`".

Before: $s \hookrightarrow n$, for some n
After: $s \hookrightarrow n+1$

Singleton heap

Definition:

$$l \mapsto v \quad \equiv \quad \lambda m.\ m = \langle l \mapsto v\rangle$$

For example, the program:

```
type point = { x : int; y : int }
let p = { x = 3; y = 4 }
```

is specified as $p \mapsto \{\!|\text{x}{=}3; \text{y}{=}4|\!\}$.

Note that the notation $l \hookrightarrow v$ is equivalent to $l \mapsto \{\!|\text{contents}{=}v|\!\}$.

Separation Logic's star

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

Definition:

$$H_1 * H_2 \quad \equiv \quad \lambda m.\ \exists m_1 m_2. \begin{cases} m_1 \perp m_2 \\ m = m_1 \uplus m_2 \\ H_1\, m_1 \\ H_2\, m_2 \end{cases}$$

Example: $(r \hookrightarrow 3) * (s \hookrightarrow 4)$ describes two distinct reference cells.

Separating conjunction and aliasing

Exercise: specify "`let r = ref 5 and s = ref 3 and t = r`".
Follow-up: specify the state after executing "`incr r; incr t`".

Wrong answer: $(r \hookrightarrow 5) * (s \hookrightarrow 3) * (t \hookrightarrow 5)$.
Correct answer:

Before: $[\,]$
After: $(r \hookrightarrow 5) * (s \hookrightarrow 3) * [t = r]$
Follow-up: $(r \hookrightarrow 7) * (s \hookrightarrow 3) * [t = r]$

Nested references

Exercise: specify "`let r = (let t = ref 3 in ref t)`".
Follow-up: specify the state after executing "`let u = !r and v = !u`".

The location `t` needs to be existentially quantified:

Before:	$[\,]$
After:	$\exists t.\ (r \hookrightarrow t) * (t \hookrightarrow 3)$
Follow-up:	$\exists t.\ (r \hookrightarrow t) * (t \hookrightarrow 3) * [u = t \ \wedge\ v = 3]$

Existential quantification

Definition of the existential quantifier for heap predicates:

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

Compare:

$$(P : \mathsf{Prop}) \ \vdash\ (\exists x.\, P) \ :\ \mathsf{Prop}$$

$$(H : \mathsf{Heap} \to \mathsf{Prop}) \ \vdash\ (\exists x.\, H) \ :\ \mathsf{Heap} \to \mathsf{Prop}$$

Existential quantification for abstraction

Existentials are introduced by allocation but also by abstraction.

For example:

Exact:	$(r \hookrightarrow 6)$
Abstract:	$\exists n.\, (r \hookrightarrow n) * [\mathsf{even}\, n]$

Summary

Definition of the core Separation Logic heap predicates:

$$\begin{array}{lcl}
[\,] & \equiv & \lambda m.\ m = \varnothing \\
[P] & \equiv & \lambda m.\ m = \varnothing \ \wedge\ P \\
l \mapsto v & \equiv & \lambda m.\ m = \langle l \mapsto v \rangle \\
l \hookrightarrow v & \equiv & \lambda m.\ m = \langle l \mapsto \{\!|\mathsf{contents}{=}v|\!\}\rangle \\
H_1 * H_2 & \equiv & \lambda m.\ \exists m_1 m_2.\ \left\{ \begin{array}{l} m_1 \perp m_2 \\ m = m_1 \uplus m_2 \\ H_1\, m_1 \\ H_2\, m_2 \end{array} \right. \\
\exists x.\, H & \equiv & \lambda m.\ \exists x.\, H\, m
\end{array}$$

Separation Logic triples

A Separation Logic *triple* is of the form $\{H\}\ t\ \{\lambda x.\, H'\}$ where

- H describes the initial heap
- t is the term being specified
- x is a name for the value produced by t
- H' describes the final heap and the output value x

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

- the pre-condition H has type: Heap → Prop
- the term t has some type A
- the post-condition Q has type: A → Heap → Prop

Examples of triples

Example 1:

$$\{\,[\,]\,\}\ (\texttt{ref 3})\ \{\lambda r.\ r \hookrightarrow 3\}$$

Example 2:

$$\{\,[\,]\,\}\ (3)\ \{\lambda x.\, [x = 3]\}$$

Example 3:

$$\{r \hookrightarrow 3\}\ (\texttt{!r})\ \{\lambda x.\ [x = 3] * (r \hookrightarrow 3)\}$$

Example 4:

$$\{r \hookrightarrow 3\}\ (\texttt{incr r})\ \{\lambda tt.\ r \hookrightarrow 4\}$$

Remark: tt denotes the unit value, and "$\lambda tt.$ ..." is like "fun() → ...".

Exercise on triples

Exercise: specify the term "`incr r; !s`" under the assumption `r` ≠ `s`.

Solution:

$$\begin{array}{ll} \forall nm. & \{(r \hookrightarrow n) * (s \hookrightarrow m)\} \\ & (\texttt{incr r; !s}) \\ & \{\lambda x.\, [x = m] * (r \hookrightarrow n + 1) * (s \hookrightarrow m)\} \end{array}$$

Specification of a function

To specify a function, we specify the behavior of its applications.

Example:

$$\forall rn.\quad \{r \hookrightarrow n\}\ (\texttt{incr r})\ \{\lambda tt.\ r \hookrightarrow (n + 1)\}$$

Exercise: specify applications of the function "`let f x = 2 * x`".
Solution:

$$\forall x.\quad \{[\,]\}\ (\texttt{f x})\ \{\lambda y.\ [y = 2x]\}$$

Specification of primitive functions

Exercise: specify the primitive operations on references.

$$(\texttt{ref v})$$
$$(\texttt{!r})$$
$$(\texttt{r := v})$$

Solution:

$$\forall v. \quad \{[\,]\}\ (\texttt{ref v})\ \{\lambda r.\ (r \hookrightarrow v)\}$$
$$\forall r v. \quad \{r \hookrightarrow v\}\ (\texttt{!r})\ \{\lambda x.\ [x = v] * (r \hookrightarrow v)\}$$
$$\forall r v v'. \quad \{r \hookrightarrow v'\}\ (\texttt{r := v})\ \{\lambda tt.\ (r \hookrightarrow v)\}$$

Function with local state

Exercise: what is the specification of `f` in the following program?

```
let r = ref 3
let f () =
  incr r
```

Follow-up: show that the code below returns 5.

```
f();
f();
!r
```

Specification:

$$\forall n. \quad \{r \hookrightarrow n\}\ (\texttt{f ()})\ \{\lambda tt.\ r \hookrightarrow n+1\}$$

Successive states: $r \hookrightarrow 3$ $\quad$ $r \hookrightarrow 4$ $\quad$ $r \hookrightarrow 5$

Function with aliased arguments

What are the two specifications of `swap`?

```
let swap r s =
  let a = !r in
  let b = !s in
  r := b;
  s := a
```

Solution:

$$\forall r s n m. \quad \{(r \hookrightarrow n) * (s \hookrightarrow m)\}\ (\texttt{swap r s})\ \{\lambda tt.\ (r \hookrightarrow m) * (s \hookrightarrow n)\}$$
$$\forall r s n. \quad \{[r = s] * (r \hookrightarrow n)\}\ (\texttt{swap r s})\ \{\lambda tt.\ r \hookrightarrow n\}$$

Function with local state and aliasing

What are the two specifications of `g`?

```
let r = ref 3 in
let g s =
  incr r;
  incr s
```

Solution:

$$\forall s n m. \quad \{(r \hookrightarrow n) * (s \hookrightarrow m)\}\ (\texttt{g s})\ \{\lambda tt.\ (r \hookrightarrow n+1) * (s \hookrightarrow m+1)\}$$
$$\forall s n. \quad \{[r = s] * (r \hookrightarrow n)\}\ (\texttt{g s})\ \{\lambda tt.\ (r \hookrightarrow n+2)\}$$

Function with garbage collection

What is the specification of `next`?

```
let next x =
  let r = ref x in
  incr r;
  !r
```

Solution:

$$\forall x. \quad \{[\,]\}\ (\texttt{next x})\ \{\lambda y.\ [y = 2x]\}$$

Correct but useless:

$$\forall x. \quad \{[\,]\}\ (\texttt{next x})\ \{\lambda y.\ [y = 2x] * \exists r.\, (r \hookrightarrow 2x)\}$$

Summary

Separation Logic triple:

$$\{H\}\ t\ \{\lambda y.\, H'\} \qquad \text{and, more generally:} \quad \{H\}\ t\ \{Q\}$$

Typical specification of a function:

$$\forall x.\ \forall \ldots. \quad \{H\}\ (f\ x)\ \{\lambda y.\, H'\}$$

Specification of primitive functions:

$$\begin{array}{ll} \forall v. & \{[\,]\}\ (\texttt{ref v})\ \{\lambda r.\ (r \hookrightarrow v)\} \\ \forall r v. & \{r \hookrightarrow v\}\ (\texttt{!r})\ \{\lambda x.\ [x = v] * (r \hookrightarrow v)\} \\ \forall r v v'. & \{r \hookrightarrow v'\}\ (\texttt{r := v})\ \{\lambda tt.\, (r \hookrightarrow v)\} \end{array}$$

Definition of Hprop

Let Hprop denote the type of heap predicates:

$$\text{Hprop} \quad \equiv \quad \text{Heap} \to \text{Prop}$$

For example, in "$\{H\}\ t\ \{Q\}$" we have $(H : \text{Hprop})$ and $(Q : A \to \text{Hprop})$.

The separation algebra

The operator $(*)$ has type "Hprop → Hprop → Hprop" and satifisfies:

associativity:	$(H_1 * H_2) * H_3 = H_1 * (H_2 * H_3)$
commutativity:	$H_1 * H_2 = H_2 * H_1$
neutral element:	$H * [\,] = H$
scope extrusion:	$(\exists x.\, H_1) * H_2 = \exists x.\, (H_1 * H_2)$ (if $x \notin H_2$)

These proofs rely on predicate extensionality:

$$(\forall x.\, P\, x \Leftrightarrow Q\, x) \ \Rightarrow\ (P = Q)$$

which entails:

$$(\forall m.\, H\, m \Leftrightarrow H'\, m) \ \Leftrightarrow\ (H = H')$$

Definition of heap implication

If a heap satisfies "$(r \hookrightarrow 6)$" then it also satisfies "$\exists n.\, (r \hookrightarrow n) * [\text{even}\, n]$".

Definition of heap implication (relation of type Hprop → Hprop → Prop):

$$H_1 \rhd H_2 \quad \equiv \quad \forall m.\, H_1\, m \;\Rightarrow\; H_2\, m$$

For example, we have $(r \hookrightarrow 6) \rhd \exists n.\, (r \hookrightarrow n) * [\text{even}\, n]$.

Heap implication as a partial order

Our goal is to manipulate only heap predicates, never explicit heaps.
Thus, we should never unfold the definition of $(\rhd)$ in the proofs.

$$H_1 \rhd H_2 \quad \equiv \quad \forall m.\, H_1\, m \;\Rightarrow\; H_2\, m$$

The relation $(\rhd)$ defines a partial order on the type Hprop:

reflexivity

$$\frac{}{H \rhd H}$$

transitivity

$$\frac{H_1 \rhd H_2 \qquad H_2 \rhd H_3}{H_1 \rhd H_3}$$

antisymmetry

$$\frac{H_1 \rhd H_2 \qquad H_2 \rhd H_1}{H_1 = H_2}$$

Regularity property

Moreover, the relation $(\rhd)$ is regular w.r.t. the operator $(*)$.

$$\frac{H_1 \rhd H_1' \qquad H_2 \rhd H_2'}{H_1 * H_2 \rhd H_1' * H_2'}$$

This regularity rule entails a cancellation rule:

$$\frac{H_2 \;\rhd H_2'}{H_1 * H_2 \;\rhd H_1 * H_2'}$$

For example, to prove:

$$(r \hookrightarrow 2) * (s \hookrightarrow 3) \;\rhd\; (r \hookrightarrow 2) * (t \hookrightarrow n)$$

it suffices to prove:

$$(s \hookrightarrow 3) \;\rhd\; (t \hookrightarrow n)$$

Simplifications in heap implications

Consider:

$$(r \hookrightarrow 6) * [\,] * (s \hookrightarrow 3) \quad \rhd \quad \exists n.\, (s \hookrightarrow 3) * (r \hookrightarrow n) * [\text{even}\, n]$$

By the algebra laws, it is equivalent to prove:

$$(s \hookrightarrow 3) * (r \hookrightarrow 6) \quad \rhd \quad (s \hookrightarrow 3) * \exists n.\, (r \hookrightarrow n) * [\text{even}\, n]$$

By the cancellation rule, it suffices to prove:

$$(r \hookrightarrow 6) \quad \rhd \quad \exists n.\, (r \hookrightarrow n) * [\text{even}\, n]$$

Heap implications: true or false?

1.	$(r \hookrightarrow 3) * (s \hookrightarrow 4) \rhd (s \hookrightarrow 4) * (r \hookrightarrow 3)$	true
2.	$(r \hookrightarrow 3) \rhd (s \hookrightarrow 4) * (r \hookrightarrow 3)$	false
3.	$(s \hookrightarrow 4) * (r \hookrightarrow 3) \rhd (r \hookrightarrow 4)$	false
4.	$(s \hookrightarrow 4) * (r \hookrightarrow 3) \rhd (r \hookrightarrow 3)$	false
5.	$[\mathsf{False}] * (r \hookrightarrow 3) \rhd (s \hookrightarrow 4) * (r \hookrightarrow 4)$	true
6.	$(r \hookrightarrow 4) * (s \hookrightarrow 3) \rhd [\mathsf{False}]$	false
7.	$(r \hookrightarrow 4) * (r \hookrightarrow 3) \rhd [\mathsf{False}]$	true
8.	$(r \hookrightarrow 3) * (r \hookrightarrow 3) \rhd [\mathsf{False}]$	true
9.	$\exists n.\, (r \hookrightarrow n) \rhd (r \hookrightarrow 3)$	false
10.	$\exists n.\, (r \hookrightarrow n) \rhd (r \hookrightarrow n)$	ill-formed
11.	$(r \hookrightarrow 3) \rhd \exists n.\, (r \hookrightarrow n)$	true
12.	$(r \hookrightarrow 3) * (s \hookrightarrow 3) \rhd \exists n.\, (r \hookrightarrow n) * (s \hookrightarrow n)$	true
13.	$\exists n.\, (r \hookrightarrow n) * [n > 0] * [n < 0] \rhd [\mathsf{False}]$	true

Instantiation of existentials

To prove $(r \hookrightarrow 6) \rhd (\exists n.\, (r \hookrightarrow n) * [\mathsf{even}\, n])$, it suffices to exhibit an even number n such that $(r \hookrightarrow n)$.

Corresponding reasoning rules:

$$\frac{H_1 \rhd (\langle v/x \rangle\, H_2)}{H_1 \rhd (\exists x.\, H_2)} \qquad \frac{(H_1 \rhd H_2) \qquad P}{H_1 \rhd (H_2 * [P])}$$

Example:

$$\dfrac{\dfrac{\dfrac{\overline{(r \hookrightarrow 6) \rhd (r \hookrightarrow 6)} \qquad \overline{\mathsf{even}\, 6}}{(r \hookrightarrow 6) \rhd (r \hookrightarrow 6) * [\mathsf{even}\, 6]}}{(r \hookrightarrow 6) \rhd \langle 6/n \rangle ((r \hookrightarrow n) * [\mathsf{even}\, n])}}{(r \hookrightarrow 6) \rhd \exists n.\, (r \hookrightarrow n) * [\mathsf{even}\, n]}$$

Extraction of existentials

To prove $(\exists n.\, [\mathsf{even}\, n] * (r \hookrightarrow n)) \rhd (\exists m.\, [\mathsf{odd}\, m] * (r \hookrightarrow m+1))$, it suffices to prove that, for any even number n, we have

$$(r \hookrightarrow n) \rhd \exists m.\, [\mathsf{odd}\, m] * (r \hookrightarrow m+1)$$

Corresponding reasoning rules:

$$\frac{\forall x.\ (H_1 \rhd H_2)}{(\exists x.\, H_1) \rhd H_2} \qquad \frac{P \Rightarrow (H_1 \rhd H_2)}{([P] * H_1) \rhd H_2}$$

Same with explicit proof contexts:

$$\frac{\Gamma,\ x : A \ \vdash\ H_1 \rhd H_2}{\Gamma \ \vdash\ (\exists (x : A).\, H_1) \rhd H_2}\ {\scriptstyle (x \notin H_2)} \qquad \frac{\Gamma,\ h : P \ \vdash\ H_1 \rhd H_2}{\Gamma \ \vdash\ ([P] * H_1) \rhd H_2}$$

Extraction and instantiation: an example

Exercise: prove the following heap implication.

$$\exists n.\, [n \neq 2 \wedge \mathsf{prime}\, n] * (r \hookrightarrow n) \quad \rhd \quad \exists k.\, (r \hookrightarrow (2k+1))$$

Hint: always apply extraction rules before instantiations rules.
Hint: use the abbreviations $A_n \equiv (r \hookrightarrow n)$ and $B_k \equiv (r \hookrightarrow (2k+1))$ and $\Gamma \equiv (n : \mathsf{int},\ p : \mathsf{int},\ n = 2p+1)$.

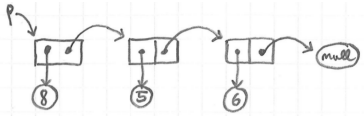
$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\overline{n : \mathsf{int} \ \vdash\ (r \hookrightarrow n) \ \rhd\ (r \hookrightarrow n)}}{n : \mathsf{int},\ p : \mathsf{int},\ n = 2p+1 \ \vdash\ (r \hookrightarrow n) \ \rhd\ (r \hookrightarrow 2p+1)}}{n : \mathsf{int},\ p : \mathsf{int},\ n = 2p+1 \ \vdash\ (r \hookrightarrow n) \ \rhd\ \langle p/k \rangle (r \hookrightarrow 2k+1)}}{n : \mathsf{int},\ p : \mathsf{int},\ n = 2p+1 \ \vdash\ (r \hookrightarrow n) \ \rhd\ \exists k.\, (r \hookrightarrow 2k+1)}}{n : \mathsf{int},\ n \neq 2,\ \mathsf{prime}\, n \ \vdash\ (r \hookrightarrow n) \ \rhd\ \exists k.\, (r \hookrightarrow 2k+1)}}{\varnothing \ \vdash\ \exists n.\, [n \neq 2 \wedge \mathsf{prime}\, n] * (r \hookrightarrow n) \ \rhd\ \exists k.\, (r \hookrightarrow 2k+1)}$$

Summary

- $(*)$ is associative, commutative, and has $[\,]$ as neutral element.
- $(\rhd)$ is a partial order, regular w.r.t. $(*)$.
- "[False] $\rhd H$" is always true.
- "$(r \hookrightarrow n) * (r \hookrightarrow m)$" is equivalent to "[False]".
- Apply rules in order: (1) extraction (2) instantiation (3) cancellation.

Null-terminated lists

Goal: reasoning about C-style mutable lists, in terms of pure lists.



$p \rightsquigarrow \mathsf{Mlist}\, L$ where $L = 8 :: 5 :: 6 :: \mathsf{nil}$

Null pointers in OCaml

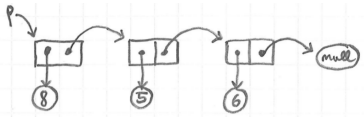
Interface:

```
val null : 'a
```

Example:

```
let x = null in
let y = ref 3 in
if x != y then ...
if x == y then ...
```


Construction of mutable lists



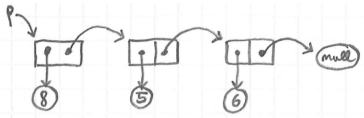
Data type for list cells:

```
type 'a cell = { mutable hd : 'a;
                 mutable tl : 'a cell }
```

Construction of the example list:

```
let p0 = null in
let p1 = { hd = 6; tl = p0 } in
let p2 = { hd = 5; tl = p1 } in
let p = { hd = 8; tl = p2 } in
p
```

Representation of mutable lists



The goal is to define a heap predicate for mutable lists:

$$p \rightsquigarrow \mathsf{Mlist}\, L$$

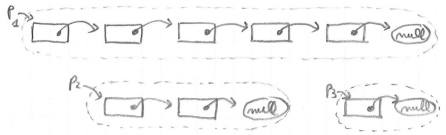
where $(p : \mathsf{loc})$ and $(L : \mathsf{list}\, A)$ for some type A.

The definition satisfies, e.g.:

$$p \rightsquigarrow \mathsf{Mlist}\,(8 :: 5 :: 6 :: \mathsf{nil}) \quad \equiv \quad \exists p_2 p_1.\ \ p \mapsto \{|\mathsf{hd}{=}8;\ \mathsf{tl}{=}p_2|\} * p_2 \mapsto \{|\mathsf{hd}{=}5;\ \mathsf{tl}{=}p_1|\} * p_1 \mapsto \{|\mathsf{hd}{=}6;\ \mathsf{tl}{=}\mathsf{null}|\}$$

Strength of the representation predicate

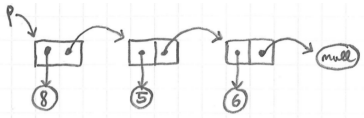
$$p_1 \rightsquigarrow \mathsf{Mlist}\, L_1 \ * \ p_2 \rightsquigarrow \mathsf{Mlist}\, L_2 \ * \ p_3 \rightsquigarrow \mathsf{Mlist}\, L_3$$



In Separation Logic, Mlist enforces:

- no cycles,
- no sharing,
- order of the items as specified by the pure lists.

Use of the representation predicate

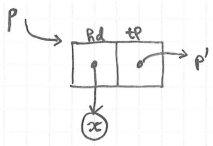


Specification of `mlength`, which computes the length of a mutable list:

$$\forall pL.\quad \{p \rightsquigarrow \mathsf{Mlist}\, L\}\ (\texttt{mlength p})\ \{\lambda n.\ [n = \mathsf{length}\, L] * p \rightsquigarrow \mathsf{Mlist}\, L\}$$

More generally, the heap predicate Mlist is used to specify functions on mutable lists in terms of corresponding operations on pure lists.

Heap predicate for list cells



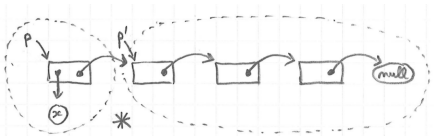
Specification of a list cell:

$$p \mapsto \{|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\}$$

Definition of Mlist

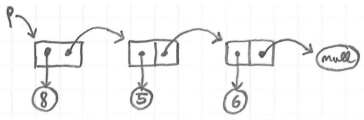
$$p \rightsquigarrow \mathsf{Mlist}\, L \;\equiv\; \mathsf{match}\, L\, \mathsf{with} \mid \mathsf{nil} \Rightarrow [p = \mathsf{null}] \mid x :: L' \Rightarrow \exists p'.\ p \mapsto \{\!|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\!\} * p' \rightsquigarrow \mathsf{Mlist}\, L'$$

Illustration of the cons case:



Iterated unfolding of Mlist

$$p \rightsquigarrow \mathsf{Mlist}\, L \;\equiv\; \mathsf{match}\, L\, \mathsf{with} \mid \mathsf{nil} \Rightarrow [p = \mathsf{null}] \mid x :: L' \Rightarrow \exists p'.\ p \mapsto \{\!|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\!\} * p' \rightsquigarrow \mathsf{Mlist}\, L'$$



$$p \rightsquigarrow \mathsf{Mlist}\,(8 :: 5 :: 6 :: \mathsf{nil}) \;\equiv\; \exists p_1.\ p \mapsto \{\!|\mathsf{hd}{=}8;\ \mathsf{tl}{=}p_1|\!\} * \exists p_2.\ p_1 \mapsto \{\!|\mathsf{hd}{=}5;\ \mathsf{tl}{=}p_2|\!\} * \exists p_3.\ p_2 \mapsto \{\!|\mathsf{hd}{=}6;\ \mathsf{tl}{=}p_3|\!\} * [p_3 = \mathsf{null}]$$

Characterisation of empty lists

$$p \rightsquigarrow \mathsf{Mlist}\, L \;\equiv\; \mathsf{match}\, L\, \mathsf{with} \mid \mathsf{nil} \Rightarrow [p = \mathsf{null}] \mid x :: L' \Rightarrow \exists p'.\ p \mapsto \{\!|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\!\} * p' \rightsquigarrow \mathsf{Mlist}\, L'$$

In "$p \rightsquigarrow \mathsf{Mlist}\, L$", if p is null or L is nil, then the mutable list is empty.

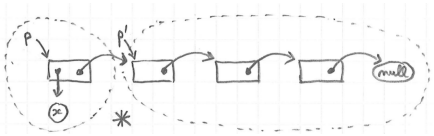
$$p \rightsquigarrow \mathsf{Mlist}\,\mathsf{nil} \;\rhd\; [p = \mathsf{null}]$$
$$\mathsf{null} \rightsquigarrow \mathsf{Mlist}\, L \;\rhd\; [L = \mathsf{nil}]$$

Reciprocally, a null pointer denotes an empty list in the empty heap:

$$[\,] \;\rhd\; \mathsf{null} \rightsquigarrow \mathsf{Mlist}\,\mathsf{nil}$$

Focus and defocus rules for non-empty lists

$$p \rightsquigarrow \mathsf{Mlist}\, L \;\equiv\; \mathsf{match}\, L\, \mathsf{with} \mid \mathsf{nil} \Rightarrow [p = \mathsf{null}] \mid x :: L' \Rightarrow \exists p'.\ p \mapsto \{\!|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\!\} * p' \rightsquigarrow \mathsf{Mlist}\, L'$$



$$p \rightsquigarrow \mathsf{Mlist}\, L * [p \neq \mathsf{null}] \;\rhd\; \exists x L' p'.\ [L = x :: L'] * p \mapsto \{\!|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\!\} * p' \rightsquigarrow \mathsf{Mlist}\, L'$$

$$p \mapsto \{\!|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\!\} * p' \rightsquigarrow \mathsf{Mlist}\, L' \;\rhd\; p \rightsquigarrow \mathsf{Mlist}\,(x :: L')$$

Verification of mlength

```
let rec mlength (p:'a cell) =
  if p == null
    then 0
    else 1 + mlength p.tl
```

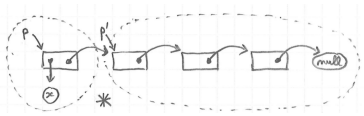
Specification:

$\forall pL.\quad \{p \rightsquigarrow \mathsf{Mlist}\, L\}\ (\texttt{mlength p})\ \{\lambda n.\ [n = \mathsf{length}\, L] * p \rightsquigarrow \mathsf{Mlist}\, L\}$

Verification in the case $p =$ null
The code returns $n = 0$. To prove that $n = \mathsf{length}\, L$, we need to show $L = \mathsf{nil}$. Since $p = \mathsf{null}$, the pre-condition is "$\mathsf{null} \rightsquigarrow \mathsf{Mlist}\, L$", which entails $L = \mathsf{nil}$.

Verification of mlength, cons case

$\forall pL.\quad \{p \rightsquigarrow \mathsf{Mlist}\, L\}\ (\texttt{mlength p})\ \{\lambda n.\ [n = \mathsf{length}\, L] * p \rightsquigarrow \mathsf{Mlist}\, L\}$



Verification in the case $p \neq$ null
In the pre-condition, the list L decomposes as $x :: L'$ and there exists p' such that the state is: $p \mapsto \{|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\} * p' \rightsquigarrow \mathsf{Mlist}\, L'$.

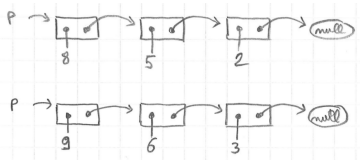
The code returns $n =$ `1 + mlength p'`. We need to show $n = \mathsf{length}\, L$.
Note that $\mathsf{length}\, L = 1 + \mathsf{length}\, L'$.
Thus, it suffices to show that `mlength p'` returns $\mathsf{length}\, L'$.

This result follows from the induction hypothesis:
$\{p' \rightsquigarrow \mathsf{Mlist}\, L'\}\ (\texttt{mlength p'})\ \{\lambda n.\ [n = \mathsf{length}\, L'] * p' \rightsquigarrow \mathsf{Mlist}\, L'\}$

Verification of mlength, frame

Precondition:
$$p \rightsquigarrow \mathsf{Mlist}\, L$$
Focus rule:
$$p \mapsto \{|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\} * p' \rightsquigarrow \mathsf{Mlist}\, L'$$
Recursive call on:
$$p' \rightsquigarrow \mathsf{Mlist}\, L'$$
Restore the *framed* cell:
$$p \mapsto \{|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\} * p' \rightsquigarrow \mathsf{Mlist}\, L'$$
Defocus rule:
$$p \rightsquigarrow \mathsf{Mlist}\, L$$

Specification of in-place increment




```
let rec list_incr (p:'a cell) =
  if p != null then begin
    p.hd <- p.hd + 1;
    list_incr p.tl
  end
```

Exercise: what is the specification of `list_incr`? Hint: use List.map.

$\{p \rightsquigarrow \mathsf{Mlist}\, L\}\ (\texttt{list_incr p})\ \{\lambda tt.\ p \rightsquigarrow \mathsf{Mlist}\, (\mathsf{map}\, (+1)\, L)\}$

Verification of in-place increment, frame

Precondition:

$$p \rightsquigarrow \mathsf{Mlist}\, L$$

Focus rule:

$$p \mapsto \{|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\} \ast p' \rightsquigarrow \mathsf{Mlist}\, L'$$

Increment at head:

$$p \mapsto \{|\mathsf{hd}{=}x+1;\ \mathsf{tl}{=}p'|\} \ast p' \rightsquigarrow \mathsf{Mlist}\, L'$$

Recursive call on:

$$p' \rightsquigarrow \mathsf{Mlist}\, L'$$

Produces:

$$p' \rightsquigarrow \mathsf{Mlist}\, (\mathsf{map}\, (+1)\, L')$$

Restore the *framed* cell:

$$p \mapsto \{|\mathsf{hd}{=}x+1;\ \mathsf{tl}{=}p'|\} \ast p' \rightsquigarrow \mathsf{Mlist}\, (\mathsf{map}\, (+1)\, L')$$

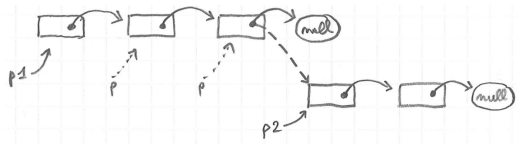
Defocus rule:

$$p \rightsquigarrow \mathsf{Mlist}\, ((x+1) :: (\mathsf{map}\, (+1)\, L'))$$

Rewrites as:

$$p \rightsquigarrow \mathsf{Mlist}\, (\mathsf{map}\, (+1)\, L)$$

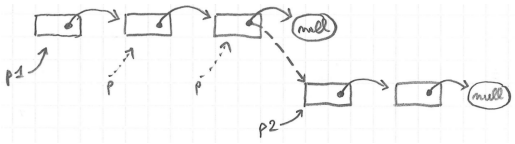
The function mappend




```
let mappend (p1:'a cell) (p2:'a cell) =
  if p1 == null then p2 else
  let rec aux p =
    if p.tl == null
      then p.tl <- p2
      else aux p.tl
    in
  aux p1;
  p1
```

What happens if `p1` and `p2` denote the same list?
What is the specification of `mappend`? and that of `aux`?

Specification of mappend



Specification of `mappend`:

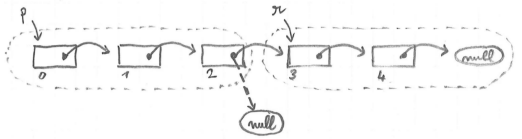
$$\begin{array}{ll} \forall p_1 p_2 L_1 L_2. & \{\ p_1 \rightsquigarrow \mathsf{Mlist}\, L_1 \ast p_2 \rightsquigarrow \mathsf{Mlist}\, L_2\ \} \\ & (\texttt{mappend p1 p2}) \\ & \{\ \lambda q.\ q \rightsquigarrow \mathsf{Mlist}\, (L_1 +\!\!+ L_2)\ \} \end{array}$$

Specification of `aux`:

$$\begin{array}{ll} \forall p L. & \{\ [p \neq \mathsf{null}] \ast p \rightsquigarrow \mathsf{Mlist}\, L \ast p_2 \rightsquigarrow \mathsf{Mlist}\, L_2\} \\ & (\texttt{aux p}) \\ & \{\ \lambda t.\ p \rightsquigarrow \mathsf{Mlist}\, (L +\!\!+ L_2)\ \} \end{array}$$

The function split-after

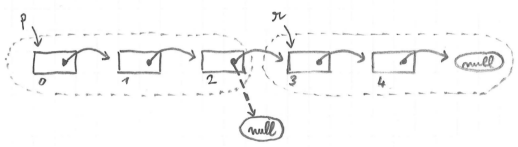
Splits a list p after the item at index i, and returns the chopped sublist:




```
let split_after (i:int) (p:'a cell) =
  if i == 0 then begin
    let r = p.tl in
    p.tl <- null
    r
  end else begin
    split_after (i-1) p.tl
  end
```

Exercise: specification? Hint: introduce L_1 and L_2 to denote the sublists.

Specification of split-after

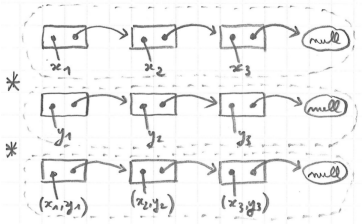


Specification of `split_after`:

$$\begin{array}{l}\forall ipL.\\ \quad \{p \rightsquigarrow \mathsf{Mlist}\, L \ast [0 \leqslant i < |L|]\}\\ \quad (\texttt{split_after i p})\\ \quad \{\lambda r.\ \exists L_1 L_2.\ p \rightsquigarrow \mathsf{Mlist}\, L_1 \ast r \rightsquigarrow \mathsf{Mlist}\, L_2\\ \qquad \ast [L = L_1 +\!\!+ L_2 \wedge |L_1| = i + 1]\}\end{array}$$

Exercises

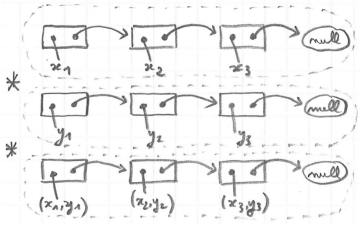
Exercise: the function mcombine




```
let rec mcombine (p1:'a cell) (p2:'a cell) =
  if p1 == null then null else
  let h = (p1.hd, p2.hd) in
  let t = mcombine p1.tl p2.tl in
  { hd = h; tl = t }
```

Specify `mcombine` in terms of `List.combine`, which, given two pure lists $x_1, x_2, ..., x_n$ and $y_1, y_2, ..., y_n$ returns $(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)$.

Solution: the function mcombine



Specification of `mcombine`:

$$\begin{array}{l}\forall p_1 p_2 L_1 L_2.\\ \quad \{p_1 \rightsquigarrow \mathsf{Mlist}\, L_1 \ast p_2 \rightsquigarrow \mathsf{Mlist}\, L_2 \ast [\mathsf{length}\, L_1 = \mathsf{length}\, L_2]\}\\ \quad (\texttt{mcombine p1 p2})\\ \quad \{\lambda q.\ q \rightsquigarrow \mathsf{Mlist}\, (\mathsf{combine}\, L_1\, L_2) \ast p_1 \rightsquigarrow \mathsf{Mlist}\, L_1 \ast p_2 \rightsquigarrow \mathsf{Mlist}\, L_2\}\end{array}$$

Exercise: factorial with a while loop

```
let facto n =
  let r = ref 1 in
  let k = ref 2 in
  while !k <= n do
    r := !r * !k;
    incr k;
  done;
  !r
```

1. Specify the state before and after the loop.
2. State the loop invariant.
3. Prove that the state before the loop entails the loop invariant.
4. Prove that, when the loop condition is false, the loop invariant entails the state after the loop.

Solution: factorial with a while loop

Before the loop:

$$k \hookrightarrow 2 \;*\; r \hookrightarrow 1$$

After the loop:

$$k \hookrightarrow (n+1) \;*\; r \hookrightarrow n!$$

Loop invariant:

$$\exists i.\;\; k \hookrightarrow i \;*\; r \hookrightarrow (i-1)! \;*\; [2 \leqslant i \leqslant n+1]$$

Entering the loop:

$$i = 2$$

Exiting the loop:

$$\neg(i \leqslant n) \quad \wedge \quad (2 \leqslant i \leqslant n+1) \quad \Rightarrow \quad i = n+1$$

Exercise: function exchange

Specify `exchange` and show that the program prints 0.

```
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: exchange

Program:

```
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 b' = exchange a' in
!(!r)
```

Specification of exchange:

$$\forall ts.\quad \{r \hookrightarrow t\}\ (\texttt{exchange s})\ \{\lambda x.\,[x = t] * r \hookrightarrow s\}$$

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

Exercise: proof of a heap implication

Prove the following heap implication:

$$\exists k.\, (r \hookrightarrow 4k) \quad \rhd \quad \exists n.\, (r \hookrightarrow n) * [\mathsf{even}\, n]$$

Remark: "$\mathsf{even}\, n$" is defined as "$\exists m.\, n = 2m$".

Solution: proof of a heap implication

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{(r \hookrightarrow 4k) \rhd (r \hookrightarrow 4k)} \qquad \dfrac{}{\mathsf{even}\,(4k)}}{(r \hookrightarrow 4k) \rhd (r \hookrightarrow 4k) * [\mathsf{even}\, 4k]}}{(r \hookrightarrow 4k) \rhd \langle 4k/n \rangle ((r \hookrightarrow n) * [\mathsf{even}\, n])}}{\forall k. \quad (r \hookrightarrow 4k) \rhd \exists n.\, (r \hookrightarrow n) * [\mathsf{even}\, n]}}{\exists k.\, (r \hookrightarrow 4k) \rhd \exists n.\, (r \hookrightarrow n) * [\mathsf{even}\, n]}$$

Exercise: proof of commutativity of star

Prove that the separating conjunction is commutative:

$$H_1 * H_2 = H_2 * H_1$$

Hint: unfold the definition of $(*)$ and apply predicate extensionality:

$$(\forall x.\, P\, x \Leftrightarrow Q\, x) \;\Rightarrow\; (P = Q).$$

Solution: proof of commutativity

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

By predicate extensionality, it suffices to prove:

$$\forall m. \quad (H_1 * H_2)\, m \quad \Leftrightarrow \quad (H_2 * H_1)\, m$$

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:

$$m_1 \perp m_2 \;\wedge\; m = m_1 \uplus m_2 \;\wedge\; H_1\, m_1 \;\wedge\; H_2\, m_2$$

The goal that we have to prove is equivalent to:

$$\exists m_1' m_2'. \quad m_1' \perp m_2' \;\wedge\; m = m_1' \uplus m_2' \;\wedge\; H_2\, m_1' \;\wedge\; H_2\, m_2'$$

We conclude by instantiating m_1' as m_2 and m_2' as m_1.