Separation Logic
1/4

Arthur Charguéraud

January 25th, 2016

Chapter 1

Separation Logic Operators

Heap predicates

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

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.

Primitive heap predicates:

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

Empty heap and pure facts

Definition:

$$\begin{array}{lcl} [\,] & \equiv & \lambda m.\ m = \varnothing \\ [P] & \equiv & \lambda m.\ m = \varnothing \ \wedge\ P \end{array}$$

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

$$\begin{array}{ll} \text{Before:} & [\,] \\ \text{After:} & [a = 3 \ \wedge\ b = 4] \end{array}$$

Observe that $[\,]$ is equivalent to $[\text{True}]$.

Singleton heap

Definition:

$$l \mapsto v \quad \equiv \quad \lambda m.\ m = \{(l, v)\} \wedge l \neq \textsf{null}$$

Example: specification of "`let r = ref 3`".

$$\begin{array}{ll} \text{Before:} & [\,] \\ \text{After:} & r \mapsto 3 \end{array}$$

Example: specification of "`incr s`".

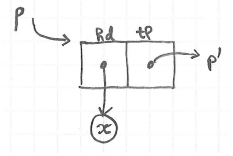
$$\begin{array}{lll} \text{Before:} & s \mapsto n & \text{for some } n \\ \text{After:} & s \mapsto (n+1) & \end{array}$$

Record fields

Heap predicate describing the field f of a record at address p:

$$p.\mathsf{f} \mapsto v$$

Example:



$$p.\mathsf{hd} \mapsto x$$

$$p.\mathsf{tl} \mapsto p'$$

In the C memory model:

$$p.\mathsf{f} \mapsto v \;\equiv\; (p + \mathsf{f}) \mapsto v$$

with $\quad \mathsf{hd} \equiv 0 \quad \text{and} \quad \mathsf{tl} \equiv 1$

Separating conjunction

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

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

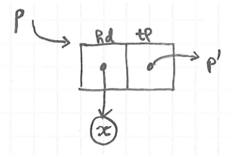
Definition:

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

where:

$$\begin{array}{lcll} m_1 \perp m_2 & \equiv & \mathsf{dom}\, m_1 \cap \mathsf{dom}\, m_2 = \varnothing & \\ m_1 \uplus m_2 & \equiv & m_1 \cup m_2 & \text{when } m_1 \perp m_2 \end{array}$$

Representation of list cells



$$p \rightsquigarrow \{\!|\,\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'\,|\!\} \quad \equiv \quad p.\mathsf{hd} \mapsto x \ \star\ p.\mathsf{tl} \mapsto p'$$

Or simply: $p \rightsquigarrow \{\!|x, p'|\!\}$

Remark: the new arrow symbol will be overloaded later.

Separating conjunction: aliasing

Exercise:

1. specify: `let r = ref 5 and s = ref 3 and t = r`.
2. specify the state after subsequently executing: `incr r`.
3. specify the state after subsequently executing: `incr t`.

Incorrect answer: $(r \mapsto 5) \star (s \mapsto 3) \star (t \mapsto 5)$.

Correct answer:

1. $(r \mapsto 5) \star (s \mapsto 3) \star [t = r]$
2. $(r \mapsto 6) \star (s \mapsto 3) \star [t = r]$
3. $(r \mapsto 7) \star (s \mapsto 3) \star [t = r]$

Existential quantification

Definition:

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

Compare:

$$\begin{array}{lcll} (\exists x.\, P) & : & \mathsf{Prop} & \text{when} \quad (P : \mathsf{Prop}) \\ (\exists x.\, H) & : & \mathsf{Heap} \to \mathsf{Prop} & \text{when} \quad (H : \mathsf{Heap} \to \mathsf{Prop}) \end{array}$$

Summary

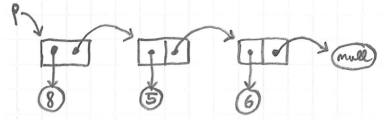
$$
\begin{array}{lcl}
[\,] & \equiv & [\mathsf{True}] \\
[P] & \equiv & \lambda m.\ m = \varnothing \ \wedge\ P \\
l \mapsto v & \equiv & \lambda m.\ m = \{(l, v)\} \\
H_1 \star H_2 & \equiv & \lambda m.\ \exists m_1 m_2.\ \left\{ \begin{array}{l} m_1 \perp m_2 \\ m = m_1 \uplus m_2 \\ H_1\, m_1 \\ H_2\, m_2 \end{array} \right. \\
\exists x.\, H & \equiv & \lambda m.\ \exists x.\ H\, m
\end{array}
$$

Chapter 2

Representation Predicate for Lists

Implementation of mutable lists

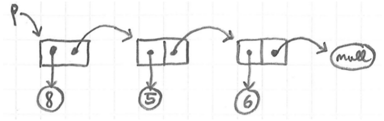
Mutable lists (C-style), expressed in OCaml extended with null pointers.



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

{ hd = 8; tl = { hd = 5; tl = { hd = 6; tl = null } } }
```

Representation of mutable lists

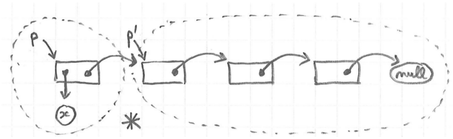


$$L \;=\; 8 :: 5 :: 6 :: \mathsf{nil}$$

$$
\begin{array}{rcl}
p \rightsquigarrow \mathsf{Mlist}\, L & \equiv & \quad \exists p_1.\;\; p \rightsquigarrow \{\!|\mathsf{hd}{=}8;\ \mathsf{tl}{=}p_1|\!\} \\
& & \star\ \exists p_2.\;\; p_1 \rightsquigarrow \{\!|\mathsf{hd}{=}5;\ \mathsf{tl}{=}p_2|\!\} \\
& & \star\ \exists p_3.\;\; p_2 \rightsquigarrow \{\!|\mathsf{hd}{=}6;\ \mathsf{tl}{=}p_3|\!\} \\
& & \star\ [p_3 = \mathsf{null}]
\end{array}
$$

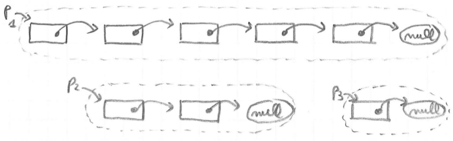
Remark: in Coq, $p \rightsquigarrow \mathsf{Mlist}\, L$ is just a convenient notation for $\mathsf{Mlist}\, L\, p$.

Representation predicate



$$
\begin{aligned}
p \rightsquigarrow \mathsf{Mlist}\, L \quad \equiv \quad & \mathsf{match}\, L\, \mathsf{with} \\
& \mid \mathsf{nil} \;\Rightarrow\; [p = \mathsf{null}] \\
& \mid x :: L' \;\Rightarrow\; \exists p'.\quad p \rightsquigarrow \{\!|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\!\} \\
& \qquad\qquad\qquad\quad \star\; p' \rightsquigarrow \mathsf{Mlist}\, L'
\end{aligned}
$$

Separation properties



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

Separation enforces: no cycles, and no sharing.

Union heap predicate

$$
\begin{array}{rcl}
p \leadsto \mathsf{Mlist}\, L & \equiv & \mathsf{match}\, L\, \mathsf{with} \\
& & \mid \mathsf{nil} \;\Rightarrow\; [p = \mathsf{null}] \\
& & \mid x :: L' \;\Rightarrow\; \exists p'.\quad p \leadsto \{\!|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\!\} \\
& & \qquad\qquad\qquad\quad \star\ p' \leadsto \mathsf{Mlist}\, L'
\end{array}
$$

Equivalent to:

$$
\begin{array}{rcl}
p \leadsto \mathsf{Mlist}\, L & \equiv & \quad [L = \mathsf{nil} \;\wedge\; p = \mathsf{null}] \\
& & \Cup \; \big(\exists x L' p'.\ [L = x :: L'] \\
& & \qquad\qquad \star\ p \leadsto \{\!|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\!\} \\
& & \qquad\qquad \star\ p' \leadsto \mathsf{Mlist}\, L' \quad \big)
\end{array}
$$

where:

$$
H_1 \Cup H_2 \quad \equiv \quad \lambda m.\ H_1\, m \;\vee\; H_2\, m
$$

In-place list reversal: code

```
let reverse p0 =
  let r = ref p0 in
  let s = ref null in
  while !r <> null do
    let p = !r in
    r := p.tl;
    p.tl <- !s;
    s := p;
  done;
  !s
```

Exercise:

1. Specify the state before the loop.
2. Specify the state after the loop.
3. Specify the loop invariant.

In-place list reversal: invariants

Before the loop:

$$r \mapsto p_0 \star s \mapsto \mathsf{null} \star p_0 \rightsquigarrow \mathsf{Mlist}\, L$$

After the loop:

$$\exists q.\ r \mapsto \mathsf{null} \star s \mapsto q \star q \rightsquigarrow \mathsf{Mlist}\,(\mathsf{rev}\, L)$$

Loop invariant:

$$\begin{aligned} \exists p q L_1 L_2.\quad & r \mapsto p \star p \rightsquigarrow \mathsf{Mlist}\, L_2 \\ & \star\ s \mapsto q \star q \rightsquigarrow \mathsf{Mlist}\, L_1 \\ & \star\ [L = \mathsf{rev}\, L_1 +\!\!+ L_2] \end{aligned}$$

In-place list reversal: proof (1/2)

Invariant:

$$\begin{aligned}\exists pqL_1L_2.\ & r \mapsto p \star s \mapsto q \\ & \star\ p \leadsto \mathsf{Mlist}\, L_2 \star q \leadsto \mathsf{Mlist}\, L_1 \\ & \star\ [L = \mathsf{rev}\, L_1 \mathbin{+\!\!+} L_2]\end{aligned}$$

Initial state implies the invariant: take $p = p_0$ and $L_1 = \mathsf{nil}$ and $L_2 = L$.

$$r \mapsto p_0 \star p_0 \leadsto \mathsf{Mlist}\, L \star s \mapsto \mathsf{null} \star \mathsf{null} \leadsto \mathsf{Mlist}\, \mathsf{nil} \star [L = \mathsf{rev}\, \mathsf{nil} \mathbin{+\!\!+} L]$$

Invariant implies the final state: exploit $p = \mathsf{null}$ to derive $L_2 = \mathsf{nil}$.

$$r \mapsto \mathsf{null} \star \mathsf{null} \leadsto \mathsf{Mlist}\, \mathsf{nil} \star s \mapsto q \star q \leadsto \mathsf{Mlist}\, L_1 \star [L = \mathsf{rev}\, L_1 \mathbin{+\!\!+} \mathsf{nil}]$$

Conversion rules for empty lists

$$\begin{array}{rcl} p \leadsto \mathsf{Mlist}\,L & \equiv & \mathsf{match}\ L\ \mathsf{with} \\ & & \mid \mathsf{nil} \Rightarrow [p = \mathsf{null}] \\ & & \mid x :: L' \Rightarrow \exists p'.\quad p \leadsto \{\!|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\!\} \\ & & \qquad\qquad\qquad \star\ p' \leadsto \mathsf{Mlist}\,L' \end{array}$$

$$\begin{array}{rcl} (p \leadsto \mathsf{Mlist}\,\mathsf{nil}) & = & [p = \mathsf{null}] \\ (\mathsf{null} \leadsto \mathsf{Mlist}\,L) & = & [L = \mathsf{nil}] \\ (\mathsf{null} \leadsto \mathsf{Mlist}\,\mathsf{nil}) & = & [\,] \end{array}$$

In-place list reversal: proof (2/2)

Transition when $p \neq \text{null}$:

$$p \rightsquigarrow \mathsf{Mlist}\, L_2 \star q \rightsquigarrow \mathsf{Mlist}\, L_1 \star [L = \mathsf{rev}\, L_1 +\!\!+ L_2]$$

to

$$\exists x L_2' p'.\quad [L_2 = x :: L_2'] \star p \rightsquigarrow \{|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\} \star p' \rightsquigarrow \mathsf{Mlist}\, L_2' \\ \star q \rightsquigarrow \mathsf{Mlist}\, L_1 \star [L = \mathsf{rev}\, L_1 +\!\!+ L_2]$$

After update of $p.\mathsf{tl}$ to the value q:

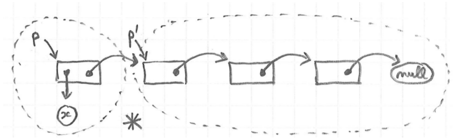
$$p \rightsquigarrow \{|\mathsf{hd}{=}x;\ \mathsf{tl}{=}q|\} \star q \rightsquigarrow \mathsf{Mlist}\, L_1 \\ \star p' \rightsquigarrow \mathsf{Mlist}\, L_2' \star [L = \mathsf{rev}\, L_1 +\!\!+ (x :: L_2')]$$

to

$$q \rightsquigarrow \mathsf{Mlist}\,(x :: \mathsf{rev}\, L_1) \star p' \rightsquigarrow \mathsf{Mlist}\, L_2' \star [L = \mathsf{rev}\,(x :: L_1) +\!\!+ L_2]$$

Conversion rules for nonempty lists

$$\begin{array}{lcl} p \rightsquigarrow \mathsf{Mlist}\,L & \equiv & \mathsf{match}\ L\ \mathsf{with} \\ & & \mid \mathsf{nil} \Rightarrow [p = \mathsf{null}] \\ & & \mid x :: L' \Rightarrow \exists p'.\quad p \rightsquigarrow \{\!|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\!\} \\ & & \qquad\qquad\qquad \star\ p' \rightsquigarrow \mathsf{Mlist}\,L' \end{array}$$



$$\begin{array}{lcll} p \rightsquigarrow \mathsf{Mlist}\,L \star [p \neq \mathsf{null}] & = & \exists x L' p'. & [L = x :: L'] \\ & & & \star\ p \rightsquigarrow \{\!|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\!\} \\ & & & \star\ p' \rightsquigarrow \mathsf{Mlist}\,L' \end{array}$$

Summary

$$
\begin{array}{rcl}
p \leadsto \mathsf{Mlist}\,L & \equiv & \mathsf{match}\,L\,\mathsf{with} \\
& & \mid \mathsf{nil} \;\Rightarrow\; [p = \mathsf{null}] \\
& & \mid x :: L' \;\Rightarrow\; \exists p'. \quad p \leadsto \{\!|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\!\} \\
& & \qquad\qquad\qquad\quad \star\ p' \leadsto \mathsf{Mlist}\,L'
\end{array}
$$

Chapter 3

Representation Predicate for List Segments

Length of a mutable list using a while loop

```
let rec mlength (p:'a cell) =
  let f = ref p in
  let t = ref 0 in
  while !f != null do
    incr t;
    f := (!f).tl;
  done
  !t
```

Exercise:

1. Specify the state before the loop.
2. Specify the state after the loop.
3. Draw a picture describing a state during the loop.
4. Try to state a loop invariant. What do you need?

Mlength: initial and final states

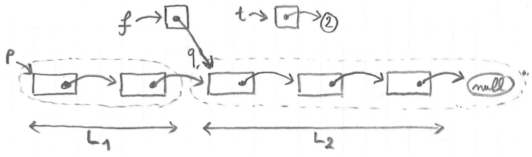
Before the loop:

$$(p \rightsquigarrow \mathsf{Mlist}\, L) \star (f \mapsto p) \star (t \mapsto 0)$$

After the loop:

$$(p \rightsquigarrow \mathsf{Mlist}\, L) \star (f \mapsto \mathsf{null}) \star (t \mapsto \mathsf{length}\, L)$$

Mlength: loop invariant

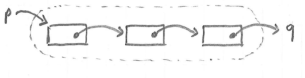


Loop invariant:

$$\exists L_1 L_2 q. \qquad [L = L_1 ++ L_2] \star (t \mapsto \mathsf{length}\, L_1) \star (f \mapsto q)$$
$$\star\ (p \rightsquigarrow \mathsf{MlistSeg}\, q\, L_1) \star (q \rightsquigarrow \mathsf{Mlist}\, L_2)$$

Representation predicate for list segments

$$
\begin{array}{lcl}
p \rightsquigarrow \mathsf{Mlist}\, L & \equiv & \mathsf{match}\ L\ \mathsf{with} \\
 & & \mid \mathsf{nil}\ \Rightarrow\ [p = \mathsf{null}] \\
 & & \mid x :: L'\ \Rightarrow\ \exists p'.\ p \rightsquigarrow \{\!|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\!\} \\
 & & \qquad\qquad\qquad \star\, p' \rightsquigarrow \mathsf{Mlist}\, L'
\end{array}
$$



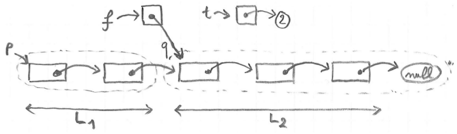
Exercise: generalize Mlist to define $p \rightsquigarrow \mathsf{MlistSeg}\, q\, L$, where L denotes the list of items in the list segment from p (inclusive) to q (exclusive).

$$
\begin{array}{lcl}
p \rightsquigarrow \mathsf{MlistSeg}\, q\, L & \equiv & \mathsf{match}\ L\ \mathsf{with} \\
 & & \mid \mathsf{nil}\ \Rightarrow\ [p = q] \\
 & & \mid x :: L'\ \Rightarrow\ \exists p'.\ p \rightsquigarrow \{\!|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\!\} \\
 & & \qquad\qquad\qquad \star\, p' \rightsquigarrow \mathsf{MlistSeg}\, q\, L'
\end{array}
$$

Remark:

$$p \rightsquigarrow \mathsf{Mlist}\, L \ = \ p \rightsquigarrow \mathsf{MlistSeg}\, \mathsf{null}\, L$$

Mlength: proof



Enter:

$$L_1 = \mathsf{nil} \ \wedge \ L_2 = L \ \wedge \ q = p$$

$$[\,] \ = \ (p \rightsquigarrow \mathsf{MlistSeg}\, p\, \mathsf{nil})$$

Exit:

$$L_1 = L \ \wedge \ L_2 = \mathsf{nil} \ \wedge \ q = \mathsf{null}$$

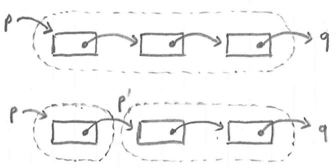
$$(p \rightsquigarrow \mathsf{MlistSeg}\, \mathsf{null}\, L) \ = \ (p \rightsquigarrow \mathsf{Mlist}\, L)$$

Step:

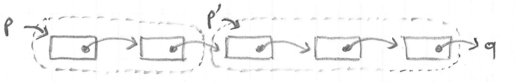
$$L_2 = x :: L_2' \ \wedge \ q \neq \mathsf{null} \ \wedge \ q.\mathsf{tl} = q'$$

$$\exists q.\ p \rightsquigarrow \mathsf{MlistSeg}\, q\, L_1 \ \star \ q \rightsquigarrow \{\!|\mathsf{hd}{=}x;\ \mathsf{tl}{=}q'|\!\}$$
$$= \quad p \rightsquigarrow \mathsf{MlistSeg}\, q'\, (L_1 +\!\!+\, x :: \mathsf{nil})$$

Splitting rules for list segments

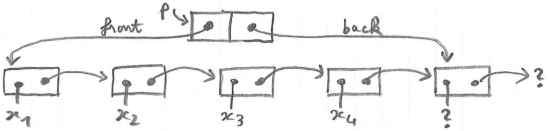


$$p \rightsquigarrow \mathsf{MlistSeg}\, q\, (x :: L') \;=\; \exists p'.\; p \rightsquigarrow \{\!|\mathsf{hd}{=}x;\, \mathsf{tl}{=}p'|\!\} \;\star\; p' \rightsquigarrow \mathsf{MlistSeg}\, q\, L'$$



$$\begin{aligned} p \rightsquigarrow \mathsf{MlistSeg}\, q\, (L_1 +\!\!+ L_2) \quad = \quad \exists p'. \quad & p \rightsquigarrow \mathsf{MlistSeg}\, p'\, L_1 \\ \star\; & p' \rightsquigarrow \mathsf{MlistSeg}\, q\, L_2 \end{aligned}$$

An implementation of mutable queues



Represent a queue as a list segment, with the last cell storing no item.

```
type 'a queue = {
  mutable front : 'a cell;
  mutable back : 'a cell; }
```

Exercise: define the representation predicate $p \rightsquigarrow \mathsf{Queue}\, L$.

$$
\begin{aligned}
p \rightsquigarrow \mathsf{Queue}\, L \;\equiv\; \exists f b. \quad & p \mapsto \{\!|\mathsf{front}{=}f;\ \mathsf{back}{=}b|\!\} \\
& \star\, f \rightsquigarrow \mathsf{MlistSeg}\, b\, L \\
& \star\, (b.\mathsf{hd} \mapsto -) \,\star\, (b.\mathsf{tl} \mapsto -)
\end{aligned}
$$

Alternative for the last cell: $\exists yq.\ b \mapsto \{\!|\mathsf{hd}{=}y;\ \mathsf{tl}{=}q|\!\}$.

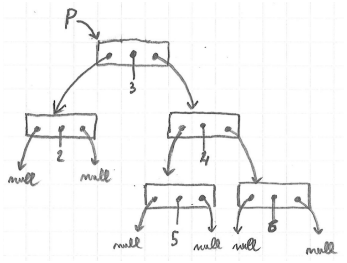
Summary

$$\begin{array}{rcl} p \leadsto \mathsf{MlistSeg}\, q\, L & \equiv & \mathsf{match}\, L\, \mathsf{with} \\ & & \mid \mathsf{nil} \;\Rightarrow\; [p = q] \\ & & \mid x :: L' \;\Rightarrow\; \exists p'.\; p \leadsto \{\!|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\!\} \\ & & \qquad\qquad\qquad \star\, p' \leadsto \mathsf{MlistSeg}\, q\, L' \end{array}$$

Chapter 4

Representation Predicate for Trees

Implementation of a mutable binary trees



Empty trees represented as null pointers. Nodes represented as records.

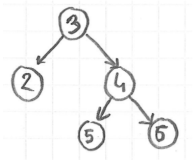
```
type node = {
  mutable item : int;
  mutable left : node;
  mutable right : node; }
```

Logical binary trees

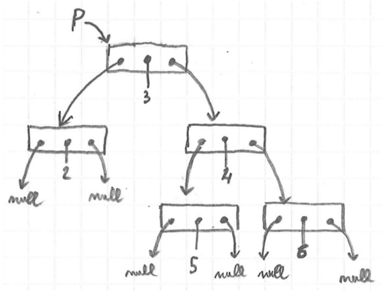
```
Inductive tree : Type :=
  | Leaf : tree
  | Node : int → tree → tree → tree.
```

Example:

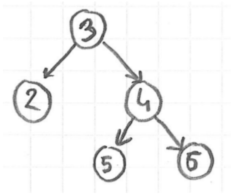
```
Node 3
     (Node 2 Leaf Leaf)
     (Node 4 (Node 5 Leaf Leaf)
             (Node 6 Leaf Leaf))
```



Representation predicate for binary trees



$T =$



Representation predicate:

$$p \rightsquigarrow \mathsf{Mtree}\, T$$

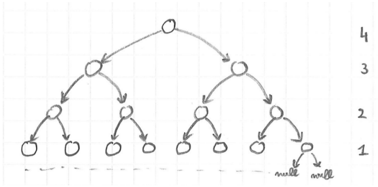
Representation predicate for binary trees

$$
\begin{array}{rcl}
p \leadsto \mathsf{Mlist}\, L & \equiv & \mathsf{match}\ L\ \mathsf{with} \\
 & & \mid \mathsf{nil} \ \Rightarrow\ [p = \mathsf{null}] \\
 & & \mid x :: L' \ \Rightarrow\ \exists p'.\quad p \leadsto \{\!|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\!\} \\
 & & \qquad\qquad\qquad\quad \star\ p' \leadsto \mathsf{Mlist}\, L'
\end{array}
$$

Exercise: define $p \leadsto \mathsf{Mtree}\, T$.

$$
\begin{array}{rcl}
p \leadsto \mathsf{Mtree}\, T & \equiv & \mathsf{match}\ T\ \mathsf{with} \\
 & & \mid \mathsf{Leaf} \ \Rightarrow\ [p = \mathsf{null}] \\
 & & \mid \mathsf{Node}\, x\, T_1\, T_2 \ \Rightarrow\ \exists p_1 p_2. \\
 & & \qquad\qquad p \mapsto \{\!|\mathsf{item}{=}x;\ \mathsf{left}{=}p_1;\ \mathsf{right}{=}p_2|\!\} \\
 & & \qquad\quad \star\ p_1 \leadsto \mathsf{Mtree}\, T_1 \\
 & & \qquad\quad \star\ p_2 \leadsto \mathsf{Mtree}\, T_2
\end{array}
$$

Complete binary tree



$$p \rightsquigarrow \mathsf{MtreeDepth}\, n\, T$$

describes a complete binary tree whose leaves are all at depth n.

Complete binary tree (1/2)

$$
\begin{array}{rcl}
p \rightsquigarrow \mathsf{Mtree}\, T & \equiv & \mathsf{match}\, T\, \mathsf{with} \\
& & \mid \mathsf{Leaf} \;\Rightarrow\; [p = \mathsf{null}] \\
& & \mid \mathsf{Node}\, x\, T_1\, T_2 \;\Rightarrow\; \exists p_1 p_2. \\
& & \qquad\qquad p \mapsto \{\!|\mathsf{item}{=}x;\ \mathsf{left}{=}p_1;\ \mathsf{right}{=}p_2|\!\} \\
& & \qquad\quad \star\ p_1 \rightsquigarrow \mathsf{Mtree}\, T_1 \\
& & \qquad\quad \star\ p_2 \rightsquigarrow \mathsf{Mtree}\, T_2
\end{array}
$$

Exercise: define $p \rightsquigarrow \mathsf{MtreeDepth}\, n\, T$ by generalizing $p \rightsquigarrow \mathsf{Mtree}\, T$.

Complete binary tree (1/2), solution

$$
\begin{array}{rcl}
p \rightsquigarrow \mathsf{MtreeDepth}\, n\, T & \equiv & \mathsf{match}\, T\, \mathsf{with} \\
& & \mid \mathsf{Leaf} \;\Rightarrow\; [p = \mathsf{null} \;\wedge\; n = 0] \\
& & \mid \mathsf{Node}\, x\, T_1\, T_2 \;\Rightarrow\; \exists p_1 p_2. \\
& & \qquad p \mapsto \{\!|\mathsf{item}{=}x;\ \mathsf{left}{=}p_1;\ \mathsf{right}{=}p_2|\!\} \\
& & \qquad \star\, p_1 \rightsquigarrow \mathsf{MtreeDepth}\,(n-1)\,T_1 \\
& & \qquad \star\, p_2 \rightsquigarrow \mathsf{MtreeDepth}\,(n-1)\,T_2
\end{array}
$$

Or:

$$
\begin{array}{rcl}
p \rightsquigarrow \mathsf{MtreeDepth}\, n\, T & \equiv & \mathsf{match}\, n, T\, \mathsf{with} \\
& & \mid O,\ \mathsf{Leaf} \;\Rightarrow\; [p = \mathsf{null}] \\
& & \mid S\, m,\ \mathsf{Node}\, x\, T_1\, T_2 \;\Rightarrow\; \exists p_1 p_2. \\
& & \qquad p \mapsto \{\!|\mathsf{item}{=}x;\ \mathsf{left}{=}p_1;\ \mathsf{right}{=}p_2|\!\} \\
& & \qquad \star\, p_1 \rightsquigarrow \mathsf{MtreeDepth}\, m\, T_1 \\
& & \qquad \star\, p_2 \rightsquigarrow \mathsf{MtreeDepth}\, m\, T_2 \\
& & \mid \_,\ \_ \;\Rightarrow\; [\mathsf{False}]
\end{array}
$$

Complete binary tree (2/2)

Exercise: give an alternative definition of "$p \rightsquigarrow \mathsf{MtreeDepth}\, n\, T$", this time by reusing the definition of $p \rightsquigarrow \mathsf{Mtree}\, T$ without modification.

$$p \rightsquigarrow \mathsf{MtreeDepth}\, n\, T \;\equiv\; p \rightsquigarrow \mathsf{Mtree}\, T \star [\mathsf{depth}\, n\, T]$$

```
Inductive depth : int → tree → Prop :=
  | depth_leaf :
      depth 0 Leaf
  | depth_node : ∀n x T1 T2,
      depth n T1 →
      depth n T2 →
      depth (n+1) (Node x T1 T2).
```

Complete binary tree of unspecified depth

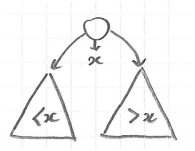
$$p \rightsquigarrow \mathsf{MtreeDepth}\, n\, T \;\equiv\; (p \rightsquigarrow \mathsf{Mtree}\, T) \star [\mathsf{depth}\, n\, T]$$

Exercise: define a predicate $p \rightsquigarrow \mathsf{MtreeComplete}\, T$ for describing a mutable complete binary tree, of some unspecified depth.

Equivalent definitions for $p \rightsquigarrow \mathsf{MtreeComplete}\, T$:

1. $\exists n.\ p \rightsquigarrow \mathsf{MtreeDepth}\, n\, T$
2. $\exists n.\ (p \rightsquigarrow \mathsf{Mtree}\, T) \star [\mathsf{depth}\, n\, T]$
3. $(p \rightsquigarrow \mathsf{Mtree}\, T) \star [\exists n. \mathsf{depth}\, n\, T]$

Binary search tree property



The proposition search $T\ E$ asserts that the pure tree T describes a valid search tree and that E describes the set integers that it contains.

```
Inductive search : tree → set int → Prop :=
  | search_leaf :
      search Leaf ∅
  | search_node : ∀x T1 T2,
      search T1 E1 →
      search T2 E2 →
      foreach (is_lt x) E1 →
      foreach (is_gt x) E2 →
      search (Node x T1 T2) ({x} ∪ E1 ∪ E2).
```

Binary search tree predicate

Exercise: define a predicate $p \rightsquigarrow \mathsf{MsearchTree}\, E$ for describing a mutable binary search tree storing the set of elements E.

$$p \rightsquigarrow \mathsf{MsearchTree}\, E \;\; \equiv \;\; \exists T.\; p \rightsquigarrow \mathsf{Mtree}\, T \,\star\, [\mathsf{search}\, T\, E]$$

For example, a call "`add x p`" can be specified as follows:

- pre-condition: $p \rightsquigarrow \mathsf{MsearchTree}\, E$
- post-condition: $p \rightsquigarrow \mathsf{MsearchTree}\, (E \cup \{x\})$

Summary

Common representation predicate for all binary trees:

$$\begin{array}{rcl} p \rightsquigarrow \mathsf{Mtree}\, T & \equiv & \mathsf{match}\, T\, \mathsf{with} \\ & & \mid \mathsf{Leaf} \;\Rightarrow\; [p = \mathsf{null}] \\ & & \mid \mathsf{Node}\, x\, T_1\, T_2 \;\Rightarrow\; \exists p_1 p_2. \\ & & \qquad\qquad p \mapsto \{\!|\mathsf{item}{=}x;\ \mathsf{left}{=}p_1;\ \mathsf{right}{=}p_2|\!\} \\ & & \qquad\quad \star\, p_1 \rightsquigarrow \mathsf{Mtree}\, T_1 \,\star\, p_2 \rightsquigarrow \mathsf{Mtree}\, T_2 \end{array}$$

Invariants are expressed on the pure trees:

$$p \rightsquigarrow \mathsf{MsearchTree}\, E \;\equiv\; \exists T.\ p \rightsquigarrow \mathsf{Mtree}\, T \,\star\, [\mathsf{search}\, T\, E]$$

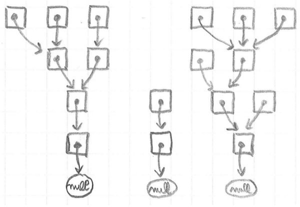
Operations are specified in terms of the model:

$$\{p \rightsquigarrow \mathsf{MsearchTree}\, E\}\ (\texttt{add x p})\ \{\lambda\_.\ p \rightsquigarrow \mathsf{MsearchTree}\,(E \cup \{x\})\}$$

Chapter 5

Structures with sharing

The union-find data structure



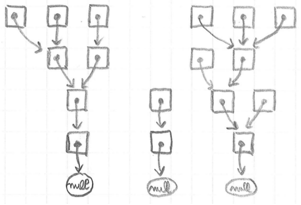
```
type node = node ref
```

Implements an equivalence relation S of type: loc → loc → Prop.

$$S\,a\,b \;\Leftrightarrow\; a \text{ and } b \text{ are two valid nodes with the same root}$$

Remark: $S\,a\,a$ holds iff a is the location of an existing node.

Representation of union-find cells

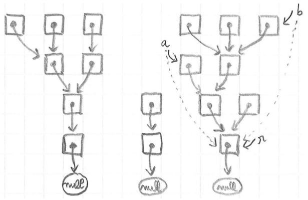


$$(p_1 \mapsto q_1) \star (p_2 \mapsto q_2) \star \ldots \star (p_n \mapsto q_n)$$

$$= \quad \circledast_{(p_i,q_i) \in G} \, (p_i \mapsto q_i)$$

where G is a finite map from locations to locations.

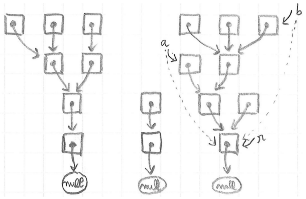
Invariants of union-find



Predicate "root $G\,a\,r$" asserts that in the graph G, node a has root r.

```
Inductive root : fmap loc loc → loc → loc → Prop :=
  | root_init : ∀G x,
      binds G x null →
      root G x x
  | root_step : ∀G x y r,
      binds G x y →
      y ≠null →
      root G y r →
      root G x r.
```

Specification of the union-find structure



$$\mathsf{UnionFind}\,S \quad\equiv\quad \exists G.\quad \left(\circledast_{(p,q)\in G}\; p \mapsto q\right)$$
$$\star\,[\forall a \in \mathsf{dom}\,G.\ \exists r.\ \mathsf{root}\,G\,a\,r]$$
$$\star\,[\forall ab.\ S\,a\,b \Leftrightarrow \exists r.\ \mathsf{root}\,G\,a\,r \wedge \mathsf{root}\,G\,b\,r]$$

For example, "`let x = is_equiv a b`" is specified as follows:

- pre-condition: $[S\,a\,a \;\wedge\; S\,b\,b] \star \mathsf{UnionFind}\,S$
- post-condition: $[x = \mathsf{true} \Leftrightarrow S\,a\,b] \star \mathsf{UnionFind}\,S$

Summary

Iterated separating conjunction, written $\circledast$.

For Union-Find:

$$\underset{(p,q)\in G}{\circledast} \; p \mapsto q$$

Chapter 6

Separation Logic Triples

Separation Logic triples

A term t is specified 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 value produced by t
- H' describes the final heap and the output value x.

$$\{H\}\ t\ \{Q\}$$

- H (pre-condition) is a predicate of type: Heap $\to$ Prop
- t has an ML type interpreted in the logic as type A
- Q (post-condition) is a predicate of type: $A \to$ Heap $\to$ Prop.

Examples of triples

Example 1:

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

Example 2:

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

Example 3:

$$\{r \mapsto 3\}\;(\texttt{!r})\;\{\lambda x.\; [x = 3] \star (r \mapsto 3)\}$$

Example 4:

$$\{r \mapsto 3\}\;(\texttt{incr r})\;\{\lambda\_.\,(r \mapsto 4)\}$$

Remark: "$\lambda\_.\,(r \mapsto 4)$" is the same as "fun (_ : unit) $\rightarrow$ $(r \mapsto 4)$" in Coq.

Specification of functions

A function f is specified using a triple of the form:

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

- H is the pre-condition
- f is the function
- a is the value of the argument
- x is a name for the return value
- H' is the post-condition

Example:

$$\forall r n. \quad \{r \mapsto n\}\ (\texttt{incr r})\ \{\lambda\_.\ r \mapsto (n+1)\}$$

Specification of operations on memory cells

Exercise: specify the primitive operations on references.

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

Solution:

$$\forall v. \quad \{[\,]\}\ (\texttt{ref v})\ \{\lambda r.\ (r \mapsto v)\}$$
$$\forall r v. \quad \{r \mapsto v\}\ (\texttt{!r})\ \{\lambda x.\ [x = v] \star (r \mapsto v)\}$$

$$\begin{array}{lr}
\forall r v v'. & \{r \mapsto v'\}\ (\texttt{r := v})\ \{\lambda\_.\,(r \mapsto v)\} \\
\forall r v. & \{\exists v'.\ r \mapsto v'\}\ (\texttt{r := v})\ \{\lambda\_.\,(r \mapsto v)\} \\
\forall r v. & \{r \mapsto -\}\ (\texttt{r := v})\ \{\lambda\_.\,(r \mapsto v)\}
\end{array}$$

where $(r \mapsto -) \equiv \exists a.\ r \mapsto a$.

Specification of partial functions

Presentation 1:

$$\forall n. \quad \{[n \geqslant 0]\}\ (\mathsf{facto}\, n)\ \{\lambda x.\, [x = n!]\}$$

Presentation 2:

$$\forall n.\ n \geqslant 0 \ \Rightarrow\ \{[\,]\}\ (\mathsf{facto}\, n)\ \{\lambda x.\, [x = n!]\}$$

Specification of operations on arrays

Exercise: specify operations on arrays in terms of $p \rightsquigarrow \text{Array}\, L$.

```
(Array.get i p)
(Array.set i p v)
(Array.length p)
(Array.create n v)
```

Notation:

$L[i]$	$\equiv$	i-th element of the list L
$L[i := v]$	$\equiv$	copy of L with v at index i
$\lvert L\rvert$	$\equiv$	length of L
$i \in \text{dom}\, L$	$\equiv$	$0 \leqslant i < \lvert L\rvert$

Specification of operations on arrays

$$i \in \mathsf{dom}\, L \;\Rightarrow\; \{p \rightsquigarrow \mathsf{Array}\, L\}\ (\texttt{Array.get i p})\ \{\lambda x.\ [x = L[i]] \star p \rightsquigarrow \mathsf{Array}\, L\}$$

$$i \in \mathsf{dom}\, L \;\Rightarrow\; \{p \rightsquigarrow \mathsf{Array}\, L\}\ (\texttt{Array.set i p v})\ \{\lambda\_.\ p \rightsquigarrow \mathsf{Array}\,(L[i := v])\}$$

$$\{p \rightsquigarrow \mathsf{Array}\, L\}\ (\texttt{Array.length p})\ \{\lambda n.\ [n = |L|] \star p \rightsquigarrow \mathsf{Array}\, L\}$$

$$n \geqslant 0 \;\Rightarrow\; \{[\,]\}\ (\texttt{Array.create n v})\ \{\lambda p.\ \exists L.\ (p \rightsquigarrow \mathsf{Array}\, L) \star [|L| = n] \star [\forall i \in \mathsf{dom}\, L.\ L[i] = v]\}$$

Interface for mutable queues

Interface:

```
create : unit -> 'a queue
is_empty : 'a queue -> bool
push : 'a -> 'a queue -> unit
pop : 'a queue -> 'a
transfer : 'a queue -> 'a queue -> unit
```

Exercise: specify the interface for mutable queues in terms of:

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

Remark: items are pushed to the back of list, and popped from the head; transfer migrates items from the second queue to the back of the first.

Specification of abstract mutable queues

$\{[\,]\}\ (\texttt{create()})\ \{\lambda q.\ q \leadsto \mathsf{Queue}\,\mathsf{nil}\}$

$\{q \leadsto \mathsf{Queue}\,L\}\ (\texttt{is\_empty q})\ \{\lambda b.\ [b = \mathsf{true} \Leftrightarrow L = \mathsf{nil}] \star q \leadsto \mathsf{Queue}\,L\}$

$\{q \leadsto \mathsf{Queue}\,L\}\ (\texttt{push x q})\ \{\lambda\_.\ q \leadsto \mathsf{Queue}\,(L \mathbin{+\!\!+} x :: \mathsf{nil})\}$

$L \neq \mathsf{nil} \ \Rightarrow\ \{q \leadsto \mathsf{Queue}\,L\}\ (\texttt{pop q})\ \{\lambda x.\ \exists L'.\,[L = x :: L'] \star q \leadsto \mathsf{Queue}\,L'\}$

$\{q \leadsto \mathsf{Queue}\,(x :: L)\}\ (\texttt{pop q})\ \{\lambda r.\ [r = x] \star q \leadsto \mathsf{Queue}\,L\}$

$$\begin{array}{l}
\{q_1 \leadsto \mathsf{Queue}\,L_1 \ \star\ q_2 \leadsto \mathsf{Queue}\,L_2\} \\
(\texttt{transfer q1 q2}) \\
\{\lambda\_.\ q_1 \leadsto \mathsf{Queue}\,(L_1 \mathbin{+\!\!+} L_2) \ \star\ q_2 \leadsto \mathsf{Queue}\,\mathsf{nil}\}
\end{array}$$

Interpretation of triples (1/3)

Assume for now that triples describe the entire state.

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

$$\forall m. \quad H\, m \quad \Rightarrow \quad \exists v.\, \exists m'. \quad t_{/m} \Downarrow v_{/m'} \ \wedge\ ([x \to v]\, H')\, m'$$

How is a triple $\{H\}\ t\ \{Q\}$ intepreted?

Let $Q = \lambda x.\, H'$. We have $Q\, v = [x \to v]\, H'$. Thus, the interpretation is:

$$\forall m. \quad H\, m \quad \Rightarrow \quad \exists v.\, \exists m'. \quad t_{/m} \Downarrow v_{/m'} \ \wedge\ Q\, v\, m'$$

Interpretation of triples (2/3)

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.

Recall that:

$$m_1 \perp m_2 \;\equiv\; (\mathsf{dom}\, m_1 \cap \mathsf{dom}\, m_2 = \varnothing)$$

How is a triple $\{H\}\; t\; \{Q\}$ intepreted?

$$\forall m_1\, m_2.\; \begin{cases} H\, m_1 \\ m_1 \perp m_2 \end{cases} \Rightarrow\; \exists v.\, \exists m_1'.\; \begin{cases} t_{/m_1 \uplus m_2} \Downarrow v_{/m_1' \uplus m_2} \\ Q\, v\, m_1' \\ m_1' \perp m_2 \end{cases}$$

Function with garbage collection

What is the *natural* specification of function `next`?

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

What is missing from our current interpretation of triple?

Solution:

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

Correct but useless:

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

The post-condition should describe only a subset of the modified heap.

Interpretation of triples (3/3)

Let m_3 describe the *garbage* heap, that is, the part of the final heap that corresponds either to cells from m_1 or to cells allocated during the evaluation of t, and that are not described by the post-condition.

We interpret a triple $\{H\}\ t\ \{Q\}$ as:

$$\forall m_1\, m_2.\ \begin{cases} H\, m_1 \\ m_1 \perp m_2 \end{cases} \Rightarrow\ \exists v m'_1 m_3.\ \begin{cases} t_{/m_1 \uplus m_2} \Downarrow v_{/m'_1 \uplus m_2 \uplus m_3} \\ Q\, v\, m'_1 \\ m'_1 \perp m_2 \perp m_3 \end{cases}$$

Interpretation of triples (3/3), revisited

We introduce a new heap predicate, written GC, that holds of any heap.

$$\mathsf{GC} \;\equiv\; \exists H.\, H$$

Definition (Separation Logic Triple)

We define $\{H\}\; t\; \{Q\}$ as:

$$\forall m_1\, m_2.\ \left\{\begin{array}{l} H\, m_1 \\ m_1 \perp m_2 \end{array}\right. \quad\Rightarrow\quad \exists v m_1'.\ \left\{\begin{array}{l} t_{/m_1 \uplus m_2} \Downarrow v_{/m_1' \uplus m_2} \\ (Q\, v \star \mathsf{GC})\, m_1' \\ m_1' \perp m_2 \end{array}\right.$$

Summary

Separation Logic triple:

$$\{H\}\ t\ \{\lambda x.\, H'\}$$

Specification of a function:

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

Specification of primitive functions:

$$\begin{array}{ll}
\forall v. & \{[\,]\}\ (\texttt{ref v})\ \{\lambda r.\ (r \mapsto v)\} \\
\forall rv. & \{r \mapsto v\}\ (\texttt{!r})\ \{\lambda x.\ [x = v] \star (r \mapsto v)\} \\
\forall rv. & \{r \mapsto -\}\ (\texttt{r := v})\ \{\lambda \_.\, (r \mapsto v)\}
\end{array}$$

Interpretation of triples: see definition.

Exercises

- Exam from 2014, Exercise 2: Circular lists.

Available from the webpage of the course.

The end!

Separation Logic 2/4

Arthur Charguéraud

Febuary 8th, 2017

Chapter 7

The Frame Rule

Preservation of independent state

We have:

$$\{r \mapsto 2\}\ (\texttt{incr r})\ \{\lambda\_.\ r \mapsto 3\}$$

We also have:

$$\{r \mapsto 2 \star s \mapsto 7\}\ (\texttt{incr r})\ \{\lambda\_.\ r \mapsto 3 \star s \mapsto 7\}$$

More generally:

$$\{r \mapsto 2 \star H\}\ (\texttt{incr r})\ \{\lambda\_.\ r \mapsto 3 \star H\}$$

The frame rule

Principle: a triple remains valid when both the pre-condition and the post-condition are extended with a same heap predicate.

General form:

$$\frac{\{H_1\}\ t\ \{\lambda x.\, H_1'\}}{\{H_1 \star H_2\}\ t\ \{\lambda x.\, H_1' \star H_2\}}$$

Frame rule and allocation

We have:

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

By the frame rule, we have:

$$\{s \mapsto 5\}\ (\texttt{ref 3})\ \{\lambda r.\ (r \mapsto 3) \star (s \mapsto 5)\}$$

This post-condition ensures $r \neq s$.

The reference cell r is thus guaranteed to be distinct from any cell that might exist prior to the allocation of r.

Length of a mutable list, recursively

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

Specification:

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

We prove this specification by induction on L.

Verification of mlength: nil case

Case $p =$ **null**. Goal is:

$$\{p \rightsquigarrow \mathsf{Mlist}\, L\}\ (\mathtt{0})\ \{\lambda n.\ [n = \mathsf{length}\, L] \star p \rightsquigarrow \mathsf{Mlist}\, L\}$$

Exploit:

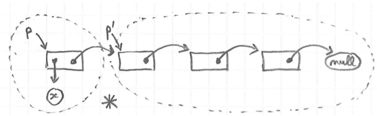
$$\mathsf{null} \rightsquigarrow \mathsf{Mlist}\, L \quad \rhd \quad [L = \mathsf{nil}] \ \star\ \mathsf{null} \rightsquigarrow \mathsf{Mlist}\, L$$

and:

$$0 = \mathsf{length}\,\mathsf{nil}$$

Verification of mlength: frame process

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



Assume $L = x :: L'$.

$p \rightsquigarrow \mathsf{Mlist}\, L$	pre-condition
$p \rightsquigarrow \{\|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'\|\} \star p' \rightsquigarrow \mathsf{Mlist}\, L'$	by unfolding
$p' \rightsquigarrow \mathsf{Mlist}\, L'$	frame begins
$p' \rightsquigarrow \mathsf{Mlist}\, L' \star [n' = \|L'\|]$	by induction
$p \rightsquigarrow \{\|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'\|\} \star p' \rightsquigarrow \mathsf{Mlist}\, L' \star [n' = \|L'\|]$	frame ends
$p \rightsquigarrow \mathsf{Mlist}\, L \star [n' + 1 = \|x :: L'\|]$	by folding
$p \rightsquigarrow \mathsf{Mlist}\, L \star [n = \|L\|]$	post-condition

Instantiation of the frame rule

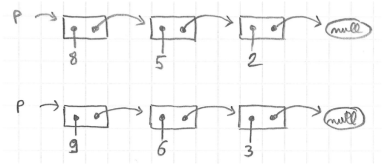
Induction hypothesis:

$$\{p' \rightsquigarrow \mathsf{Mlist}\, L'\}$$
$$(\texttt{mlength p'})$$
$$\{\lambda n'.\ [n = \mathsf{length}\, L'] \star p' \rightsquigarrow \mathsf{Mlist}\, L'\}$$

By the frame rule:

$$\{p' \rightsquigarrow \mathsf{Mlist}\, L' \star p \rightsquigarrow \{\!|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\!\}\,\}$$
$$(\texttt{mlength p'})$$
$$\{\lambda n.\ [n = \mathsf{length}\, L'] \star p' \rightsquigarrow \mathsf{Mlist}\, L' \star p \rightsquigarrow \{\!|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\!\}\,\}$$

Verification 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
```

$$\forall pL. \quad \{p \rightsquigarrow \mathsf{Mlist}\, L\}\ (\texttt{list\_incr p})\ \{\lambda\_.\ p \rightsquigarrow \mathsf{Mlist}\,(\mathsf{map}\,(+1)\,L)\}$$

Exercise: describe the frame process for in-place increment.

Verification of in-place increment: frame process

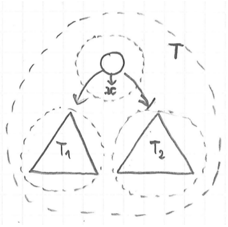
$$\forall pL. \quad \{p \rightsquigarrow \mathsf{Mlist}\, L\}\ (\texttt{list\_incr p})\ \{\lambda\_.\ p \rightsquigarrow \mathsf{Mlist}\,(\mathsf{map}\,(+1)\,L)\}$$

Assume $L = x :: L'$.

$p \rightsquigarrow \mathsf{Mlist}\, L$		pre-condition
$p \rightsquigarrow \{\mid \mathsf{hd}{=}x;\ \mathsf{tl}{=}p' \mid\}$	$\star\ p' \rightsquigarrow \mathsf{Mlist}\, L'$	by unfolding
$p \rightsquigarrow \{\mid \mathsf{hd}{=}x+1;\ \mathsf{tl}{=}p' \mid\}$	$\star\ p' \rightsquigarrow \mathsf{Mlist}\, L'$	incrementing
	$p' \rightsquigarrow \mathsf{Mlist}\, L'$	frame begins
	$p' \rightsquigarrow \mathsf{Mlist}\,(\mathsf{map}\,(+1)\,L')$	by induction
$p \rightsquigarrow \{\mid \mathsf{hd}{=}x+1;\ \mathsf{tl}{=}p' \mid\}$	$\star\ p' \rightsquigarrow \mathsf{Mlist}\,(\mathsf{map}\,(+1)\,L')$	frame ends
$p \rightsquigarrow \mathsf{Mlist}\,((x+1) :: (\mathsf{map}\,(+1)\,L'))$		by folding
$p \rightsquigarrow \mathsf{Mlist}\,(\mathsf{map}\,(+1)\,L)$		by rewriting
		post-condition

Specification of tree copy

```
let rec copy (p:node) : node =
  if p == null then null else
  let p1' = copy p.left in
  let p2' = copy p.right in
  { item = p.item;
    left = p1';
    right = p2' }
```



Exercise: specify the tree copy function.

$$\forall pT.\quad \{p \rightsquigarrow \mathsf{Mtree}\,T\}\ (\texttt{copy p})\ \{\lambda p'.\ p \rightsquigarrow \mathsf{Mtree}\,T\ \star\ p' \rightsquigarrow \mathsf{Mtree}\,T\}$$

Exercise: describe the frame process for tree copy.

Verification of tree copy: frame process

$p \rightsquigarrow \mathsf{Mtree}\, T$	by pre-condition
$p \mapsto \{\!\|x; p_1; p_2\|\!\} \star p_1 \rightsquigarrow \mathsf{Mtree}\, T_1 \star p_2 \rightsquigarrow \mathsf{Mtree}\, T_2$	by unfolding
$p_1 \rightsquigarrow \mathsf{Mtree}\, T_1$	frame begins
$p_1 \rightsquigarrow \mathsf{Mtree}\, T_1$ $\star\, p_1' \rightsquigarrow \mathsf{Mtree}\, T_1$	by induction
$p \mapsto \{\!\|x; p_1; p_2\|\!\} \star p_1 \rightsquigarrow \mathsf{Mtree}\, T_1 \star p_2 \rightsquigarrow \mathsf{Mtree}\, T_2$ $\star\, p_1' \rightsquigarrow \mathsf{Mtree}\, T_1$	frame ends
$p_2 \rightsquigarrow \mathsf{Mtree}\, T_2$	frame begins
$p_2 \rightsquigarrow \mathsf{Mtree}\, T_2$ $\star\, p_2' \rightsquigarrow \mathsf{Mtree}\, T_2$	by induction
$p \mapsto \{\!\|x; p_1; p_2\|\!\} \star p_1 \rightsquigarrow \mathsf{Mtree}\, T_1 \star p_2 \rightsquigarrow \mathsf{Mtree}\, T_2$ $\star\, p_1' \rightsquigarrow \mathsf{Mtree}\, T_1 \star p_2' \rightsquigarrow \mathsf{Mtree}\, T_2$	frame ends
$p \mapsto \{\!\|x; p_1; p_2\|\!\} \star p_1 \rightsquigarrow \mathsf{Mtree}\, T_1 \star p_2 \rightsquigarrow \mathsf{Mtree}\, T_2$ $\star\, p' \mapsto \{\!\|x; p_1'; p_2'\|\!\} \star p_1' \rightsquigarrow \mathsf{Mtree}\, T_1 \star p_2' \rightsquigarrow \mathsf{Mtree}\, T_2$	by allocation
$p \rightsquigarrow \mathsf{Mtree}\, T$ $\star\, p' \rightsquigarrow \mathsf{Mtree}\, T$	by folding

Summary

$$\frac{\{H_1\}\ t\ \{\lambda x.\, H_1'\}}{\{H_1 \star H_2\}\ t\ \{\lambda x.\, H_1' \star H_2\}}\ \text{FRAME}$$

In-place mutable list increment, when $L = x :: L'$.

$p \rightsquigarrow \mathsf{Mlist}\, L$		
$p \rightsquigarrow \{\!\mid \mathsf{hd}{=}x;\ \mathsf{tl}{=}p' \mid\!\}$	$\star\ p' \rightsquigarrow \mathsf{Mlist}\, L'$	by unfolding
$p \rightsquigarrow \{\!\mid \mathsf{hd}{=}x+1;\ \mathsf{tl}{=}p' \mid\!\}$	$\star\ p' \rightsquigarrow \mathsf{Mlist}\, L'$	incrementing
	$p' \rightsquigarrow \mathsf{Mlist}\, L'$	frame begins
	$p' \rightsquigarrow \mathsf{Mlist}\, (\mathsf{map}\, (+1)\, L')$	by induction
$p \rightsquigarrow \{\!\mid \mathsf{hd}{=}x+1;\ \mathsf{tl}{=}p' \mid\!\}$	$\star\ p' \rightsquigarrow \mathsf{Mlist}\, (\mathsf{map}\, (+1)\, L')$	frame ends
$p \rightsquigarrow \mathsf{Mlist}\, ((x+1) :: (\mathsf{map}\, (+1)\, L'))$		by folding
$p \rightsquigarrow \mathsf{Mlist}\, (\mathsf{map}\, (+1)\, L)$		by rewriting

Chapter 8

Small footprint specifications

Small footprint access to records

$$p \rightsquigarrow \{\!|\,\mathsf{hd}{=}v;\ \mathsf{tl}{=}q\,|\!\} \quad \equiv \quad p.\mathsf{hd} \mapsto v \star p.\mathsf{tl} \mapsto q$$

Specification of a write on the head field:

$$\{p \rightsquigarrow \{\!|\,\mathsf{hd}{=}w;\ \mathsf{tl}{=}q\,|\!\}\,\}\ (p.\mathsf{hd}\ \texttt{<-}\ v)\ \{\lambda\_.\ p \rightsquigarrow \{\!|\,\mathsf{hd}{=}v;\ \mathsf{tl}{=}q\,|\!\}\,\}$$

Same, but with a smaller footprint:

$$\{p.\mathsf{hd} \mapsto w\}\,(p.\mathsf{hd}\ \texttt{<-}\ v)\ \{\lambda\_.\ p.\mathsf{hd} \mapsto v\}$$

or $\{p.\mathsf{hd} \mapsto -\}\ (p.\mathsf{hd}\ \texttt{<-}\ v)\ \{\lambda\_.\ p.\mathsf{hd} \mapsto v\}$

From small to large footprint using frame:

$$\{p.\mathsf{hd} \mapsto w \star p.\mathsf{tl} \mapsto q\}\ (p.\mathsf{hd}\ \texttt{<-}\ v)\ \{\lambda\_.\ p.\mathsf{hd} \mapsto v \star p.\mathsf{tl} \mapsto q\}$$

Representation predicate for arrays

Representation predicate for C arrays:

$$p \leadsto \mathsf{Array}\, L \quad \equiv \quad \underset{v \text{ at index } i \text{ in } L}{\circledast} \quad p[i] \mapsto v$$

where:

$$p[i] \mapsto v \quad \equiv \quad (p + i) \mapsto v$$

Representation predicate for ML arrays:

$$p \leadsto \mathsf{Array}\, L \quad \equiv \quad p.\mathsf{length} \mapsto |L| \star \underset{v \text{ at index } i \text{ in } L}{\circledast} \quad p[i] \mapsto v$$

where p.length $\mapsto n$ and $p[i] \mapsto v$ are abstract definitions for the user.

Small footprint specifications of array operations

$i \in \mathsf{dom}\, L \;\Rightarrow$

$$\{p \rightsquigarrow \mathsf{Array}\, L\}\ (\texttt{Array.get i p})\ \{\lambda x.\ [x = L[i]] \star p \rightsquigarrow \mathsf{Array}\, L\}$$
$$\{p \rightsquigarrow \mathsf{Array}\, L\}\ (\texttt{Array.set i p v})\ \{\lambda\_.\ p \rightsquigarrow \mathsf{Array}\,(L[i := v])\}$$
$$\{p \rightsquigarrow \mathsf{Array}\, L\}\ (\texttt{Array.length p})\ \{\lambda n.\ [n = |L|] \star p \rightsquigarrow \mathsf{Array}\, L\}$$

Exercise: give small footprint specifications for array operations.
How to derive the large footprint specifications from them?

$$\{p[i] \mapsto v\}\quad (\texttt{Array.get i p})\quad \{\lambda x.\ [x = v] \star p[i] \mapsto v\}$$
$$\{p[i] \mapsto -\}\quad (\texttt{Array.set i p})\quad \{\lambda\_.\ p[i] \mapsto v\}$$
$$\{p.\mathsf{length} \mapsto n\}\ (\texttt{Array.length p})\ \{\lambda x.\ [x = n] \star p.\mathsf{length} \mapsto n\}$$

$$\begin{aligned} p \rightsquigarrow \mathsf{Array}\, L \;=\; & \quad p[i] \mapsto L[i] \\ & \star\ p.\mathsf{length} \mapsto |L| \\ & \star \bigotimes_{\substack{v \text{ at index } j \text{ in } L \\ \text{with } j \neq i}} p[j] \mapsto v \end{aligned}$$

Access to a memory cell

In C, record and array accesses are treated uniformly:

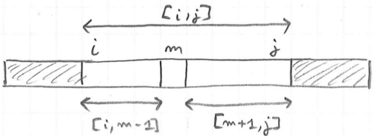
```
p.hd = v      compiles to      *(p+hd)=v
p[i] = v      compiles to      *(p+i)=v
```

Common small footprint specification for accessing a memory cell:

$$\{p \mapsto -\}\ (\texttt{*p = v})\ \{\lambda\_.\ p \mapsto v\}$$
$$\{p \mapsto v\}\ (\texttt{*p})\qquad \{\lambda x.\ [x = v] \star p \mapsto v\}$$

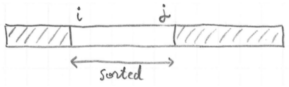
All other specifications for read and write operations are derivable.

Quicksort code



```
let rec qsort i j t =
  if j - i > 1 then begin
    let m = pivot i j t in
    qsort i (m-1) t;
    qsort (m+1) j t;
  end
```

Large-footprint specification of quicksort



$$
\begin{array}{l}
\forall p L i j.\quad 0 \leqslant i \leqslant j < |L| \;\Rightarrow \\
\quad \{p \rightsquigarrow \mathsf{Array}\, L\} \\
\quad (\texttt{qsort i j p}) \\
\quad \{\lambda\_.\ \exists L'.\quad (p \rightsquigarrow \mathsf{Array}\, L') \\
\qquad\qquad \star\, [\mathsf{sortedSeg}\, i\, j\, L'] \\
\qquad\qquad \star\, [\mathsf{permut}\, L\, L'] \\
\qquad\qquad \star\, [\forall a \in [0,i) \cup (j,|L|).\ L'[a] = L[a]] \quad \}
\end{array}
$$

where: $\mathsf{sortedSeg}\, i\, j\, L' \;\equiv\; \forall a, b \in [i,j].\ a \leqslant b \Rightarrow L'[a] \leqslant L'[b]$.

Group of array cells

Definition
$$p \leadsto \mathsf{Cells}\, M \quad \equiv \quad \underset{(i,v)\in M}{\circledast} \; p[i] \mapsto v$$

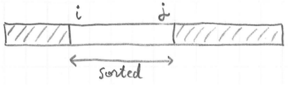
where M has type "map int A", for some A.

Conversions:

$$\begin{array}{rcl}
p \leadsto \mathsf{Array}\, L & = & p.\mathsf{length} \mapsto |L| \star p \leadsto \mathsf{Cells}\,(\mathsf{mapOfList}\, L) \\
p \leadsto \mathsf{Cells}\, \varnothing & = & [\,] \\
p \leadsto \mathsf{Cells}\, \{(i,v)\} & = & p[i] \mapsto v \\
p \leadsto \mathsf{Cells}\,(M_1 \uplus M_2) & = & p \leadsto \mathsf{Cells}\, M_1 \star p \leadsto \mathsf{Cells}\, M_2 \qquad (\text{when } M_1 \perp M_2)
\end{array}$$

where: $\mathsf{mapOfList}\, L \equiv \{(i,v) \mid v \text{ at index } i \text{ in } L\}$.

Small-footprint specification of quicksort

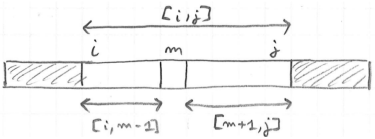


Exercise: give a small-footprint specification for quicksort.

- use $p \rightsquigarrow \mathsf{Cells}\, M$ with some constraints to describe a segment,
- use $\mathsf{sortedSeg}\, i\; j\; M$ to assert that a segment is sorted,
- use $\mathsf{permut}\, M\; M'$ to assert that values in a segment are permutted.

$$\begin{array}{ll} \forall p M i j. & \mathsf{dom}\, M = [i, j] \;\Rightarrow \\ & \{p \rightsquigarrow \mathsf{Cells}\, M\} \\ & (\texttt{qsort i j p}) \\ & \{\lambda\_.\; \exists M'.\quad p \rightsquigarrow \mathsf{Cells}\, M' \qquad\quad \} \\ & \qquad\qquad\quad \star\, [\mathsf{sortedSeg}\, i\; j\; M'] \\ & \qquad\qquad\quad \star\, [\mathsf{permut}\, M\; M'] \end{array}$$

Segment splitting in Quicksort



Assume dom $M = [i,j]$ with $j - i > 1$. Then:

$$\begin{aligned} p \rightsquigarrow \mathsf{Cells}\, M \quad = \quad & \ p \rightsquigarrow \mathsf{Cells}\,(M_{|\,[i,m-1]}) \\ & \star\ p \rightsquigarrow \mathsf{Cells}\,(M_{|\,[m,m]}) \\ & \star\ p \rightsquigarrow \mathsf{Cells}\,(M_{|\,[m+1,j]}) \end{aligned}$$

Note that:

$$p \rightsquigarrow \mathsf{Cells}\,(M_{|\,[m,m]}) \quad = \quad p[m] \mapsto M[m]$$

In recursive calls, we frame over the cells that are not involved.

Operations on groups of cells

Convenient specifications for operating directly on groups of cells:

$$i \in \mathsf{dom}\, M \;\Rightarrow$$

$$\{p \rightsquigarrow \mathsf{Cells}\, M\}\ (\texttt{Array.get i p})\ \{\lambda x.\ [x = M[i]] \star p \rightsquigarrow \mathsf{Cells}\, M\}$$

$$\{p \rightsquigarrow \mathsf{Cells}\, M\}\ (\texttt{Array.set i p v})\ \{\lambda\_.\ p \rightsquigarrow \mathsf{Cells}\, (M[i := v])\}$$

Summary

Representation of a full array using a list:

$$p \rightsquigarrow \mathsf{Array}\, L \quad \equiv \quad p.\mathsf{length} \mapsto |L| \star \mathop{\circledast}_{v \text{ at index } i \text{ in } L} p[i] \mapsto v$$

Small footprint accesses:

$$\begin{array}{lll}
\{p[i] \mapsto -\} & (\texttt{Array.get i p}) & \{\lambda\_.\ p[i] \mapsto v\} \\
\{p[i] \mapsto v\} & (\texttt{Array.set i p v}) & \{\lambda x.\ [x = v] \star p[i] \mapsto v\} \\
\{p.\mathsf{length} \mapsto n\} & (\texttt{Array.length p}) & \{\lambda x.\ [x = n] \star p.\mathsf{length} \mapsto n\}
\end{array}$$

Representation of a set of array cells using a finite map:

$$p \rightsquigarrow \mathsf{Cells}\, M \quad \equiv \quad \mathop{\circledast}_{(i,v) \in M} p[i] \mapsto v$$

Chapter 9

Heap entailment

Definition of Hprop

Let:

$$\mathsf{Hprop} \ \equiv \ \mathsf{Heap} \to \mathsf{Prop}$$

For example:

$$\begin{aligned} [\,] \quad &: \quad \mathsf{Hprop} \\ l \mapsto v \quad &: \quad \mathsf{Hprop} \\ H_1 \star H_2 \quad &: \quad \mathsf{Hprop} \end{aligned}$$

In particular:

$$(\star) \ : \ \mathsf{Hprop} \to \mathsf{Hprop} \to \mathsf{Hprop}$$

The separation algebra

Associativity: $(H_1 \star H_2) \star H_3 \;=\; H_1 \star (H_2 \star H_3)$

Commutativity: $H_1 \star H_2 \;=\; H_2 \star H_1$

Neutral element: $H \star [\,] \;=\; H$

Extrusion of existentials: $(\exists x.\, H_1) \star H_2 \;=\; \exists x.\, (H_1 \star H_2)$ (if $x \notin H_2$)

Extrusion of pure facts: $([P] \star H)\, m \;=\; P \;\wedge\; (H\, m)$

Extensionality

Functional extensionality:

$$(\forall x.\, f\,x = g\,x) \;\Rightarrow\; (f = g) \qquad \text{(if } f,g{:}A\to B\text{)}$$

Propositional extensionality:

$$(P \Leftrightarrow Q) \;\Rightarrow\; (P = Q) \qquad \text{(if } P,Q{:}\text{Prop)}$$

Predicate extensionality:

$$(\forall x.\, P\,x \Leftrightarrow Q\,x) \;\Rightarrow\; (P = Q) \qquad \text{(if } P,Q{:}A\to\text{Prop)}$$

Heap predicate extensionality:

$$(\forall m.\, H\,m \Leftrightarrow H'\,m) \;\Leftrightarrow\; (H = H') \qquad \text{(if } H,H'{:}\text{Hprop)}$$

Heap entailment

Definition:

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

For example:

$$(r \mapsto 6) \;\rhd\; \exists n.\, (r \mapsto n) \star [\text{even}\, n]$$

Thanks to $(\rhd)$, we never need to manipulate heaps explicitly.

Heap entailment as a partial order

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

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

$$\textsc{Reflexivity}\ \frac{}{H \rhd H} \qquad \textsc{Transitivity}\ \frac{H_1 \rhd H_2 \qquad H_2 \rhd H_3}{H_1 \rhd H_3} \qquad \textsc{Antisymmetry}\ \frac{H_1 \rhd H_2 \qquad H_2 \rhd H_1}{H_1 = H_2}$$

Frame property for heap entailment

$$\frac{H_1 \;\; \rhd H_1'}{H_1 \star H_2 \rhd H_1' \star H_2} \text{ ENTAIL-FRAME}$$

For example, to prove:

$$(r \mapsto 2) \star (s \mapsto 3) \;\; \rhd \;\; (r \mapsto 2) \star (t \mapsto n)$$

it suffices and prove:

$$(s \mapsto 3) \;\; \rhd \;\; (t \mapsto n).$$

Heap implications: true or false?

1.	$(r \mapsto 3) \star (s \mapsto 4) \rhd (s \mapsto 4) \star (r \mapsto 3)$	true
2.	$(r \mapsto 3) \rhd (s \mapsto 4) \star (r \mapsto 3)$	false
3.	$(s \mapsto 4) \star (r \mapsto 3) \rhd (r \mapsto 4)$	false
4.	$(s \mapsto 4) \star (r \mapsto 3) \rhd (r \mapsto 3)$	false
5.	$[\mathsf{False}] \star (r \mapsto 3) \rhd (s \mapsto 4) \star (r \mapsto 4)$	true
6.	$(r \mapsto 4) \star (s \mapsto 3) \rhd [\mathsf{False}]$	false
7.	$(r \mapsto 4) \star (r \mapsto 3) \rhd [\mathsf{False}]$	true
8.	$(r \mapsto 3) \star (r \mapsto 3) \rhd [\mathsf{False}]$	true

Instantiation of existentials and propositions

$$(r \mapsto 6) \;\rhd\; (\exists n.\, (r \mapsto n) \star [\mathsf{even}\, n])$$

To prove the above, we exhibit an even number n for which $r \mapsto n$.

Rules:

$$\frac{H_1 \rhd ([x \to v]\, H_2)}{H_1 \rhd (\exists x.\, H_2)}\ \textsc{exists-r} \qquad\qquad \frac{(H_1 \rhd H_2) \qquad P}{H_1 \rhd (H_2 \star [P])}\ \textsc{prop-r}$$

Example:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{(r \mapsto 6) \;\rhd\; (r \mapsto 6)}\ \textsc{refl} \qquad \dfrac{}{\mathsf{even}\, 6}\ \textsc{math}}{(r \mapsto 6) \;\rhd\; (r \mapsto 6) \star [\mathsf{even}\, 6]}\ \textsc{prop-r}}{(r \mapsto 6) \;\rhd\; [n \to 6]\, ((r \mapsto n) \star [\mathsf{even}\, n])}\ \textsc{subst}}{(r \mapsto 6) \;\rhd\; \exists n.\, (r \mapsto n) \star [\mathsf{even}\, n]}\ \textsc{exists-r}$$

Extraction of existentials and propositions

$$(\exists n.\,[\mathsf{even}\, n] \star (r \mapsto n)) \;\rhd\; (\exists m.\,[\mathsf{even}\, m] \star (r \mapsto m + 2))$$

To prove the above, we show that for any even number n, we have:

$$(r \mapsto n) \;\rhd\; \exists m.\,[\mathsf{even}\, m] \star (r \mapsto m + 2)$$

Reasoning rules:

$$\frac{\forall x.\;(H_1 \rhd H_2)}{(\exists x.\,H_1) \rhd H_2}\;\textsc{exists-l} \qquad\qquad \frac{P \;\Rightarrow\; (H_1 \rhd H_2)}{([P] \star H_1) \rhd H_2}\;\textsc{prop-l}$$

Same with explicit proof contexts:

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

Heap implications: true or false?

1.	$(r \mapsto 3) \rhd \exists n.\, (r \mapsto n)$	true
2.	$\exists n.\, (r \mapsto n) \rhd (r \mapsto 3)$	false
3.	$\exists n.\, (r \mapsto n) \star [n > 0] \rhd \exists n.\, [n > 1] \star (r \mapsto (n - 1))$	true
4.	$(r \mapsto 3) \star (s \mapsto 3) \rhd \exists n.\, (r \mapsto n) \star (s \mapsto n)$	true
5.	$\exists n.\, (r \mapsto n) \star [n > 0] \star [n < 0] \rhd (r \mapsto n) \star (r \mapsto n)$	true

Proving heap entailment relations

Systematic approach to dealing with heap entailment:

1. extract from left hand side,
2. instantiate in right hand side,
3. cancel equal predicates on both sides.

Example:

$$\begin{array}{c}
\hline
a : \mathsf{int},\ a > 5 \ \vdash\ (r \mapsto 3) \star (s \mapsto a) \ \rhd\ (r \mapsto 3) \star (s \mapsto a) \\
\hline
a : \mathsf{int},\ a > 5 \ \vdash\ (r \mapsto 3) \star (s \mapsto a) \ \rhd\ (r \mapsto 3) \star (s \mapsto 3 + (a - 3)) \\
\hline
a : \mathsf{int},\ a > 5 \ \vdash\ (r \mapsto 3) \star (s \mapsto a) \ \rhd\ \exists m.\, (r \mapsto 3) \star (s \mapsto 3 + m) \\
\hline
a : \mathsf{int},\ a > 5 \ \vdash\ (r \mapsto 3) \star (s \mapsto a) \ \rhd\ \exists nm.\, (r \mapsto n) \star (s \mapsto n + m) \\
\hline
\emptyset \ \vdash\ \exists a.\, [a > 5] \star (r \mapsto 3) \star (s \mapsto a) \ \rhd\ \exists nm.\, (r \mapsto n) \star (s \mapsto n + m) \\
\hline
\emptyset \ \vdash\ (r \mapsto 3) \star \exists a.\, [a > 5] \star (s \mapsto a) \ \rhd\ \exists nm.\, (s \mapsto n + m) \star (r \mapsto n)
\end{array}$$

Summary

$(\star)$ is associative, commutative, and has $[\,]$ as neutral element.

Existentials and pure facts may be extruded from stars.

$(\rhd)$ is a partial order, satisfying the frame property.

"$[\text{False}] \rhd H$" is always true.

"$(r \mapsto n) \star (r \mapsto m)$" is equivalent to "$[\text{False}]$".

Strategy: extract from the right, instantiate on the left, then cancel out.

Chapter 10

Structural rules

Frame rule

$$\frac{\{H_1\}\ t\ \{\lambda x.\, H_1'\}}{\{H_1 \star H_2\}\ t\ \{\lambda x.\, H_1' \star H_2\}}$$

Reformulation:

$$\frac{\{H_1\}\ t\ \{Q_1\}}{\{H_1 \star H_2\}\ t\ \{Q_1 \star H_2\}}\ \textsc{frame}$$

with the overloading:

$$Q \star H \ \equiv\ \lambda x.\ (Q\, x \star H)$$

Consequence rule

$$\frac{H \rhd H' \qquad \{H'\}\; t\; \{Q'\} \qquad Q' \rhd Q}{\{H\}\; t\; \{Q\}}\ \textsc{consequence}$$

with the overloading:

$$Q' \rhd Q \;\equiv\; \forall x.\, (Q'\, x \rhd Q\, x)$$

Note that H and H' must cover the same set of memory cells, that is, no garbage collection is allowed here. Similarly for Q and Q'.

Recall the need for garbage collection

```
let myref x =
  let r = ref x in
  let s = ref r in
  r
```

From:

$$\{[\,]\}\ (\texttt{myref x})\ \{\lambda r.\ r \mapsto x \star \exists s.\, s \mapsto r\}$$

To:

$$\{[\,]\}\ (\texttt{myref x})\ \{\lambda r.\ r \mapsto x\}$$

Can the following rule be used?

$$\frac{\{H\}\ t\ \{Q \star H'\}}{\{H\}\ t\ \{Q\}}\ \textsc{gc-post}'$$

Garbage collection rule

$$\frac{\{H\}\ t\ \{Q \star \mathsf{GC}\}}{\{H\}\ t\ \{Q\}}\ \textsc{gc-post} \qquad \text{where: } \mathsf{GC}\ \equiv\ \exists H'.\, H'$$

Same as:

$$\frac{\{H\}\ t\ \{\lambda x.\ (Q\,x \star \exists H'.\, H')\}}{\{H\}\ t\ \{Q\}}$$

Observe that H' may depend on the return value x:
For $(\lambda r.\ r \mapsto x\ \star\ \exists s.\, s \mapsto r)$, we may instantiate H' as $(\exists s.\, s \mapsto r)$.

Garbage collection in the pre-condition

$$\frac{\{H\}\ t\ \{Q\}}{\{H \star \mathsf{GC}\}\ t\ \{Q\}}\ \textsc{gc-pre} \qquad \frac{\{H\}\ t\ \{Q\}}{\{H \star H'\}\ t\ \{Q\}}\ \textsc{gc-pre'}$$

Exercise: show that GC-PRE is derivable from GC-POST and FRAME.

$$\frac{\{H\}\ t\ \{Q \star \mathsf{GC}\}}{\{H\}\ t\ \{Q\}}\ \textsc{gc-post} \qquad \frac{\{H_1\}\ t\ \{Q_1\}}{\{H_1 \star H_2\}\ t\ \{Q_1 \star H_2\}}\ \textsc{frame}$$

Proof:

$$\dfrac{\dfrac{\{H\}\ t\ \{Q\}}{\{H \star \mathsf{GC}\}\ t\ \{Q \star \mathsf{GC}\}}\ \textsc{frame}}{\{H \star \mathsf{GC}\}\ t\ \{Q\}}\ \textsc{gc-post}$$

Combined structural rule

$$\frac{H \rhd H' \qquad \{H'\}\ t\ \{Q'\} \qquad Q' \rhd Q}{\{H\}\ t\ \{Q\}}\ \textsc{consequence}$$

$$\frac{\{H\}\ t\ \{Q \star \mathsf{GC}\}}{\{H\}\ t\ \{Q\}}\ \textsc{gc-post} \qquad\qquad \frac{\{H_1\}\ t\ \{Q_1\}}{\{H_1 \star H_2\}\ t\ \{Q_1 \star H_2\}}\ \textsc{frame}$$

$$\frac{H = H_1 \star H_2 \qquad \{H_1\}\ t\ \{Q_1\} \qquad Q_1 \star H_2 = Q}{\{H\}\ t\ \{Q\}}\ \textsc{frame'}$$

$$\frac{H \rhd H_1 \star H_2 \qquad \{H_1\}\ t\ \{Q_1\} \qquad Q_1 \star H_2 \rhd Q \star \mathsf{GC}}{\{H\}\ t\ \{Q\}}\ \textsc{combined}$$

Extraction of existentials and propositions

$$\{\exists n.\,(r \mapsto n) \star [\mathsf{even}\,n]\}\ (\texttt{!r})\ \{\lambda x.\,...\}$$

To prove the above, we need to show that:

$$\forall n.\ \mathsf{even}\,n \ \ \Rightarrow\ \ \{r \mapsto n\}\ (\texttt{!r})\ \{\lambda x.\,...\}$$

Rules:

$$\frac{\forall x.\ \{H\}\ t\ \{Q\}}{\{\exists x.\,H\}\ t\ \{Q\}}\ \textsc{exists} \qquad\qquad \frac{P\ \Rightarrow\ \{H\}\ t\ \{Q\}}{\{[P] \star H\}\ t\ \{Q\}}\ \textsc{prop}$$

Application: copying a tree with invariants

Specification of copy for binary trees:

$$\{p \rightsquigarrow \mathsf{Mtree}\, T\}\ (\mathtt{copy\ \ p})\ \{\lambda p'.\ p \rightsquigarrow \mathsf{Mtree}\, T \star p' \rightsquigarrow \mathsf{Mtree}\, T\}$$

Description of complete binary trees:

$$p \rightsquigarrow \mathsf{MtreeComplete}\, T \;\; \equiv \;\; \exists n.\ (p \rightsquigarrow \mathsf{Mtree}\, T) \star [\mathsf{depth}\, n\, T]$$

Exercise: give a specification of copy in terms of MtreeComplete; which rules are used to derive this specification?

$$\{p \rightsquigarrow \mathsf{MtreeComplete}\, T\}\ (\mathtt{copy\ \ p})\ \{\lambda p'.\ \begin{array}{l} \quad p \rightsquigarrow \mathsf{MtreeComplete}\, T \\ \star\, p' \rightsquigarrow \mathsf{MtreeComplete}\, T \end{array}\}$$

Proof of the derived specification

(1) By unfolding of MtreeComplete:

$$\begin{array}{l}\{\exists n.\ (p \leadsto \mathsf{Mtree}\,T) \star [\mathsf{depth}\,n\,T]\} \\ (\texttt{copy p}) \\ \{\lambda p'.\quad \exists n.\ (p \leadsto \mathsf{Mtree}\,T) \star [\mathsf{depth}\,n\,T] \quad \} \\ \qquad\quad \star\, \exists n.\ (p' \leadsto \mathsf{Mtree}\,T) \star [\mathsf{depth}\,n\,T]\end{array}$$

(2) By the EXISTS and PROP rules:

$$\forall n.\ \mathsf{depth}\,n\,T \ \Rightarrow\ \begin{array}{l}\{p \leadsto \mathsf{Mtree}\,T\} \\ (\texttt{copy p}) \\ \{\lambda p'.\quad \exists n.\ (p \leadsto \mathsf{Mtree}\,T) \star [\mathsf{depth}\,n\,T] \quad \} \\ \qquad\quad \star\, \exists n.\ (p' \leadsto \mathsf{Mtree}\,T) \star [\mathsf{depth}\,n\,T]\end{array}$$

(3) By the CONSEQUENCE rule:

$$p \leadsto \mathsf{Mtree}\,T \star p' \leadsto \mathsf{Mtree}\,T \ \rhd \ \begin{array}{l}\ \ \exists n.\ (p \leadsto \mathsf{Mtree}\,T) \star [\mathsf{depth}\,n\,T] \\ \star\, \exists n.\ (p' \leadsto \mathsf{Mtree}\,T) \star [\mathsf{depth}\,n\,T]\end{array}$$

(4) Conclude using comm., assoc., extrusion, and EXISTS-R and PROP-R.

Summary

Structural rules:

$$\frac{H \rhd H_1 \star H_2 \qquad \{H_1\}\ t\ \{Q_1\} \qquad Q_1 \star H_2 \rhd Q \star \mathsf{GC}}{\{H\}\ t\ \{Q\}}\ \textsc{combined}$$

$$\frac{\forall x.\ \{H\}\ t\ \{Q\}}{\{\exists x.\, H\}\ t\ \{Q\}}\ \textsc{exists} \qquad\qquad \frac{P\ \Rightarrow\ \{H\}\ t\ \{Q\}}{\{[P] \star H\}\ t\ \{Q\}}\ \textsc{prop}$$

Other structural rules are derivable.

Chapter 11

Reasoning rules for terms

Reasoning rule for sequences

Example:

$$\frac{\{r \mapsto n\}\ (\texttt{incr r})\ \{\lambda\_.\, r \mapsto n+1\} \qquad \{r \mapsto n+1\}\ (\texttt{!r})\ \{\lambda x.\,[x = n+1] \star r \mapsto n+1\}}{\{r \mapsto n\}\ (\texttt{incr r; !r})\ \{\lambda x.\,[x = n+1] \star r \mapsto n+1\}}$$

Exercise: complete the rule for sequences.

$$\frac{\{...\}\ t_1\ \{...\} \qquad \{...\}\ t_2\ \{...\}}{\{H\}\ (t_1\ ;\ t_2)\ \{Q\}}$$

Reasoning rule for sequences

Solution 1:

$$\frac{\{H\}\ t_1\ \{\lambda\_.\, H'\} \qquad \{H'\}\ t_2\ \{Q\}}{\{H\}\ (t_1\ ;\ t_2)\ \{Q\}}$$

Solution 2:

$$\frac{\{H\}\ t_1\ \{Q'\} \qquad \{Q'\,()\}\ t_2\ \{Q\}}{\{H\}\ (t_1\ ;\ t_2)\ \{Q\}}\ \text{SEQ}$$

Remark: $Q' = \lambda\_.\, H'$ is equivalent to $Q'\,() = H'$.

Reasoning rule for let-bindings

Exercise: complete the reasoning rule for let-bindings.

$$\frac{\{...\}\ t_1\ \{...\} \qquad \forall x.\ \big(\{...\}\ t_2\ \{...\}\big)}{\{H\}\ (\mathsf{let}\, x = t_1\, \mathsf{in}\, t_2)\ \{Q\}}$$

Solution:

$$\frac{\{H\}\ t_1\ \{Q'\} \qquad \forall x.\ \{Q'\,x\}\ t_2\ \{Q\}}{\{H\}\ (\mathsf{let}\, x = t_1\, \mathsf{in}\, t_2)\ \{Q\}}\ \text{LET}$$

Example of let-binding

$$\frac{\{H\}\; t_1\; \{Q'\} \qquad \forall x.\; \{Q'\,x\}\; t_2\; \{Q\}}{\{H\}\; (\mathsf{let}\, x = t_1 \,\mathsf{in}\, t_2)\; \{Q\}}$$

Exercise: instantiate the rule for let-bindings on the following code.

$$\{r \mapsto 3\}\; (\texttt{let a = !r in a+1})\; \{Q\}$$

Solution:

$$\begin{array}{lcl} H & \equiv & (r \mapsto 3) \\ Q & \equiv & \lambda x.\; [x = 4] \star (r \mapsto 3) \\ Q' & \equiv & \lambda y.\; [y = 3] \star (r \mapsto 3) \end{array}$$

Reasoning rule for values

Example:

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

Rule:

$$\frac{}{\{\,[\,]\,\}\ v\ \{\lambda x.\,[x = v]\}}\ \textsc{val}$$

Exercise: state a reasoning rule for values using a heap implication.

$$\frac{\ldots\ \rhd\ \ldots}{\{H\}\ v\ \{Q\}}$$

Solution:

$$\frac{H\ \rhd\ Q\,v}{\{H\}\ v\ \{Q\}}\ \textsc{val-frame}$$

Derivability of the val-frame rule

$$\frac{H \rhd Q\, v}{\{H\}\ v\ \{Q\}}\ \textsc{val-frame}$$

Proof:

$$\dfrac{\dfrac{\dfrac{}{\{\,[\,]\,\}\ v\ \{\lambda x.\,[x = v]\}}\ \textsc{val}}{\{H\}\ v\ \{\lambda x.\,[x = v] \star H\}}\ \textsc{frame} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{H \rhd Q\, v}\ \textsc{hypothesis}}{\forall x.\ x = v \Rightarrow (H \rhd Q\, x)}\ \textsc{subst}}{\forall x.\ ([x = v] \star H) \rhd (Q\, x)}\ \textsc{prop-l}}{(\lambda x.\,[x = v] \star H) \rhd Q}\ \textsc{def of} \rhd}{\{H\}\ v\ \{Q\}}\ \textsc{conseq}$$

Reasoning rule for conditionals

Rule:

$$\frac{(v = \textsf{true} \;\Rightarrow\; \{H\}\ t_1\ \{Q\}) \qquad (v = \textsf{false} \;\Rightarrow\; \{H\}\ t_2\ \{Q\})}{\{H\}\ (\textsf{if}\, v \,\textsf{then}\, t_1 \,\textsf{else}\, t_2)\ \{Q\}}$$

Transformation to A-normal form:

$$(\textsf{if}\, t_0 \,\textsf{then}\, t_1 \,\textsf{else}\, t_2) \quad = \quad (\textsf{let}\, v = t_0 \,\textsf{in}\, (\textsf{if}\, v \,\textsf{then}\, t_1 \,\textsf{else}\, t_2))$$

Reasoning rule for top-level functions

Rule:

$$\frac{v_1 \;=\; \lambda x.\, t \qquad \{H\}\ ([x \to v_2]\, t)\ \{Q\}}{\{H\}\ (v_1\, v_2)\ \{Q\}}$$

Transformation to A-normal form:

$$(t_1\, t_2) \quad = \quad (\mathsf{let}\, f = t_1\, \mathsf{in}\, \mathsf{let}\, v = t_2\, \mathsf{in}\, (f\ v))$$

Verification of a simple function

```
let incr r =
  let a = !r in
  r := a+1
```

Specification:

$$\forall r n. \quad \{r \mapsto n\}\ (\texttt{incr r})\ \{\lambda\_.\ r \mapsto n+1\}$$

Verification:
Fix r and n. We need to prove that the body satisfies the specification:

$$\{r \mapsto n\}\ (\texttt{let a = !r in r := a+1})\ \{\lambda\_.\ r \mapsto n+1\}$$

We conclude using the let-binding rule: $Q' \equiv \lambda x. [x = n] \star (r \mapsto n)$.

Reasoning rule for top-level recursive functions

Rule:

$$\frac{v_1 \ = \ \mu f.\lambda x.t \qquad \{H\}\ ([f \to v_1]\,[x \to v_2]\,t)\ \{Q\}}{\{H\}\ (v_1\, v_2)\ \{Q\}}$$

Specification of recursive functions may be established by induction.

Verification of a recursive function

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

Specification:

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

We prove this specification by induction on L.
Consider p and L. Apply the "if" rule.

Verification of mlength: nil case

Case $p =$ **null**. Goal is:

$$\{p \leadsto \mathsf{Mlist}\, L\}\ (\mathsf{0})\ \{\lambda n.\ [n = |L|] \star p \leadsto \mathsf{Mlist}\, L\}$$

– Replace p with null.
– Rewrite null $\leadsto$ Mlist L to $[L = \mathsf{nil}]$ in the pre and the post.
– By the PROP rule:

$$L = \mathsf{nil} \ \ \Rightarrow\ \ \{[\,]\}\ (\mathsf{0})\ \{\lambda n.\ [n = |L|] \star [L = \mathsf{nil}]\}$$

– Replace L with nil.

$$\{[\,]\}\ (\mathsf{0})\ \{\lambda n.\ [n = |\mathsf{nil}|] \star [\mathsf{nil} = \mathsf{nil}]\}$$

– Apply the VAL-FRAME rule.

$$[\,] \rhd [0 = 0] \star [\mathsf{nil} = \mathsf{nil}]$$

Verification of mlength: cons case (1/2)

Case $p \neq$ **null**. Goal is:

$$\{p \rightsquigarrow \mathsf{Mlist}\, L\}$$
$$\big(\texttt{let p' = p.tl in let n' = mlength p' in 1 + n'}\big)$$
$$\{\lambda n.\ [n = |L|] \star p \rightsquigarrow \mathsf{Mlist}\, L\}$$

– Unfold Mlist in pre and post, and decompose L as $x :: L'$:

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

– Apply the let-binding rule, and the read axiom. Remains:

$$\{p' \rightsquigarrow \mathsf{Mlist}\, L' \star p \rightsquigarrow \{\!|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\!\}\ \}$$
$$\big(\texttt{let n' = mlength p' in 1 + n'}\big)$$
$$\{\lambda n.\ [n = |L|] \star p' \rightsquigarrow \mathsf{Mlist}\, L' \star p \rightsquigarrow \{\!|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\!\}\ \}$$

– Apply the frame rule to remove: $p \rightsquigarrow \{\!|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\!\}$.
– Apply the let-binding rule with : $Q \equiv \lambda n'. [n' = |L|'] \star p' \rightsquigarrow \mathsf{Mlist}\, L'$.

Verification of mlength: cons case (2/2)

There remains to prove the two premises of the let-rule.

– First branch, exploit the induction hypothesis:

$$\{p' \rightsquigarrow \mathsf{Mlist}\, L'\}\ (\texttt{mlength p'})\ \{\lambda n'.\ [n = |L|']\ \star\ p' \rightsquigarrow \mathsf{Mlist}\, L'\}$$

– Second branch:

$$\{p' \rightsquigarrow \mathsf{Mlist}\, L'\ \star\ [n' = |L|']\}\ (\texttt{1 + n'})\ \{\lambda n.\ [n = |L|]\ \star\ p' \rightsquigarrow \mathsf{Mlist}\, L'\}$$

– Apply the PROP rule and the VAL-FRAME rule.

$$n' = |L'| \quad \Rightarrow \quad p' \rightsquigarrow \mathsf{Mlist}\, L'\ \rhd\ [1 + n' = |L|]\ \star\ p' \rightsquigarrow \mathsf{Mlist}\, L'$$

– Cancel equal parts, conclude using $|L| = |x :: L'| = 1 + |L'| = 1 + n'$.

Reasoning rule for local functions

Rule template:

$$\frac{\forall f.\ (\ldots)\ \Rightarrow\ \{H\}\ t_2\ \{Q\}}{\{H\}\ (\mathsf{let\,rec}\ f\ x = t_1\ \mathsf{in}\ t_2)\ \{Q\}}$$

Hypothesis about f:

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

Rule:

$$\frac{\forall f.\ \ \big(\forall x H' Q'.\ \{H'\}\ t_1\ \{Q'\}\ \Rightarrow\ \{H'\}\ (f\ x)\ \{Q'\}\big)\ \ \Rightarrow\ \ \{H\}\ t_2\ \{Q\}}{\{H\}\ (\mathsf{let\,rec}\ f\ x = t_1\ \mathsf{in}\ t_2)\ \{Q\}}$$

Summary

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

$$\frac{\{H\}\ t_1\ \{Q'\} \qquad \forall x.\ \{Q'\,x\}\ t_2\ \{Q\}}{\{H\}\ (\mathsf{let}\, x = t_1\, \mathsf{in}\, t_2)\ \{Q\}}$$

$$\frac{v = \mathsf{true}\ \Rightarrow\ \{H\}\ t_1\ \{Q\} \qquad v = \mathsf{false}\ \Rightarrow\ \{H\}\ t_2\ \{Q\}}{\{H\}\ (\mathsf{if}\, v\, \mathsf{then}\, t_1\, \mathsf{else}\, t_2)\ \{Q\}}$$

$$\frac{\forall f.\ \big(\forall x H' Q'.\, \{H'\}\ t_1\ \{Q'\}\ \Rightarrow\ \{H'\}\ (f\ x)\ \{Q'\}\big)\ \Rightarrow\ \{H\}\ t_2\ \{Q\}}{\{H\}\ (\mathsf{let\ rec}\, f\ x = t_1\, \mathsf{in}\, t_2)\ \{Q\}}$$

Summary of Course 2

Summary of chapter 7

$$\frac{\{H_1\}\ t\ \{\lambda x.\, H_1'\}}{\{H_1 \star H_2\}\ t\ \{\lambda x.\, H_1' \star H_2\}}\ \textsc{frame}$$

In-place mutable list increment, when $L = x :: L'$.

$p \rightsquigarrow \mathsf{Mlist}\, L$		
$p \rightsquigarrow \{\!\vert \mathsf{hd}{=}x;\ \mathsf{tl}{=}p' \vert\!\} \quad \star\ p' \rightsquigarrow \mathsf{Mlist}\, L'$		by unfolding
$p \rightsquigarrow \{\!\vert \mathsf{hd}{=}x+1;\ \mathsf{tl}{=}p' \vert\!\} \ \star\ p' \rightsquigarrow \mathsf{Mlist}\, L'$		incrementing
$p' \rightsquigarrow \mathsf{Mlist}\, L'$		frame begins
$p' \rightsquigarrow \mathsf{Mlist}\, (\mathsf{map}\, (+1)\, L')$		by induction
$p \rightsquigarrow \{\!\vert \mathsf{hd}{=}x+1;\ \mathsf{tl}{=}p' \vert\!\} \ \star\ p' \rightsquigarrow \mathsf{Mlist}\, (\mathsf{map}\, (+1)\, L')$		frame ends
$p \rightsquigarrow \mathsf{Mlist}\, ((x+1) :: (\mathsf{map}\, (+1)\, L'))$		by folding
$p \rightsquigarrow \mathsf{Mlist}\, (\mathsf{map}\, (+1)\, L)$		by rewriting

Summary of chapter 8

Small footprint specification for C-style memory accesses:

$$\begin{array}{lll} \{p \mapsto w\} & (\texttt{*p = v}) & \{\lambda\_.\ p \mapsto v\} \\ \{p \mapsto v\} & (\texttt{*p}) & \{\lambda x.\ [x = v] \star p \mapsto v\} \end{array}$$

Representation of a full array using a list:

$$p \rightsquigarrow \mathsf{Array}\, L \quad \equiv \quad p.\mathsf{length} \mapsto |L| \star \underset{v \text{ at index } i \text{ in } L}{\circledast} p[i] \mapsto v$$

Representation of a set of array cells using a finite map:

$$p \rightsquigarrow \mathsf{Cells}\, M \quad \equiv \quad \underset{(i,v) \in M}{\circledast} p[i] \mapsto v$$

Summary of chapter 9

$(\star)$ is associative, commutative, and has $[\,]$ as neutral element.

$(\rhd)$ is a partial order, regular w.r.t. $(\star)$.

"$[\text{False}] \rhd H$" is always true.

"$(r \mapsto n) \star (r \mapsto m)$" is equivalent to "$[\text{False}]$".

Strategy: extract from the right, instantiate on the left, then cancel out.

Summary of chapter 10

Structural rules:

$$\frac{H \rhd H_1 \star H_2 \qquad \{H_1\}\ t\ \{Q_1\} \qquad Q_1 \star H_2 \rhd Q \star \mathsf{GC}}{\{H\}\ t\ \{Q\}}\ \textsc{combined}$$

$$\frac{\forall x.\ \{H\}\ t\ \{Q\}}{\{\exists x.\ H\}\ t\ \{Q\}}\ \textsc{exists} \qquad\qquad \frac{P \ \Rightarrow\ \{H\}\ t\ \{Q\}}{\{[P] \star H\}\ t\ \{Q\}}\ \textsc{prop}$$

Other structural rules are derivable.

Summary of chapter 11

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

$$\frac{\{H\}\ t_1\ \{Q'\} \qquad \forall x.\ \{Q'\,x\}\ t_2\ \{Q\}}{\{H\}\ (\mathsf{let}\,x = t_1\,\mathsf{in}\,t_2)\ \{Q\}}$$

$$\frac{v = \mathsf{true}\ \Rightarrow\ \{H\}\ t_1\ \{Q\} \qquad v = \mathsf{false}\ \Rightarrow\ \{H\}\ t_2\ \{Q\}}{\{H\}\ (\mathsf{if}\,v\,\mathsf{then}\,t_1\,\mathsf{else}\,t_2)\ \{Q\}}$$

$$\frac{\forall f.\ \big(\forall x H' Q'.\ \{H'\}\ t_1\ \{Q'\}\ \Rightarrow\ \{H'\}\ (f\,x)\ \{Q'\}\big)\ \Rightarrow\ \{H\}\ t_2\ \{Q\}}{\{H\}\ (\mathsf{let\,rec}\,f\,x = t_1\,\mathsf{in}\,t_2)\ \{Q\}}$$

Exercises

- Exam from 2015, Exercise 2: Operations on binary search trees.

Available from the webpage of the course.

The end!

Separation Logic
3/4

Arthur Charguéraud

Febuary 15th, 2016

Chapter 12

Loops in Separation Logic

Verification of a for-loop

```
let facto n =
  let r = ref 1 in
  for i = 2 to n do
    let v = !r in
    r := v * i;
  done;
  !r
```

Before the loop:

$$r \mapsto 1$$

At each iteration:

from $r \mapsto (i-1)!$ to $r \mapsto i!$

After the loop:

$$r \mapsto n!$$

Loop invariant $(I : \text{int} \to \text{Hprop})$ that applies for any $i \in [2, n+1]$:

$$I\,i \quad \equiv \quad r \mapsto (i-1)!$$

Reasoning rule for for-loops

Reasoning rule for the case $a \leqslant b$:

$$\frac{\begin{array}{c} H \rhd I\,a \\ \forall i \in [a,b].\quad \{I\,i\}\ t\ \{\lambda\_.\, I\,(i+1)\} \\ I\,(b+1) \rhd Q\,() \end{array}}{\{H\}\ (\mathsf{for}\, i = a\, \mathsf{to}\, b\, \mathsf{do}\, t)\ \{Q\}}$$

General rule, also covering the case $a > b$:

$$\frac{\begin{array}{c} H \rhd I\,a \\ \forall i \in [a,b].\quad \{I\,i\}\ t\ \{\lambda\_.\, I\,(i+1)\} \\ I\,(\mathsf{max}\, a\,(b+1)) \rhd Q\,() \end{array}}{\{H\}\ (\mathsf{for}\, i = a\, \mathsf{to}\, b\, \mathsf{do}\, t)\ \{Q\}}$$

Reasoning rule for while loops: partial correctness

The loop invariant I describes the state between every iterations.
The post-condition J describes the state after the evaluation of t_1.

$$\frac{H \rhd I \qquad \{I\}\ t_1\ \{J\} \qquad \{J\,\mathsf{true}\}\ t_2\ \{\lambda\_.\, I\} \qquad J\,\mathsf{false} \rhd Q\,()}{\{H\}\ (\mathsf{while}\, t_1\, \mathsf{do}\, t_2)\ \{Q\}}$$

where $(I : \mathsf{Hprop})$ and $(J : \mathsf{bool} \to \mathsf{Hprop})$.

For total correctness: parameterize the invariant with a measure.

Reasoning rule for while loops

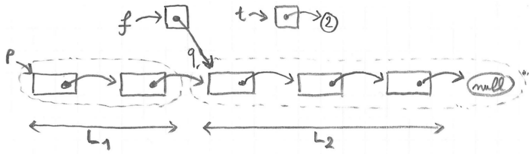
We focus on a different approach that:

- inherently supports total correctness;
- allows to apply frame during iterations.

Prove a triple $\{H\}\ (\mathsf{while}\, t_1 \,\mathsf{do}\, t_2)\ \{Q\}$ by induction, using:

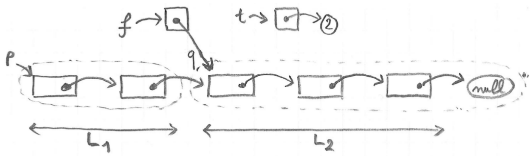
$$\frac{\{H\}\ (\mathsf{if}\, t_1 \,\mathsf{then}\, (t_2\, ;\ (\mathsf{while}\, t_1 \,\mathsf{do}\, t_2))\, \mathsf{else}\, ())\ \{Q\}}{\{H\}\ (\mathsf{while}\, t_1 \,\mathsf{do}\, t_2)\ \{Q\}}$$

Length with a while loop



```
let rec mlength (p:'a cell) =
  let t = ref 0 in
  let f = ref p in
  while !f != null do
    incr t;
    f := (!f).tl;
  done
  !t
```

Length with a while loop: induction



We prove by induction on L_2 that for any n and q:

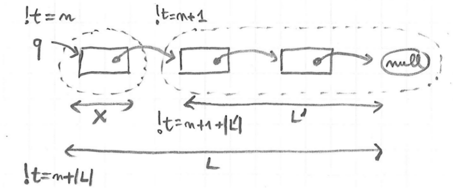
$$\{q \rightsquigarrow \mathsf{Mlist}\, L_2 \star f \mapsto q \star t \mapsto n\}$$
$$(\texttt{while !f != null do incr t; f := (!f).tl; done})$$
$$\{q \rightsquigarrow \mathsf{Mlist}\, L_2 \star f \mapsto \mathsf{null} \star t \mapsto (n + \mathsf{length}\, L_2)\}$$

The loop unfolds to:

```
if !f != null
  then (incr t; f := (!f).tl; while .. do .. done)
  else ()
```

Exercise: describe the frame process in the induction for `length`.

Length with a while loop: frame process



$q \leadsto \mathsf{Mlist}\, L_2$	$\star\ f \mapsto q$	$\star\ t \mapsto n$	begin
$q \mapsto \{\mid x; q' \mid\} \star q' \leadsto \mathsf{Mlist}\, L_2'$	$\star\ f \mapsto q$	$\star\ t \mapsto n$	unfold
$q \mapsto \{\mid x; q' \mid\} \star q' \leadsto \mathsf{Mlist}\, L_2'$	$\star\ f \mapsto q$	$\star\ t \mapsto n+1$	increment
$q \mapsto \{\mid x; q' \mid\} \star q' \leadsto \mathsf{Mlist}\, L_2'$	$\star\ f \mapsto q'$	$\star\ t \mapsto n+1$	shift head
$\star\ q' \leadsto \mathsf{Mlist}\, L_2'$	$\star\ f \mapsto q'$	$\star\ t \mapsto n+1$	begin frame
$\star\ q' \leadsto \mathsf{Mlist}\, L_2'$	$\star\ f \mapsto \mathsf{null}$	$\star\ t \mapsto n+1+\lvert L_2' \rvert$	induction
$q \mapsto \{\mid x; q' \mid\} \star q' \leadsto \mathsf{Mlist}\, L_2'$	$\star\ f \mapsto \mathsf{null}$	$\star\ t \mapsto n+1+\lvert L_2' \rvert$	end frame
$q \leadsto \mathsf{Mlist}\, L_2$	$\star\ f \mapsto \mathsf{null}$	$\star\ t \mapsto n+\lvert L_2 \rvert$	fold

Chapter 13

Aliasing and local state

Functions with aliasing: swap

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

Find three useful specifications for `swap`:

1. a specification for non-aliased (distinct) arguments,
2. a specification for aliased (equal) arguments,
3. a most-general specification, stated using iterated conjunction.

Functions with aliasing: 3 specifications for swap

Specification 1:

$$\forall r s n m.\ \{(r \mapsto n) \star (s \mapsto m)\}\ (\texttt{swap r s})\ \{\lambda\_.\ (r \mapsto m) \star (s \mapsto n)\}$$

Specification 2:

$$\forall r s n.\ \{[r = s] \star (r \mapsto n)\}\ (\texttt{swap r s})\ \{\lambda\_.\ r \mapsto n\}$$

or simply:

$$\forall r n.\ \{r \mapsto n\}\ (\texttt{swap r r})\ \{\lambda\_.\ r \mapsto n\}$$

Specification 3:

$$\begin{aligned}\forall r s M.\ r, s \in \mathsf{dom}\, M \ \Rightarrow\ & \{\textstyle\circledast_{(p,n)\in M}\ p \mapsto n\} \\ & (\texttt{swap r s}) \\ & \{\lambda\_.\ \textstyle\circledast_{(p,n)\in(M[r:=M[s]][s:=M[r]])}\ p \mapsto n\}\end{aligned}$$

Function with local state

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

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

Then, show that the code below returns 5.

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

Specification:

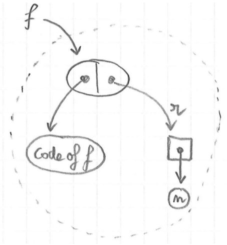
$$\forall n. \quad \{r \mapsto n\}\, (\texttt{f ()})\, \{\lambda\_.\ r \mapsto n+1\}$$

Successive states:

$$r \mapsto 3 \qquad r \mapsto 4 \qquad r \mapsto 5$$

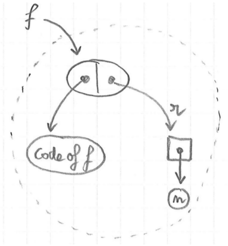
Counter function: code

```
let mkcounter () =
  let r = ref 0 in
  (fun () -> incr r; get r)
```



```
let c = mkcounter() in
let x = c() in
let y = c() in
assert (x = 1 && y = 2)
```

Counter function: specification



$$f \leadsto \mathsf{Count}\, n \;\equiv\; \exists r.\, (r \mapsto n) \star [\forall i.\, \{r \mapsto i\}\, (f\ ())\, \{\lambda x.\, [x = i+1] \star (r \mapsto i+1)\}]$$

Exercise: specify a counter function, only in terms of $f \leadsto \mathsf{Count}\, n$.

$$\{[\,]\}\ (\texttt{mkcounter()})\ \{\lambda f.\ f \leadsto \mathsf{Count}\, 0\}$$

$$\forall f i.\quad \{f \leadsto \mathsf{Count}\, i\}\ (f\ ())\ \{\lambda x.\, [x = i+1] \star f \leadsto \mathsf{Count}\,(i+1)\}$$

Chapter 14

Basic higher-order functions

Apply

```
let apply f x =
  f x
```

Specification:

$$\begin{array}{rl} \forall f x H Q. & \{H\}\ (f\ x)\ \{Q\} \\ \Rightarrow & \{H\}\ (\texttt{apply}\ f\ x)\ \{Q\} \end{array}$$

This is equivalent to the form below, which involves nested triples:

$$\forall f x H Q. \quad \{H \star [\ \{H\}\ (f\ x)\ \{Q\}\ ]\}\ (\texttt{apply}\ f\ x)\ \{Q\}$$

Apply on a reference

```
let refapply r f =
  r := f !r
```

Exercise: give two specifications for the function `refapply`.
In the first, assume `f` to be pure, and introduce a predicate $P\,x\,y$.
In the second, assume that `f` also modifies the state from H to H'.

$$\begin{array}{ll} \forall r f x P. & \{[\,]\}\;(f\,x)\;\{\lambda y.\,[P\,x\,y]\} \\ \quad\Rightarrow & \{r \mapsto x\}\;(\texttt{refapply}\,r\,f)\;\{\lambda\_.\;\exists y.\,[P\,x\,y] \star r \mapsto y\} \end{array}$$

$$\begin{array}{ll} \forall r f x H H' P. & \{H\}\;(f\,x)\;\{\lambda y.\,[P\,x\,y] \star H'\} \\ \quad\Rightarrow & \{(r \mapsto x) \star H\} \\ & (\texttt{refapply}\,r\,f) \\ & \{\lambda\_.\;\exists y.\,[P\,x\,y] \star (r \mapsto y) \star H'\} \end{array}$$

Function twice

```
let twice f =
  f(); f()
```

Specification:

$$\begin{array}{lrl} \forall f\, H'\, Q. & & \{H\}\ (f\,())\ \{\lambda\_.\ H'\} \\ & \wedge & \{H'\}\ (f\,())\ \{Q\} \\ & \Rightarrow & \{H\}\ (\texttt{twice}\, f)\ \{Q\} \end{array}$$

Function repeat

```
let repeat n f =
  for i = 0 to n-1 do
    f()
  done
```

Exercise: specify `repeat`, using an invariant I, of type int → Hprop.

$$\begin{array}{lll} \forall n f I. & & (\forall i \in [0,n)). \quad \{I\,i\}\;(f\,())\;\{\lambda\_.\;I\,(i+1)\}) \\ & \Rightarrow & \{I\,0\}\;(\texttt{repeat}\,n\,f)\;\{\lambda\_.\;I\,n\} \end{array}$$

The premise consists of a family of hypotheses describing the behavior of applications of f to particular arguments.

Chapter 15

Higher order iteration

Iteration over a pure list

```
let rec iter f l =
  match l with
  | [] -> ()
  | x::t -> f x; iter f t
```

Exercise: specify `iter`, using an invariant I, of type list $\alpha \rightarrow$ Hprop.

$$\begin{array}{ll} \forall f l I. & \big(\forall x k.\ \{I\,k\}\ (f\,x)\ \{\lambda\_.\ I\,(k \,\&\, x)\}\big) \\ \Rightarrow & \{I\,\mathsf{nil}\}\ (\mathtt{iter}\,f\,l)\ \{\lambda\_.\ I\,l\} \end{array}$$

where $k \,\&\, x \;\equiv\; k +\!\!+ (x :: \mathsf{nil})$.

Length using iter

$$\begin{array}{rl} & \big(\forall x k.\ \{I\,k\}\ (f\,x)\ \{\lambda\_.\ I\,(k\&x)\}\big) \\ \Rightarrow & \{I\ \mathsf{nil}\}\ (\mathsf{iter}\,f\,l)\ \{\lambda\_.\ I\,l\} \end{array}$$

```
let length l =
  let r = ref 0 in
  iter (fun x -> incr r) l;
  !r
```

Exercise: give the instantiatiation of the invariant I for `iter`;
then, write the specialization of the specification of `iter` to I and to `(fun x -> incr r)`; finally, check that the premise is provable.

Invariant: $I \equiv \lambda k.\ r \mapsto |k|$.

$$\begin{array}{rl} & \big(\forall x k.\ \{r \mapsto |k|\}\ (\texttt{incr r})\ \{\lambda\_.\ r \mapsto |k| + 1\}\big) \\ \Rightarrow & \{r \mapsto 0\}\ (\mathsf{iter}\,f\,l)\ \{\lambda\_.\ r \mapsto |l|\} \end{array}$$

Sum using iter

$$\begin{array}{rl} & \big(\forall x k.\ \{I\,k\}\ (f\,x)\ \{\lambda\_.\ I\,(k\&x)\}\big) \\ \Rightarrow & \{I\,\mathsf{nil}\}\ (\mathsf{iter}\,f\,l)\ \{\lambda\_.\ I\,l\} \end{array}$$

```
let sum l =
  let r = ref 0 in
  iter (fun x -> r := !r + x) l;
  !r
```

Exercise: give the invariant I involved in the above call to `iter`.

$$I \ \equiv\ \lambda k.\ r \mapsto \mathsf{Sum}\,k$$

where:

$$\mathsf{Sum}\,k \ \equiv\ \mathsf{Fold}\,(+)\,0\,k$$

Constraints over the items

$$\begin{array}{rl} & \big(\forall x k.\ \{I\, k\}\ (f\, x)\ \{\lambda\_.\ I\,(k\&x)\}\big) \\ \Rightarrow & \{I\, \mathsf{nil}\}\ (\mathsf{iter}\, f\, l)\ \{\lambda\_.\ I\, l\} \end{array}$$

Given a list $x_1 :: x_2 :: ... :: x_n :: \mathsf{nil}$, let us compute: $\sqrt{x_1} + ... + \sqrt{x_n}$.

```
iter (fun x -> r := !r +. sqrt x) [2.0; 3.0]
```

The above specification of `iter` is too weak. More general specification:

$$\begin{array}{rl} \forall f I l. & \big(\forall x k.\ x \in l\ \Rightarrow\ \{I\, k\}\ (f\, x)\ \{\lambda\_.\ I\,(k\&x)\}\big) \\ \Rightarrow & \{I\, \mathsf{nil}\}\ (\mathsf{iter}\, f\, l)\ \{\lambda\_.\ I\, l\} \end{array}$$

Constraints over the items, in order

$$\begin{array}{ll} \forall f I l. & \big(\forall x k.\ x \in l \Rightarrow \{I\,k\}\ (f\,x)\ \{\lambda\_.\ I\,(k \& x)\}\big) \\ & \Rightarrow\ \ \{I\ \mathsf{nil}\}\ (\mathsf{iter}\ f\ l)\ \{\lambda\_.\ I\,l\} \end{array}$$

Given a list $x_1 :: x_2 :: \ldots :: x_n :: \mathsf{nil}$, let us compute: $\sqrt{\sqrt{\sqrt{x_1} + x_2} + x_3}$.

```
iter (fun x -> r := sqrt (!r +. x)) [2.; -1.; 3.]
```

The above specification of `iter` is too weak. Most-general specification:

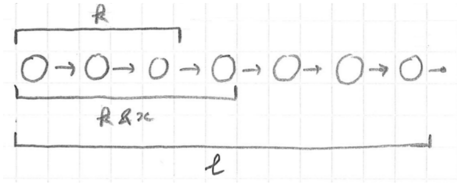
$$\begin{array}{ll} \forall f I l. & \big(\forall x k s.\ l = k +\!\!+ x :: s \Rightarrow \{I\,k\}\ (f\,x)\ \{\lambda\_.\ I\,(k \& x)\}\big) \\ & \Rightarrow\ \ \{I\ \mathsf{nil}\}\ (\mathtt{iter}\ f\ l)\ \{\lambda\_.\ I\,l\} \end{array}$$

Verification of iter

$$\begin{array}{ll} & \big(\forall x k.\ \{I\,k\}\ (f\,x)\ \{\lambda\_.\ I\,(k\&x)\}\big) \\ \Rightarrow & \{I\ \mathsf{nil}\}\ (\mathsf{iter}\ f\ l)\ \{\lambda\_.\ I\,l\} \end{array}$$

```
let rec iter f l =
  match l with
  | [] -> ()
  | x::t -> f x; iter f t
```

How to prove that the code satisfies its specification?



Verification of iter: generalized principle

Assume:

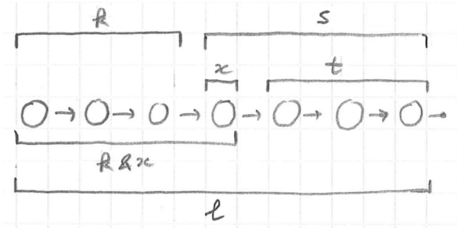
$$\forall x k.\ \{I\, k\}\ (f\, x)\ \{\lambda \_.\ I\, (k \& x)\}$$

Prove:

$$\{I\, \mathsf{nil}\}\ (\mathsf{iter}\, f\, l)\ \{\lambda \_.\ I\, l\}$$

Proof by induction over a generalized statement:

$$\forall k s.\quad \{I\, k\}\ (\mathsf{iter}\, f\, s)\ \{\lambda \_.\ I\, (k +\!\!+ s)\}$$



Verification of iter: induction

```
let rec iter f s =
  match s with
  | [] -> ()
  | x::t -> f x; iter f t
```

Assume: $\forall x k.\quad \{I\,k\}\;(f\;x)\;\{\lambda\_.\;I\,(k\&x)\}$

Prove: $\forall k s.\quad \{I\,k\}\;(\mathsf{iter}\,f\;s)\;\{\lambda\_.\;I\,(k \mathbin{+\!\!+} s)\}$

By induction on s:

- Case $s = \mathsf{nil}$. Goal is: $\{I\,k\}\;(\mathsf{iter}\,f\;\mathsf{nil})\;\{\lambda\_.\,I\,(k \mathbin{+\!\!+} \mathsf{nil})\}$.
  This triple simplifies to: $\{I\,k\}\;()\;\{\lambda\_.\,I\,k\}$, which is correct.
- Case $s = x :: t$. Goal is: $\{I\,k\}\;(\mathsf{iter}\,f\;(x :: t))\;\{\lambda\_.\,I\,(k \mathbin{+\!\!+} (x :: t))\}$.

$$\dfrac{\dfrac{\text{HYPOTHESIS-ON-F}}{\{I\,k\}\;(f\;x)\;\{\lambda\_.\;I\,(k\&x)\}} \qquad \dfrac{\text{INDUCTION-HYPOTHESIS}}{\{I\,(k\&x)\}\;(\mathsf{iter}\,f\;t)\;\{\lambda\_.\;I\,((k\&x) \mathbin{+\!\!+} t)\}}}{\{I\,k\}\;(f\;x;\;\mathsf{iter}\,f\;t)\;\{I\,((k\&x) \mathbin{+\!\!+} t)\}}\;\text{SEQ}$$

Invariant on remaining items

$$\begin{aligned} & \big(\forall x k.\ \{I\,k\}\ (f\,x)\ \{\lambda\_.\ I\,(k\&x)\}\big) \\ \Rightarrow\quad & \{I\,\mathsf{nil}\}\ (\mathsf{iter}\,f\,l)\ \{\lambda\_.\ I\,l\} \end{aligned}$$

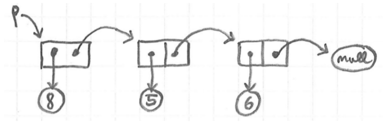
$$\begin{aligned} & \big(\forall \ldots.\ \{\ldots\}\ (f\,x)\ \{\lambda\_.\ \ldots\}\big) \\ \Rightarrow\quad & \{I'\,l\}\ (\mathsf{iter}\,f\,l)\ \{\lambda\_.\ I'\,\mathsf{nil}\} \end{aligned}$$

Exercise: specify `iter` using an invariant that depends on the list of items remaining to process, instead of on the list of items already processed. Then, prove the new specification derivable from the old one.

$$\begin{aligned} & \big(\forall x s.\ \{I'\,(x :: s)\}\ (f\,x)\ \{\lambda\_.\ I'\,s\}\big) \\ \Rightarrow\quad & \{I'\,l\}\ (\mathsf{iter}\,f\,l)\ \{\lambda\_.\ I'\,\mathsf{nil}\} \end{aligned}$$

Derivable using: $I \equiv \lambda k.\ \exists s.\,[l = k +\!\!+ s] \star I'\,s.$

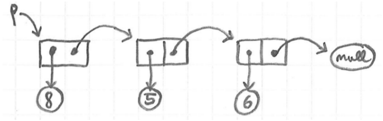
Iterating over a mutable list



```
let rec miter f p =
  if p == null
    then ()
    else (f p.hd; miter f p.tl)
```

Iterating over a mutable list

$$\begin{array}{ll} \forall f l I. & \big(\forall x k.\ \{I\,k\}\ (f\,x)\ \{\lambda\_.\ I\,(k\&x)\}\big) \\ \Rightarrow & \{I\ \mathsf{nil}\}\ (\mathsf{iter}\ f\ l)\ \{\lambda\_.\ I\ l\} \end{array}$$



Specification:

$$\begin{array}{ll} \forall f p I l. & \big(\forall x k.\ \{I\,k\}\ (f\,x)\ \{\lambda\_.\ I\,(k\&x)\}\big) \\ \Rightarrow & \{p \rightsquigarrow \mathsf{Mlist}\,l\ \star\ I\ \mathsf{nil}\}\ (\mathtt{miter}\ f\ p)\ \{\lambda\_.\ p \rightsquigarrow \mathsf{Mlist}\,l\ \star\ I\ l\} \end{array}$$

Remark: calls to f cannot modify the structure of the list while iterating.

Summary

Simplified:

$$\big(\forall x k.\ \{I\,k\}\ (f\,x)\ \{\lambda\_.\ I\,(k\&x)\}\big) \Rightarrow \{I\ \mathsf{nil}\}\ (\mathsf{iter}\,f\,l)\ \{\lambda\_.\ I\,l\}$$

Order-irrelevant:

$$\big(\forall x k.\ x \in l \Rightarrow \{I\,k\}\ (f\,x)\ \{\lambda\_.\ I\,(k\&x)\}\big) \Rightarrow \{I\ \mathsf{nil}\}\ (\mathsf{iter}\,f\,l)\ \{\lambda\_.\ I\,l\}$$

Most-general:

$$\big(\forall x k s.\ l = k +\!\!+ x :: s \Rightarrow \{I\,k\}\ (f\,x)\ \{\lambda\_.\ I\,(k\&x)\}\big) \Rightarrow \{I\ \mathsf{nil}\}\ (\mathsf{iter}\,f\,l)\ \{\lambda\_.\ I\,l\}$$

Extension to mutable lists:

$$\big(\forall x k s.\ l = k +\!\!+ x :: s \Rightarrow \{I\,k\}\ (f\,x)\ \{\lambda\_.\ I\,(k\&x)\}\big) \Rightarrow \{p \leadsto \mathsf{Mlist}\,l \star I\ \mathsf{nil}\}\ (\mathtt{miter}\,f\,p)\ \{\lambda\_.\ p \leadsto \mathsf{Mlist}\,l \star I\,l\}$$

Chapter 16

Other classic higher-order functions

Fold-left

```
let rec fold_left f a l =
  match l with
  | [] -> a
  | x::k -> fold_left f (f a x) k
```

Example:

$$\texttt{fold\_left}\, f\, a\, [6 :: 4 :: 7] \;=\; f\,(f\,(f\,a\,6)\,4)\,7$$

Specification:

$$\begin{array}{rl} \forall falJ. & \big(\forall xik.\ \{J\,i\,k\}\,(f\,i\,x)\,\{\lambda j.\ J\,j\,(k\&x)\}\big) \\ \Rightarrow & \{J\,a\,\mathsf{nil}\}\,\big(\texttt{fold\_left}\,f\,a\,l\big)\,\{\lambda b.\ J\,b\,l\} \end{array}$$

Application of fold-left

$$\forall f a l J. \quad \big(\forall x i k.\ \{J\,i\,k\}\,(f\,i\,x)\,\{\lambda j.\ J\,j\,(k\&x)\}\big) \Rightarrow \{J\,a\,\mathsf{nil}\}\,(\texttt{fold\_left}\,f\,a\,l)\,\{\lambda b.\ J\,b\,l\}$$

```
let r = ref 0
let count_and_sum l =
  fold_left (fun a x -> incr r; a+x) 0 l
```

Exercise: give the instantiation of the invariant J in the code above.

$$J\,i\,k \ \equiv\ (r \mapsto |k|) \star [i = \mathsf{Sum}\,k]$$

where $\mathsf{Sum}\,k \ \equiv\ \mathsf{Fold}\,(+)\,0\,k$.

Fold-right

```
let rec fold_right f l a =
  match l with
  | [] -> a
  | x::k -> f x (fold_right f k a)
```

Example:

$$\text{fold}\, f\, [6 :: 4 :: 7]\, a \;\equiv\; f\,6\,(f\,4\,(f\,7\,a))$$

Exercise: give a specification for `fold_right`.

$$\forall f l a J. \qquad \big(\forall x i k.\ \{J\,i\,k\}\ (f\,x\,i)\ \{\lambda j.\ J\,j\,(x :: k)\}\big)$$
$$\Rightarrow \quad \{J\,a\,\text{nil}\}\ (\texttt{fold\_right}\,f\,l\,a)\ \{\lambda b.\ J\,b\,l\}$$

Map: simple specification for pure functions

```
let rec map f l =
  match l with
  | [] -> []
  | x::k -> (f x)::(map f k)
```

Simple specification, for the case where `f` is pure:

$$\begin{array}{rl} \forall f l P. & (\forall x.\ \{[\,]\}\ (f\,x)\ \{\lambda x'.\ [P\,x\,x']\}) \\ \Rightarrow & \{[\,]\}\ (\texttt{map}\,f\,l)\ \{\lambda l'.\ [\mathsf{Forall2}\,P\,l\,l']\} \end{array}$$

where:

$$\frac{}{\mathsf{Forall2}\,P\,\mathsf{nil}\,\mathsf{nil}} \qquad \frac{P\,x\,x' \qquad \mathsf{Forall2}\,P\,l\,l'}{\mathsf{Forall2}\,P\,(x :: l)\,(x' :: l')}$$

Map: general specification

Specification of `map`:

$$\begin{array}{lrl} \forall f l P. & & (\forall x.\ \{[\,]\}\ (f\ x)\ \{\lambda x'.\,[P\,x\,x']\}) \\ & \Rightarrow & \{[\,]\}\ (\mathsf{map}\,f\ l)\ \{\lambda l'.\ [\mathsf{Forall2}\,P\,l\,l']\} \end{array}$$

Specification of `iter`:

$$\begin{array}{lrl} \forall f l I. & & \big(\forall x k.\ \{I\,k\}\ (f\ x)\ \{\lambda\_.\ I\,(k\&x)\}\big) \\ & \Rightarrow & \{I\,\mathsf{nil}\}\ (\mathsf{iter}\,f\ l)\ \{\lambda\_.\ I\,l\} \end{array}$$

Combining the two:

$$\begin{array}{lrl} \forall f l P I. & & \big(\forall x k.\ \{I\,k\}\ (f\ x)\ \{\lambda x'.\,[P\,x\,x'] \star I\,(k\&x)\}\big) \\ & \Rightarrow & \{I\,\mathsf{nil}\}\ (\mathsf{map}\,f\ l)\ \{\lambda l'.\ [\mathsf{Forall2}\,P\,l\,l'] \star I\,l\} \end{array}$$

Map: general specification, alternative

$$\begin{array}{rl} \forall f l P I. & \big(\forall x k.\ \{I\,k\}\ (f\,x)\ \{\lambda x'.\,[P\,x\,x'] \star J\,(k\&x)\}\big) \\ \Rightarrow & \{I\ \mathsf{nil}\}\ (\mathsf{map}\,f\,l)\ \{\lambda l'.\ [\mathsf{Forall2}\,P\,l\,l'] \star I\,l\} \end{array}$$

Alternative specification:

$$\begin{array}{rl} \forall f l J'. & \big(\forall x k k'.\ \{J'\,k\,k'\}\ (f\,x)\ \{\lambda x'.\,J'\,(k\&x)\,(k'\&x')\}\big) \\ \Rightarrow & \{J'\,\mathsf{nil}\,\mathsf{nil}\}\ (\mathsf{map}\,f\,l)\ \{\lambda l'.\ J'\,l\,l'\} \end{array}$$

Above specification derivable from the previous one:

$$J'\,k\,k' \;\equiv\; [\mathsf{Forall2}\,P\,k\,k'] \star I\,k$$

Sorting with comparison function

Example:

```
List.sort (fun x y -> x - y) [2;4;5;3;2;9]
```

Specification:

$$\begin{array}{ll} \forall f l.\ \forall(\preceq). & \\ & \text{total-order}\,(\preceq) \\ \wedge & (\forall xy.\ \{[\,]\}\ (f\,x\,y)\ \{\lambda n.\,[n \leqslant 0 \Leftrightarrow x \preceq y]\}) \\ \Rightarrow & \{[\,]\}\ (\texttt{sort}\,f\,l)\ \{\lambda l'.\ [\text{permut}\,l\,l' \ \wedge\ \text{sorted}\,(\preceq)\,l']\} \end{array}$$

More general specification of comparison functions:

$$\{[\,]\}\ (f\,x\,y)\ \{\lambda n.\,[\text{if}\,n = 0\,\text{then}\,x \approx y\,\text{else if}\,n < 0\,\text{then}\,x \prec y\,\text{else}\,x \succ y]\}$$

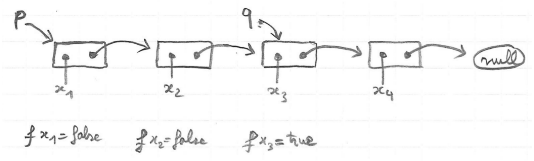
Find with a boolean predicate, on pure lists

```
let rec find f l =
  match l with
  | [] -> None
  | x::k -> if f x
              then Some x
              else find f k
```

Specification:

$$
\begin{array}{rl}
\forall f l P. & (\forall x.\ \{[\,]\}\ (f\ x)\ \{\lambda b.\ [b = \mathsf{true} \Leftrightarrow P\,x]\}) \\
\Rightarrow & \{[\,]\}\ (\texttt{find}\ f\ l)\ \{\lambda o.\ [\ \begin{array}{l} \mathsf{match}\ o\ \mathsf{with} \\ \mid \mathsf{None} \Rightarrow \mathsf{Forall}\,(\neg P)\,l \\ \mid \mathsf{Some}\,x \Rightarrow \exists k t.\ l = k +\!\!+\, x :: t \\ \qquad\qquad \wedge\ \mathsf{Forall}\,(\neg P)\,k\ \wedge\ P\,x \end{array} \ ]\}
\end{array}
$$

Find with a boolean predicate, on mutable lists



Specification:

$$
\begin{array}{ll}
\forall f p l P. & (\forall x.\ \{[\,]\}\ (f\ x)\ \{\lambda b.\,[b = \mathsf{true} \Leftrightarrow P\,x]\}) \\
\Rightarrow & \{p \leadsto \mathsf{Mlist}\,l\} \\
& (\mathtt{mfind}\ f\ p) \\
& \{\lambda o.\,\mathsf{match}\ o\ \mathsf{with} \qquad\qquad\qquad\qquad\qquad\qquad \} \\
& \quad \mid \mathsf{None} \Rightarrow p \leadsto \mathsf{Mlist}\,l\ \star\ [\mathsf{Forall}\,(\neg P)\,l] \\
& \quad \mid \mathsf{Some}\,q \Rightarrow \exists kt.\ p \leadsto \mathsf{MlistSeg}\,q\,k\ \star\ q \leadsto \mathsf{Mlist}\,(x :: t) \\
& \qquad\qquad \star\,[l = k +\!\!+ x :: t\ \wedge\ \mathsf{Forall}\,(\neg P)\,k\ \wedge\ P\,x]
\end{array}
$$

Summary

$$\begin{array}{rl} & \big(\forall x k.\ \{I\,k\}\ (f\,x)\ \{\lambda\_.\ I\,(k\&x)\}\big) \\ \Rightarrow & \{I\ \mathsf{nil}\}\ (\mathsf{iter}\,f\,l)\ \{\lambda\_.\ I\,l\} \end{array}$$

$$\begin{array}{rl} & \big(\forall x i k.\ \{J\,i\,k\}\ (f\,i\,x)\ \{\lambda j.\ J\,j\,(k\&x)\}\big) \\ \Rightarrow & \{J\,a\ \mathsf{nil}\}\ (\mathsf{fold}\,f\,a\,l)\ \{\lambda b.\ J\,b\,l\} \end{array}$$

$$\begin{array}{rl} & \big(\forall x k k'.\ \{J\,k\,k'\}\ (f\,x)\ \{\lambda x'.\,J\,(k\&x)\,(k'\&x')\}\big) \\ \Rightarrow & \{J\ \mathsf{nil}\ \mathsf{nil}\}\ (\mathsf{map}\,f\,l)\ \{\lambda l'.\ J\,l\,l'\} \end{array}$$

- Add the hypothesis $l = k +\!\!+\, x :: s$ if the position of x matters.

- Boolean predicates: $\forall x.\ \{[\,]\}\ (f\,x)\ \{\lambda b.\,[b = \mathsf{true} \Leftrightarrow P\,x]\}$.
- Order functions: $\forall x y.\ \{[\,]\}\ (f\,x\,y)\ \{\lambda n.\,[n \leqslant 0 \Leftrightarrow x \preceq y]\}$.

Chapter 17

Principle of Characteristic Formulae

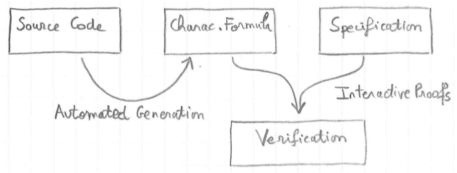
Idea of characteristic formulae

Goal: perform all the Separation Logic reasoning inside Coq.

Idea: build a logical formula $[\![t]\!]$ satisfying the equivalence below.

$$\forall H Q. \quad [\![t]\!]\, H\, Q \;\Leftrightarrow\; \{H\}\, t\, \{Q\}$$

Schema:



Properties of characteristic formulae

The characteristic formula $[\![t]\!]$ of a term t is a predicate such that:

$$\forall HQ. \quad [\![t]\!]\, H\, Q \;\Leftrightarrow\; \{H\}\, t\, \{Q\}$$

Properties:

- $[\![t]\!]$ has type (Heap → Prop) → (Val → Heap → Prop) → Prop
- $[\![t]\!]$ characterizes the set of valid specifications for t
- $[\![t]\!]$ is a higher-order logic formula built using $\wedge$, $\vee$, $\Rightarrow$, $\exists$, ...
- $[\![t]\!]$ is built automatically and compositionally
- $[\![t]\!]$ has size linear in the size of t and is easy to read

Characteristic formula for sequence

$$\forall H Q.\quad [\![t]\!]\, H\, Q \;\Leftrightarrow\; \{H\}\ t\ \{Q\}$$

$$\frac{\{H\}\ t_1\ \{Q'\} \qquad \{Q'\,()\}\ t_2\ \{Q\}}{\{H\}\ (t_1\,;\,t_2)\ \{Q\}}\ \textsc{seq}$$

Goal:

$$\forall H Q.\quad [\![t_1\,;\,t_2]\!]\, H\, Q \;\Leftrightarrow\; \{H\}\ (t_1\,;\,t_2)\ \{Q\}$$

Exercise: define the characteristic formula for sequences.

$$[\![t_1\,;\,t_2]\!] \;\equiv\; \lambda H.\,\lambda Q.\;\exists Q'.\ [\![t_1]\!]\, H\, Q' \,\wedge\, [\![t_2]\!]\, (Q'\,())\, Q$$

Characteristic formula for let bindings

$$\frac{\{H\}\ t_1\ \{Q'\} \qquad \forall x.\ \{Q'\,x\}\ t_2\ \{Q\}}{\{H\}\ (\mathsf{let}\,x = t_1\,\mathsf{in}\,t_2)\ \{Q\}}$$

Definition:

$$[\![\mathsf{let}\,x = t_1\,\mathsf{in}\,t_2]\!] \ \equiv\ \lambda HQ.\ \exists Q'.\ [\![t_1]\!]\,H\,Q' \ \wedge\ \forall x.\ [\![t_2]\!]\,(Q'\,x)\,Q$$

Technically, x has type var and:

$$\begin{aligned}[\![\mathsf{let}\,x = t_1\,\mathsf{in}\,t_2]\!] \ \equiv\ \lambda HQ.\ \exists Q'.\quad & [\![t_1]\!]\,H\,Q' \\ & \wedge\ \forall (X : \mathsf{Val}).\ [\![([x \to X]\,t_2)]\!]\,(Q'\,X)\,Q\end{aligned}$$

Characteristic formula for values

$$\frac{H \rhd Q\,v}{\{H\}\ v\ \{Q\}}\ \textsc{val-frame}$$

Definition:

$$[\![v]\!] \;\equiv\; \lambda HQ.\ H \rhd Q\,v$$

Characteristic formula for conditionals

$$\frac{(b = \mathsf{true} \;\Rightarrow\; \{H\}\ t_1\ \{Q\}) \qquad (b = \mathsf{false} \;\Rightarrow\; \{H\}\ t_2\ \{Q\})}{\{H\}\ (\mathsf{if}\, b\, \mathsf{then}\, t_1\, \mathsf{else}\, t_2)\ \{Q\}}\ \textsc{if}$$

Exercise: define the characteristic formula for conditionals.

$$\begin{array}{rcll} [\![\mathsf{if}\, b\, \mathsf{then}\, t_1\, \mathsf{else}\, t_2]\!] & \equiv & \lambda HQ. & \quad (b = \mathsf{true} \Rightarrow [\![t_1]\!]\, H\, Q) \\ & & \wedge & \quad (b = \mathsf{false} \Rightarrow [\![t_2]\!]\, H\, Q) \end{array}$$

The App predicate

The goal of characteristic formulae is do proofs without involving triples.

Let "App" be an abstract predicate with the following interpretation:

$$\mathsf{App}\, f\, v\, H\, Q \quad \Leftrightarrow \quad \{H\}\ (f\ v)\ \{Q\}$$

Remark:

$$\mathsf{App} \ \ : \ \ \mathsf{Val} \to \mathsf{Val} \to \mathsf{Hprop} \to (\mathsf{Val} \to \mathsf{Hprop}) \to \mathsf{Prop}$$

Reasoning about function calls

Interpretation of App: $\mathsf{App}\,f\,v\,H\,Q \quad \Leftrightarrow \quad \{H\}\,(f\,v)\,\{Q\}$

Interpretation of formulae: $[\![f\,v]\!]\,H\,Q \quad \Leftrightarrow \quad \{H\}\,(f\,v)\,\{Q\}$

Definition:

$$[\![f\,v]\!] \;\equiv\; \lambda HQ.\;\mathsf{App}\,f\,v\,H\,Q$$

Instances of App are used on calls and introduced on function definitions.

Reasoning about function definitions

$$\frac{\forall f.\ \ P f \ \Rightarrow\ \{H\}\ t_2\ \{Q\} \qquad P f\ =\ \big(\forall x H' Q'.\ \{H'\}\ t_1\ \{Q'\}\ \Rightarrow\ \{H'\}\ (f\,x)\ \{Q'\}\big)}{\{H\}\ (\mathsf{let\,rec}\ f\ x = t_1\ \mathsf{in}\ t_2)\ \{Q\}}\ \textsc{fix}$$

Definition:

$$[\![\mathsf{let\,rec}\ f = \lambda x.\, t_1\ \mathsf{in}\ t_2]\!]\ \ \equiv\ \ \lambda H Q.\ \ \forall f.\ \ P f\ \Rightarrow\ [\![t_2]\!]\,H\,Q$$

$$\text{where}\ \ P f\ \equiv\ (\forall x H' Q'.\ [\![t_1]\!]\,H'\,Q'\ \Rightarrow\ \mathsf{App}\,f\,x\,H'\,Q')$$

Complete definition

$$\begin{array}{lcl}
[\![v]\!] & \equiv & \lambda HQ.\ \ H \rhd Q\,v \\[1em]
[\![\mathsf{let}\,x = t_1\,\mathsf{in}\,t_2]\!] & \equiv & \lambda HQ.\ \ \exists Q'.\ [\![t_1]\!]\,H\,Q' \ \wedge\ \forall x.\ [\![t_2]\!]\,(Q'\,x)\,Q \\[1em]
[\![\mathsf{if}\,b\,\mathsf{then}\,t_1\,\mathsf{else}\,t_2]\!] & \equiv & \lambda HQ.\ \ \qquad (b = \mathsf{true} \Rightarrow [\![t_1]\!]\,H\,Q) \\
 & & \qquad\quad\ \wedge\ \ (b = \mathsf{false} \Rightarrow [\![t_2]\!]\,H\,Q) \\[1em]
[\![v_1\,v_2]\!] & \equiv & \lambda HQ.\ \ \mathsf{App}\,v_1\,v_2\,H\,Q \\[1em]
[\![\mathsf{let\,rec}\,f = \lambda x.\,t_1\,\mathsf{in}\,t_2]\!] & \equiv & \lambda HQ.\ \ \forall f.\ P f \ \Rightarrow\ [\![t_2]\!]\,H\,Q
\end{array}$$

$$\text{where } P f \ \equiv\ (\forall x H' Q'.\ [\![t_1]\!]\,H'\,Q' \ \Rightarrow\ \mathsf{App}\,f\,x\,H'\,Q')$$

The end!

Separation Logic
4/4

Arthur Charguéraud

Febuary 22th, 2016

Chapter 18

Characteristic Formulae with structural rules

Integration of structural rules

$$
\begin{array}{lcl}
[\![v]\!] & \equiv & \text{local}\,(\lambda HQ.\ \ H \rhd Q\,v) \\
[\![\text{let}\,x = t_1\,\text{in}\,t_2]\!] & \equiv & \text{local}\,(\lambda HQ.\ \ \exists Q'.\ [\![t_1]\!]\,H\,Q' \\
 & & \qquad\qquad \wedge\, \forall x.\ [\![t_2]\!]\,(Q'\,x)\,Q\ ) \\
[\![\text{if}\,b\,\text{then}\,t_1\,\text{else}\,t_2]\!] & \equiv & \text{local}\,(\lambda HQ.\ \ \ \ (b = \text{true} \Rightarrow [\![t_1]\!]\,H\,Q) \\
 & & \qquad\quad \wedge\ \ (b = \text{false} \Rightarrow [\![t_2]\!]\,H\,Q)\ ) \\
[\![v_1\,v_2]\!] & \equiv & \text{local}\,(\lambda HQ.\ \ \text{App}\,v_1\,v_2\,H\,Q) \\
[\![\text{let rec}\,f = \lambda x.\,t_1\,\text{in}\,t_2]\!] & \equiv & \text{local}\,(\lambda HQ.\ \ \forall f.\ P f \Rightarrow [\![t_2]\!]\,H\,Q)
\end{array}
$$

$$\text{where } P f \equiv (\forall x H' Q'.\ [\![t_1]\!]\,H'\,Q' \Rightarrow \text{App}\,f\,x\,H'\,Q')$$

Definition of the local predicate (1/2)

To support:

$$\frac{H = H_1 \star H_2 \qquad [\![t]\!]\, H_1\, Q_1 \qquad Q_1 \star H_2 = Q}{[\![t]\!]\, H\, Q}\ \text{FRAME},$$

we would define:

$$\mathsf{local}\, \mathcal{F} \ \equiv\ \lambda H Q.\ \exists H_1 H_2 Q_1. \begin{cases} H = H_1 \star H_2 \\ \mathcal{F}\, H_1\, Q_1 \\ Q_1 \star H_2 = Q \end{cases}$$

Framing using the local predicate

To prove "$\{H\}\ t\ \{Q\}$", by the rule FRAME', it suffices to show:

$$H = H_1 \star H_2 \quad \land \quad \{H_1\}\ t\ \{Q_1\} \quad \land \quad Q_1 \star H_2 = Q$$

To prove "local $[\![t]\!]\ H\ Q$", by definition of "local", it suffices to show:

$$H = H_1 \star H_2 \quad \land \quad [\![t]\!]\ H_1\ Q_1 \quad \land \quad Q_1 \star H_2 = Q$$

Definition of the local predicate (2/2)

To support:

$$\frac{H \rhd H_1 \star H_2 \qquad \{H_1\}\ t\ \{Q_1\} \qquad Q_1 \star H_2 \rhd Q \star \mathsf{GC}}{\{H\}\ t\ \{Q\}}\ \textsc{combined}$$

$$\frac{\forall x.\ \{H\}\ t\ \{Q\}}{\{\exists x.\, H\}\ t\ \{Q\}}\ \textsc{exists} \qquad\qquad \frac{P \ \Rightarrow\ \{H\}\ t\ \{Q\}}{\{[P] \star H\}\ t\ \{Q\}}\ \textsc{prop}$$

we define:

$$\mathsf{local}\,\mathcal{F} \ \equiv\ \lambda HQ.\ \forall h.\ H\,h \Rightarrow \exists H_1 H_2 Q_1.\left\{\begin{array}{l}(H_1 \star H_2)\,h \\ \mathcal{F}\,H_1\,Q_1 \\ Q_1 \star H_2 \rhd Q \star \mathsf{GC}\end{array}\right.$$

Iterated applications of structural rules

The local predicate may be duplicated as many times as needed:

$$\mathsf{local}\,[\![t]\!]\,H\,Q \;=\; \mathsf{local}\,(\mathsf{local}\,[\![t]\!])\,H\,Q$$

For example, to prove "$\mathsf{local}\,[\![t]\!]\,H\,Q$", it suffices to show:

$$H = H_1 \star H_2 \;\wedge\; \mathsf{local}\,[\![t]\!]\,H_1\,Q_1 \;\wedge\; Q_1 \star H_2 = Q$$

When not needed, "local" may be simply erased:

$$[\![t]\!]\,H\,Q \;\Rightarrow\; \mathsf{local}\,[\![t]\!]\,H\,Q$$

Notation for characteristic formulae

$$[\![\mathsf{let}\, x = t_1 \,\mathsf{in}\, t_2]\!] \;\equiv\; \mathsf{local}\,(\lambda H Q.\;\; \exists Q'.\; [\![t_1]\!]\, H\, Q' \;\wedge\; \forall x.\; [\![t_2]\!]\,(Q'\,x)\,Q)$$

Definition of Coq notation:

$$(\mathsf{Let}\; x = \mathcal{F}_1 \;\mathsf{in}\; \mathcal{F}_2) \;\equiv\; \mathsf{local}\,(\lambda H Q.\;\; \exists Q'.\; \mathcal{F}_1\, H\, Q' \;\wedge\; \forall x.\; \mathcal{F}_2\,(Q'\,x)\,Q)$$

With this notation:

$$[\![\mathsf{let}\, x = t_1 \,\mathsf{in}\, t_2]\!] \;\equiv\; (\mathsf{Let}\; x = [\![t_1]\!] \;\mathsf{in}\; [\![t_2]\!])$$

Technically:

$$[\![\mathsf{let}\, x = t_1 \,\mathsf{in}\, t_2]\!] \;\equiv\; (\mathsf{Let}\; X = [\![t_1]\!] \;\mathsf{in}\; [\![([x \to X]\, t_2)]\!])$$

Characteristic formulae generation, with notation

$$\begin{array}{lcl}
[\![v]\!] & \equiv & \mathsf{Ret}\ v \\
[\![\mathsf{let}\, x = t_1 \,\mathsf{in}\, t_2]\!] & \equiv & \mathsf{Let}\ x = [\![t_1]\!]\ \mathsf{in}\ [\![t_2]\!] \\
[\![\mathsf{if}\, b \,\mathsf{then}\, t_1 \,\mathsf{else}\, t_2]\!] & \equiv & \mathsf{If}\ b\ \mathsf{then}\ [\![t_1]\!]\ \mathsf{else}\ [\![t_2]\!] \\
[\![v_1\, v_2]\!] & \equiv & \mathsf{App}\, v_1\, v_2 \\
[\![\mathsf{let\,rec}\, f = \lambda x.\, t_1 \,\mathsf{in}\, t_2]\!] & \equiv & \mathsf{Let\,Rec}\, f\, x\ =\ [\![t_1]\!]\ \mathsf{in}\ [\![t_2]\!]
\end{array}$$

Tactics for characteristic formulae

What the user sees:

$$\text{Let } x = \mathcal{F}_1 \text{ in } \mathcal{F}_2$$

What is hidden behind the notation:

$$\text{local}\,(\lambda H Q.\ \ \exists Q'.\ \mathcal{F}_1\, H\, Q' \ \wedge\ \forall x.\ \mathcal{F}_2\,(Q'\,x)\,Q)$$

What the user would need to execute:

```
apply local_erase; esplit; split.
```

What the user writes:

```
xlet.
```

Chapter 19

Higher-order representation predicates

Overview

1. Higher-order predicate:

   $$p \rightsquigarrow \mathsf{Mlist}\, L \qquad \text{is generalized into} \qquad p \rightsquigarrow \mathsf{Mlistof}\, R\, L$$

2. Identity representation predicate:

   $$p \rightsquigarrow \mathsf{Mlistof}\, \mathsf{Id}\, L \qquad \text{is the same as} \qquad p \rightsquigarrow \mathsf{Mlist}\, L$$

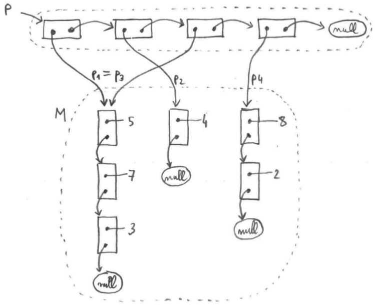
3. Control accesses:

   $$\{p \rightsquigarrow \mathsf{Mcellof}\, \mathsf{Id}\, v_1\, R_2\, V_2\}\ (p.\mathsf{hd})\ \{\lambda x.\, [x = v_1] \star \ldots\}$$

4. Compose recursively:

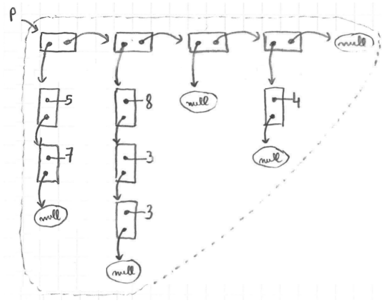
   $$p \rightsquigarrow \mathsf{Nodeof}\, R\, X\, (\mathsf{Mlistof}\, (\mathsf{Narytreeof}\, R))\, L$$

Mutable list of possibly-aliased lists



$$p \rightsquigarrow \mathsf{Mlist}\, K \;\star\; \left(\underset{(p_i, L_i) \in M}{\circledast} p_i \rightsquigarrow \mathsf{Mlist}\, L_i \right) \;\star\; [\forall p_i \in K.\;\; p_i \in \mathsf{dom}\, M]$$

Mutable list of disjoint mutable lists

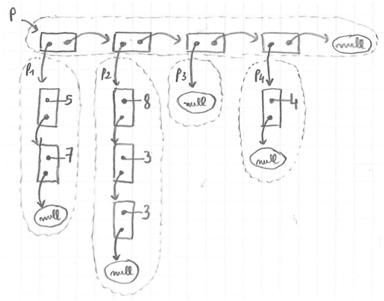


$$L \;=\; (5::7::\mathsf{nil})::(8::3::3::\mathsf{nil})$$
$$::(\mathsf{nil})::(4::\mathsf{nil})\;::\;\mathsf{nil}$$

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

(to be later generalized into: $p \rightsquigarrow \mathsf{Mlistof}\, R\, L$)

Representation using iterated star

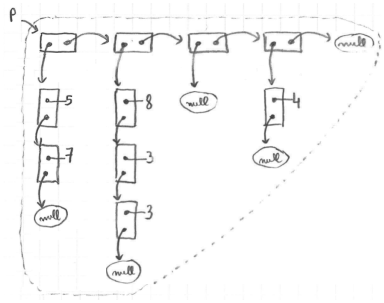


$$L = (5 :: 7 :: \mathsf{nil}) :: (8 :: 3 :: 3 :: \mathsf{nil}) :: (\mathsf{nil}) :: (4 :: \mathsf{nil}) :: \mathsf{nil}$$

$$K = p_1 :: p_2 :: p_3 :: p_4 :: \mathsf{nil}$$

$$p \leadsto \mathsf{MlistofMlist}\, L \quad \equiv \quad \exists K. \quad p \leadsto \mathsf{Mlist}\, K \\ \star \ \circledast_{i \,\in\, [0, |L|)} \ (K[i]) \leadsto \mathsf{Mlist}\, (L[i]) \\ \star \ [\, |K| = |L|\,]$$

Representation using a recursive predicate



$$L \;=\; (5 :: 7 :: \mathsf{nil}) :: (8 :: 3 :: 3 :: \mathsf{nil}) \\ :: (\mathsf{nil}) :: (4 :: \mathsf{nil}) \;::\; \mathsf{nil}$$

$$\begin{aligned} p \leadsto \mathsf{MlistofMlist}\, L \;\;\equiv\;\; & \mathsf{match}\; L\; \mathsf{with} \\ & \mid \mathsf{nil} \;\Rightarrow\; [p = \mathsf{null}] \\ & \mid X :: L' \;\Rightarrow\; \exists x p'.\;\; p \leadsto \{\!|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\!\} \\ & \qquad\qquad\qquad\quad \star\; p' \leadsto \mathsf{MlistofMlist}\, L' \\ & \qquad\qquad\qquad\quad \star\; x \leadsto \mathsf{Mlist}\, X \end{aligned}$$

Generalization to a higher-order predicate

$$\begin{array}{rcl} p \rightsquigarrow \mathsf{MlistofMlist}\, L & \equiv & \mathsf{match}\, L\, \mathsf{with} \\ & & \mid \mathsf{nil} \;\Rightarrow\; [p = \mathsf{null}] \\ & & \mid X :: L' \;\Rightarrow\; \exists x p'.\;\; p \rightsquigarrow \{\!|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\!\} \\ & & \qquad\qquad\qquad\quad \star\; p' \rightsquigarrow \mathsf{MlistofMlist}\, L' \\ & & \qquad\qquad\qquad\quad \star\; x \rightsquigarrow \mathsf{Mlist}\, X \end{array}$$

Generalization:

$$\begin{array}{rcl} p \rightsquigarrow \mathsf{Mlistof}\, R\, L & \equiv & \mathsf{match}\, L\, \mathsf{with} \\ & & \mid \mathsf{nil} \;\Rightarrow\; [p = \mathsf{null}] \\ & & \mid X :: L' \;\Rightarrow\; \exists x p'.\;\; p \rightsquigarrow \{\!|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\!\} \\ & & \qquad\qquad\qquad\quad \star\; p' \rightsquigarrow \mathsf{Mlistof}\, R\, L' \\ & & \qquad\qquad\qquad\quad \star\; x \rightsquigarrow R\, X \end{array}$$

In particular:

$$p \rightsquigarrow \mathsf{MlistofMlist}\, L \;=\; p \rightsquigarrow \mathsf{Mlistof}\, \mathsf{Mlist}\, L$$

Type-checking

$p \rightsquigarrow \mathsf{Mlistof}\, R\, L$	*is a notation for*	$\mathsf{Mlistof}\, R\, L\, p$	(of type Hprop)
$x \rightsquigarrow R\, X$	*is a notation for*	$R\, X\, x$	(of type Hprop)

$$
\begin{array}{lll}
p \rightsquigarrow \mathsf{Mlistof}\, R\, L & \equiv & \mathsf{match}\, L\, \mathsf{with} \\
& & \mid \mathsf{nil} \Rightarrow [p = \mathsf{null}] \\
& & \mid X :: L' \Rightarrow \exists x p'.\quad \begin{array}{l} \; p \rightsquigarrow \{\!|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\!\} \\ \star\; p' \rightsquigarrow \mathsf{Mlistof}\, R\, L' \\ \star\; x \rightsquigarrow R\, X \end{array}
\end{array}
$$

Exercise: since $(p : \mathsf{loc})$ and $(x : \mathsf{Val})$ and $(X : A)$ for some A, what is the type of R? What is the type of Mlistof?

- $R : A \to \mathsf{Val} \to \mathsf{Hprop}$
- $\mathsf{Mlistof} : \forall A.\ (A \to \mathsf{Val} \to \mathsf{Hprop}) \to \mathsf{list} A \to \mathsf{loc} \to \mathsf{Hprop}$

The identity representation predicate

$$p \rightsquigarrow \mathsf{Mlistof}\, R\, L \;\equiv\; \begin{array}{l} \mathsf{match}\; L\; \mathsf{with} \\ \mid \mathsf{nil} \;\Rightarrow\; [p = \mathsf{null}] \\ \mid X :: L' \;\Rightarrow\; \exists x p'.\quad \begin{array}{l} p \rightsquigarrow \{\!|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\!\} \\ \star\; p' \rightsquigarrow \mathsf{Mlistof}\, R\, L' \\ \star\; x \rightsquigarrow R\, X \end{array} \end{array}$$

$$p \rightsquigarrow \mathsf{Mlist}\, L \;\equiv\; \begin{array}{l} \mathsf{match}\; L\; \mathsf{with} \\ \mid \mathsf{nil} \;\Rightarrow\; [p = \mathsf{null}] \\ \mid x :: L' \;\Rightarrow\; \exists p'.\quad \begin{array}{l} p \rightsquigarrow \{\!|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\!\} \\ \star\; p' \rightsquigarrow \mathsf{Mlist}\, L' \end{array} \end{array}$$

Exercise: define the identity representation predicate Id such that

$$p \rightsquigarrow \mathsf{Mlistof}\, \mathsf{Id}\, L \;=\; p \rightsquigarrow \mathsf{Mlist}\, L$$

Definition:

$$x \rightsquigarrow \mathsf{Id}\, X \;\equiv\; [x = X]$$

Summary

1. Higher-order predicate:

$$p \rightsquigarrow \mathsf{Mlist}\, L \qquad \text{is generalized into} \qquad p \rightsquigarrow \mathsf{Mlistof}\, R\, L$$

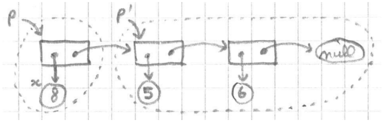
2. Identity representation predicate:

$$p \rightsquigarrow \mathsf{Mlistof}\, \mathsf{Id}\, L \qquad \text{is the same as} \qquad p \rightsquigarrow \mathsf{Mlist}\, L$$

Chapter 20

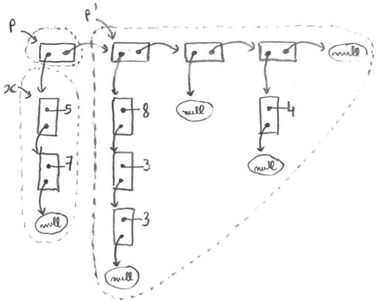
Higher-order representation predicates and the access problem

Specification of construction, for basic values



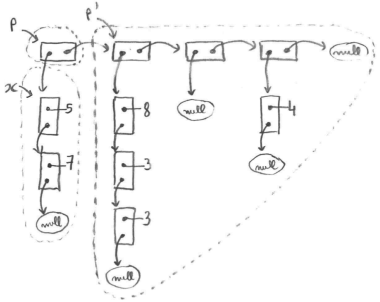
$$\{p' \leadsto \mathsf{Mlist}\, L\}\ (\mathsf{cons}\, x\, p')\ \{\lambda p.\ p \leadsto \mathsf{Mlist}\,(x :: L)\}$$

Specification of construction



$$\{x \rightsquigarrow R\,X \,\star\, p' \rightsquigarrow \mathsf{Mlistof}\,R\,L\}\ (\mathsf{cons}\,x\,p')\ \{\lambda p.\ p \rightsquigarrow \mathsf{Mlistof}\,R\,(X :: L)\}$$

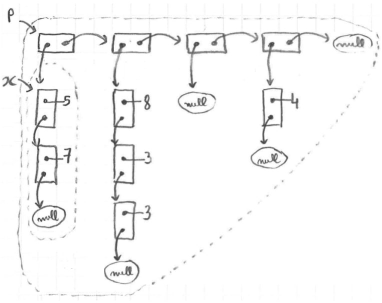
Specification of deconstruction



$\{p \rightsquigarrow \mathsf{Mlistof}\, R\, (X :: L)\}$ $(\mathsf{uncons}\, p)$

$\{\lambda(x, p').\ x \rightsquigarrow R\, X \star p' \rightsquigarrow \mathsf{Mlistof}\, R\, L\}$

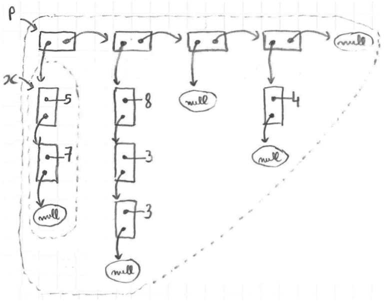
Specification of accesses: the problem



Incorrect specification for `head`:

$$\{p \leadsto \mathsf{Mlistof}\, R\,(X :: L)\}\ (\mathsf{head}\, p)$$
$$\{\lambda x.\ x \leadsto R\, X\ \star\ p \leadsto \mathsf{Mlistof}\, R\,(X :: L)\}$$

Specification of accesses: a partial solution



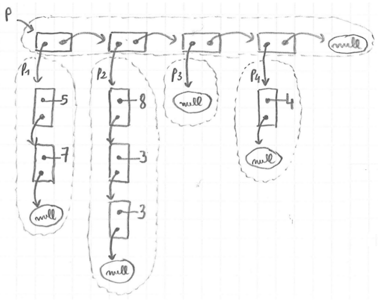
Correct yet limited specification:

$$\{p \rightsquigarrow \mathsf{Mlistof}\, R\,(X :: L)\}\ (\texttt{head}\, p)$$
$$\{\lambda x.\ x \rightsquigarrow R\,X \star (x \rightsquigarrow R\,X \mathrel{-\!\!\star} p \rightsquigarrow \mathsf{Mlistof}\, R\,(X :: L))\}$$

Magic wand rule:

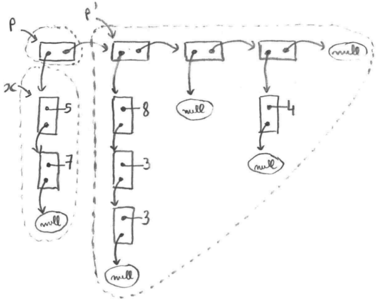
$$H \star (H \mathrel{-\!\!\star} H') = H'$$

Specification of accesses: a brute force solution



$$p \rightsquigarrow \mathsf{Mlistof}\, R\, L \quad = \quad \exists K. \quad p \rightsquigarrow \mathsf{Mlist}\, K$$
$$\star \quad \circledast_{i \,\in\, [0,\,|L|)} \; (K[i]) \rightsquigarrow R\,(L[i])$$
$$\star \quad [\, |K| = |L| \,]$$

Specification of accesses: focus before read



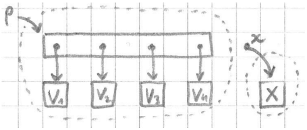
$$
\begin{aligned}
p \rightsquigarrow \mathsf{Mlistof}\, R\, (X :: L) \;\; = \;\; \exists x p'. \quad & p \rightsquigarrow \{\!|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\!\} \\
\star\;\; & x \rightsquigarrow R\, X \\
\star\;\; & p' \rightsquigarrow \mathsf{Mlistof}\, R\, L'
\end{aligned}
$$

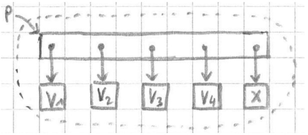
Then read using:

$$\{p \mapsto \{\!|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\!\}\}\ (p.\mathsf{hd})\ \{\lambda y.\ [y = x] \star p \mapsto \{\!|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\!\}\}$$

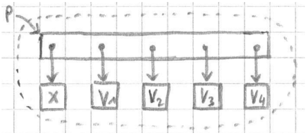
Ownership transfer with a queue of mutable items

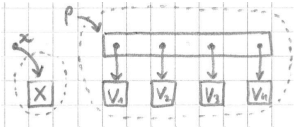
Push:





Pop:





Specification of queues of basic items

$\{[\,]\}\ (\texttt{create()})\ \{\lambda p.\ p \rightsquigarrow \mathsf{Queue}\,\mathsf{nil}\}$

$\{p \rightsquigarrow \mathsf{Queue}\,L\}\ (\texttt{push x p})\ \{\lambda\_.\ p \rightsquigarrow \mathsf{Queue}\,(L\&x)\}$

$\{p \rightsquigarrow \mathsf{Queue}\,(x :: L)\}\ (\texttt{pop p})\ \{\lambda r.\ [r = x] \star p \rightsquigarrow \mathsf{Queue}\,L\}$

$\{p \rightsquigarrow \mathsf{Queue}\,L \star p' \rightsquigarrow \mathsf{Queue}\,L'\}\ (\texttt{concat p p'})\ \{\lambda\_.\ p \rightsquigarrow \mathsf{Queue}\,(L +\!\!+ L')\}$

Specification of queues of mutable items

Exercise: specify functions over queues using a higher-order representation predicate written $p \rightsquigarrow \mathsf{Queueof}\, R\, L$.
Shorthand: just write "Q R" instead of "Queueof R".

$$\{[\,]\}\ (\texttt{create()})\ \{\lambda p.\ p \rightsquigarrow \mathsf{Queueof}\, R\, \mathsf{nil}\}$$

$$\{p \rightsquigarrow \mathsf{Queueof}\, R\, L \star x \rightsquigarrow R\, X\}\ (\texttt{push}\, x\, p)\ \{\lambda\_.\ p \rightsquigarrow \mathsf{Queueof}\, R\, (L \& X)\}$$

$$\{p \rightsquigarrow \mathsf{Queueof}\, R\, (X :: L)\}\ (\texttt{pop}\, p)\ \{\lambda x.\ p \rightsquigarrow \mathsf{Queueof}\, R\, L \star x \rightsquigarrow R\, X\}$$

$$\{p \rightsquigarrow \mathsf{Queueof}\, R\, L \star p' \rightsquigarrow \mathsf{Queueof}\, R\, L'\}\ (\texttt{concat}\, p\, p')$$
$$\{\lambda\_.\ p \rightsquigarrow \mathsf{Queueof}\, R\, (L \mathbin{+\!\!+} L')\}$$

The copy problem

Incorrect specification for `copy`:

$$\begin{array}{l}\{p \rightsquigarrow \mathsf{Queueof}\,R\,L\}\\ (\mathsf{copy}\,p)\\ \{\lambda p'.\ p \rightsquigarrow \mathsf{Queueof}\,R\,L \,\star\, p' \rightsquigarrow \mathsf{Queueof}\,R\,L\}\end{array}$$

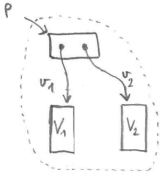
Exercise: specify a function `copy` $f\,p$ that duplicables a mutable queue specified using Queueof, where f is a function to duplicate items.

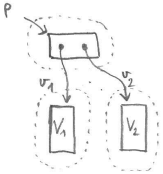
$$\begin{array}{rl} & \big(\forall x X.\ \{x \rightsquigarrow R\,X\}\ (f\ x)\ \{\lambda x'.\ x \rightsquigarrow R\,X \,\star\, x' \rightsquigarrow R\,X\}\big)\\ \Rightarrow & \{p \rightsquigarrow \mathsf{Queueof}\,R\,L\}\\ & (\mathsf{copy}\,f\,p)\\ & \{\lambda p'.\ p \rightsquigarrow \mathsf{Queueof}\,R\,L \,\star\, p' \rightsquigarrow \mathsf{Queueof}\,R\,L\}\end{array}$$

Chapter 21

Higher-order representation predicates for records

Representation for records





$$p \rightsquigarrow \mathsf{Mcellof}\, R_1\, V_1\, R_2\, V_2 \;\equiv\; \exists v_1 v_2.\quad \begin{array}{l} p \rightsquigarrow \{\!|\, \mathsf{hd}{=}v_1;\ \mathsf{tl}{=}v_2 \,|\!\} \\ \star\ v_1 \rightsquigarrow R_1\, V_1 \\ \star\ v_2 \rightsquigarrow R_2\, V_2 \end{array}$$

Representation predicate for lists, revisited

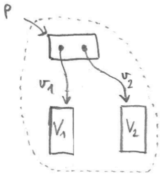
$$
\begin{array}{l}
p \rightsquigarrow \mathsf{Mlistof}\, R\, L \;\equiv\; \mathsf{match}\, L\, \mathsf{with} \\
\qquad \mid \mathsf{nil} \;\Rightarrow\; [p = \mathsf{null}] \\
\qquad \mid X :: L' \;\Rightarrow\; \exists x p'.\quad p \rightsquigarrow \{\!|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\!\} \\
\qquad\qquad \star\; x \rightsquigarrow R\, X \\
\qquad\qquad \star\; p' \rightsquigarrow \mathsf{Mlistof}\, R\, L'
\end{array}
$$

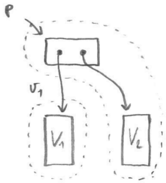
$$
\begin{array}{l}
p \rightsquigarrow \mathsf{Mcellof}\, R_1\, V_1\, R_2\, V_2 \;\equiv\; \exists v_1 v_2.\quad p \rightsquigarrow \{\!|\mathsf{hd}{=}v_1;\ \mathsf{tl}{=}v_2|\!\} \\
\qquad \star\; v_1 \rightsquigarrow R_1\, V_1 \\
\qquad \star\; v_2 \rightsquigarrow R_2\, V_2
\end{array}
$$

Exercise: rewrite the specification of Mlistof using Mcellof.

$$
\begin{array}{l}
p \rightsquigarrow \mathsf{Mlistof}\, R\, L \;\equiv\; \mathsf{match}\, L\, \mathsf{with} \\
\qquad \mid \mathsf{nil} \;\Rightarrow\; [p = \mathsf{null}] \\
\qquad \mid X :: L' \;\Rightarrow\; p \rightsquigarrow \mathsf{Mcellof}\, R\, X\, (\mathsf{Mlistof}\, R)\, L'
\end{array}
$$

Focus/unfocus for accessing a record field





Focus on a field:

$$p \leadsto \mathsf{Mcellof}\, R_1\, V_1\, R_2\, V_2 \quad = \quad \exists v_1.\ p \leadsto \mathsf{Mcellof}\, \mathsf{Id}\, v_1\, R_2\, V_2 \,\star\, v_1 \leadsto R_1\, V_1$$

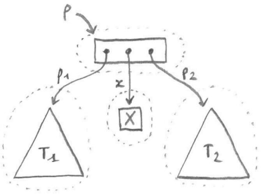
Access to a focused field:

$$\{p \leadsto \mathsf{Mcellof}\, \mathsf{Id}\, v_1\, R_2\, V_2\}\ (p.\mathsf{hd})\ \{\lambda x.\, [x = v_1] \,\star\, p \leadsto \mathsf{Mcellof}\, \mathsf{Id}\, v_1\, R_2\, V_2\}$$
$$\{p \leadsto \mathsf{Mcellof}\, \mathsf{Id}\, v_1\, R_2\, V_2\}\ (p.\mathsf{hd} \,\texttt{<-}\, w)\ \{\lambda\_.\ p \leadsto \mathsf{Mcellof}\, \mathsf{Id}\, w\, R_2\, V_2\}$$

Chapter 22

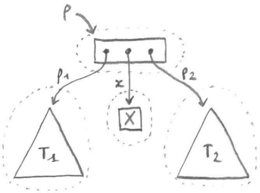
Higher-order representation predicates for trees

Binary tree: representation



$$
\begin{array}{l}
p \rightsquigarrow \mathsf{Mtreeof}\, R\, T \;\equiv\; \mathsf{match}\, T\, \mathsf{with} \\
\qquad\qquad\qquad\qquad\; |\, \mathsf{Leaf} \Rightarrow [p = \mathsf{null}] \\
\qquad\qquad\qquad\qquad\; |\, \mathsf{Node}\, X\, T_1\, T_2 \Rightarrow \exists x p_1 p_2. \\
\qquad\qquad\qquad\qquad\qquad\qquad p \mapsto \{|\mathsf{item}{=}x;\ \mathsf{left}{=}p_1;\ \mathsf{right}{=}p_2|\} \\
\qquad\qquad\qquad\qquad\qquad \star\ x \rightsquigarrow R\, X \\
\qquad\qquad\qquad\qquad\qquad \star\ p_1 \rightsquigarrow \mathsf{Mtreeof}\, R\, T_1 \\
\qquad\qquad\qquad\qquad\qquad \star\ p_2 \rightsquigarrow \mathsf{Mtreeof}\, R\, T_2
\end{array}
$$

Binary tree: representation, revisited

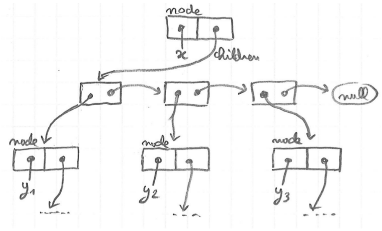


Representation predicate for tree cells:

$$p \leadsto \mathsf{Nodeof}\, R_1\, V_1\, R_2\, V_2\, R_3\, V_3 \quad \equiv$$
$$\exists v_1 v_2 v_3. \quad p \mapsto \{\!|\mathsf{item}{=}v_1;\ \mathsf{left}{=}v_2;\ \mathsf{right}{=}v_3|\!\}$$
$$\star\ v_1 \leadsto R_1\, V_1\ \star\ v_2 \leadsto R_2\, V_2\ \star\ v_3 \leadsto R_3\, V_3$$

$$p \leadsto \mathsf{Mtreeof}\, R\, T \;\equiv\; \mathsf{match}\, T\, \mathsf{with}$$
$$\mid \mathsf{Leaf} \;\Rightarrow\; [p = \mathsf{null}]$$
$$\mid \mathsf{Node}\, X\, T_1\, T_2 \;\Rightarrow$$
$$p \leadsto \mathsf{Nodeof}\, R\, X\, (\mathsf{Mtreeof}\, R)\, T_1\, (\mathsf{Mtreeof}\, R)\, T_2$$

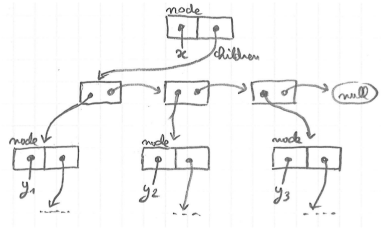
Trees with list of subtrees: implementation



```
type 'a node = {
  mutable item : 'a;
  mutable children : ('a node) cell }
```

```
Inductive tree (A:Type) : Type :=
  | Leaf : tree A
  | Node : A → list (tree A) → tree A.
```

Trees with list of subtrees: specification



$p \rightsquigarrow \mathsf{Narytreeof}\, R\, T \equiv$

$\quad \mathsf{match}\, T\, \mathsf{with}$

$\quad |\, \mathsf{Leaf} \;\Rightarrow\; [p = \mathsf{null}]$

$\quad |\, \mathsf{Node}\, X\, L \;\Rightarrow\; \exists xc.\quad p \mapsto \{|\mathsf{item}{=}x;\ \mathsf{children}{=}c|\}$

$\qquad\qquad\qquad\qquad \star\ x \rightsquigarrow R\, X$

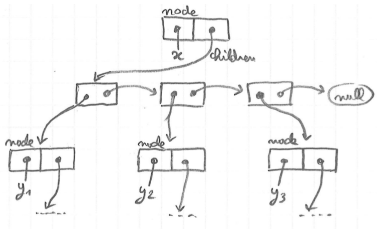
$\qquad\qquad\qquad\qquad \star\ c \rightsquigarrow \mathsf{Mlistof}\, (\mathsf{Narytreeof}\, R)\, L$

Trees with list of subtrees: representation of nodes

$$\begin{aligned} p \leadsto \mathsf{Nodeof}\, R_1\, V_1\, R_2\, V_2 \quad \equiv & \\ \exists v_1 v_2. \quad & p \mapsto \{\!|\mathsf{item}{=}v_1;\ \mathsf{children}{=}v_2|\!\} \\ & \star\, v_1 \leadsto R_1\, V_1 \\ & \star\, v_2 \leadsto R_2\, V_2 \end{aligned}$$

$$\begin{aligned} & p \leadsto \mathsf{Narytreeof}\, R\, T \equiv \\ & \quad \mathsf{match}\, T\, \mathsf{with} \\ & \quad |\, \mathsf{Leaf} \Rightarrow [p = \mathsf{null}] \\ & \quad |\, \mathsf{Node}\, X\, L \Rightarrow \exists xc. \quad p \mapsto \{\!|\mathsf{item}{=}x;\ \mathsf{children}{=}c|\!\} \\ & \qquad\qquad\qquad\qquad\quad \star\, x \leadsto R\, X \\ & \qquad\qquad\qquad\qquad\quad \star\, c \leadsto \mathsf{Mlistof}\,(\mathsf{Narytreeof}\, R)\, L \end{aligned}$$

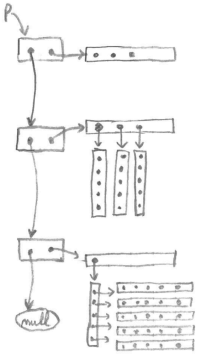
Trees with list of subtrees, revisited



Exercise: rewrite the specification of Narytreeof using Nodeof.

$$\begin{array}{l} p \rightsquigarrow \mathsf{Narytreeof}\, R\, T \equiv \\ \quad \mathsf{match}\, T\, \mathsf{with} \\ \quad \mid \mathsf{Leaf} \;\Rightarrow\; [p = \mathsf{null}] \\ \quad \mid \mathsf{Node}\, X\, L \;\Rightarrow\; p \rightsquigarrow \mathsf{Nodeof}\, R\, X\, (\mathsf{Mlistof}\, (\mathsf{Narytreeof}\, R))\, L \end{array}$$

Exercises



- Exam from 2015, Exercise 2: Bootstrapped chunked bags.

Available from the webpage of the course.

Chapter 23

Iteration with higher-order representation predicates

Iteration on lists

Recall:

$$\begin{array}{rl}\forall f l I. & \big(\forall x k.\ \{I\,k\}\ (f\ x)\ \{\lambda\_.\ I\,(k\&x)\}\big) \\ \Rightarrow & \{I\,\mathsf{nil}\}\ (\mathtt{iter}\ f\ l)\ \{\lambda\_.\ I\,l\}\end{array}$$

$$\begin{array}{rl}\forall f p l I. & \big(\forall x k.\ \{I\,k\}\ (f\ x)\ \{\lambda\_.\ I\,(k\&x)\}\big) \\ \Rightarrow & \{p \rightsquigarrow \mathsf{Mlist}\,l\ \star\ I\,\mathsf{nil}\}\ (\mathtt{miter}\ f\ p)\ \{\lambda\_.\ p \rightsquigarrow \mathsf{Mlist}\,l\ \star\ I\,l\}\end{array}$$

$$\begin{array}{rl}\forall f l J'. & \big(\forall x k k'.\ \{J'\,k\,k'\}\ (f\ x)\ \{\lambda x'.\,J\,(k\&x)\,(k'\&x')\}\big) \\ \Rightarrow & \{p \rightsquigarrow \mathsf{Mlist}\,l\ \star\ J'\,\mathsf{nil}\,\mathsf{nil}\}\ (\mathtt{mmap}\ f\ l)\ \{\lambda l'.\ p \rightsquigarrow \mathsf{Mlist}\,l\ \star\ J'\,l\,l'\}\end{array}$$

Challenge:

$$\begin{array}{rl} & \big(\forall x \ldots.\ \{\ldots\}\ (f\ x)\ \{\lambda\_.\ \ldots\}\big) \\ \Rightarrow & \{p \rightsquigarrow \mathsf{Mlistof}\,R\,L\ \star\ \ldots\}\ (\mathtt{miter}\ f\ p)\ \{\lambda\_.\ p \rightsquigarrow \ldots\ \star\ \ldots\}\end{array}$$

Iterating over a mutable list of mutable items

Exercise: specify the function `miter`, using an invariant of the form $J\,K\,K'$, describing the state before and the state after the iteration.

$$\begin{array}{ll} \forall f p R L J. & \Big(\forall x X K K'.\ \begin{array}{l}\{x \rightsquigarrow R\,X \,\star\, J\,K\,K'\} \\ (f\,x) \\ \{\lambda\_.\ \exists X'.\ x \rightsquigarrow R\,X' \,\star\, J\,(K \& X)\,(K' \& X')\}\end{array}\Big) \\ \Rightarrow & \{p \rightsquigarrow \mathsf{Mlistof}\,R\,L \,\star\, J\,\mathsf{nil}\,\mathsf{nil}\} \\ & (\texttt{miter}\,f\,p) \\ & \{\lambda\_.\ \exists L'.\ p \rightsquigarrow \mathsf{Mlistof}\,R\,L' \,\star\, J\,L\,L'\} \end{array}$$

Incrementing a mutable list of distinct references (1/2)

```
let incr_all p =
  miter (fun x -> incr x) p

let example_p =
  { hd = ref 5; tl = { hd = ref 3; tl = null } }
```

$$x \rightsquigarrow \mathsf{Ref}\, X \;\equiv\; x \mapsto X$$

Exercise: using the representation predicates Ref and Mlistof, specify the function `(fun x -> incr x)` and `incr_all`.

$$\{x \rightsquigarrow \mathsf{Ref}\, X\}\ (\texttt{incr x})\ \{\lambda\_.\ x \rightsquigarrow \mathsf{Ref}\,(X+1)\}$$

$$\{p \rightsquigarrow \mathsf{Mlistof}\,\mathsf{Ref}\,L\}\ (\texttt{incr\_all}\,p)\ \{\lambda\_.\ p \rightsquigarrow \mathsf{Mlistof}\,\mathsf{Ref}\,(\mathsf{map}\,(+1)\,L)\}$$

Incrementing a mutable list of distinct references (2/2)

$$\begin{array}{lll}\forall f p R L J. & \Big(\forall x X K K'. & \{x \leadsto R\, X \star J\, K\, K'\} \\ & & (f\, x) \\ & & \{\lambda\_.\ \exists X'.\ x \leadsto R\, X' \star J\, (K \& X)\, (K' \& X')\}\Big) \\ \Rightarrow & \{p \leadsto \mathsf{Mlistof}\, R\, L \star J\, \mathsf{nil}\, \mathsf{nil}\} & \\ & (\texttt{miter}\, f\, p) & \\ & \{\lambda\_.\ \exists L'.\ p \leadsto \mathsf{Mlistof}\, R\, L' \star J\, L\, L'\} & \end{array}$$

Consider:

$$J\, K\, K' \;\equiv\; [K' = \mathsf{map}\,(+1)\,K]$$

Derives:

$$\big(\forall x X.\ \{x \leadsto \mathsf{Ref}\, X\}\ (\texttt{fun x -> incr x})\ \{\lambda\_.\ x \leadsto \mathsf{Ref}\,(X+1)\}\big) \Rightarrow$$
$$\{p \leadsto \mathsf{Mlistof}\,\mathsf{Ref}\, L\}\ (\texttt{incr\_all}\, p)\ \{\lambda\_.\ p \leadsto \mathsf{Mlistof}\,\mathsf{Ref}\,(\mathsf{map}\,(+1)\, L)\}$$

Chapter 24

Resource analysis in Separation Logic

Controlling deallocation

(1) Remove the garbage collection rule:

$$\frac{\{H\}\ t\ \{Q \star \mathsf{GC}\}}{\{H\}\ t\ \{Q\}}\ \textsc{gc-post}$$

(2) Add a "free" function for explicit deallocation:

$$\{r \mapsto v\}\ (\texttt{free r})\ \{\lambda\_.\ [\,]\}$$

(3) Theorem: for a full program execution starting in the empty heap, all the data still allocated at the end is described in the post-condition.

(4) Corollary: terminating on the empty heap ensures no memory leaks.

$$\{[\,]\}\ t\ \{\lambda n.\ [P\,n]\}$$

File handle protocols

Goal: ensure that if a file is open then it is eventually closed.

$$f \rightsquigarrow \mathsf{File}\, L$$

where $(f : \mathsf{loc})$ denotes the file handler,
and $(L : \mathsf{list\,char})$ denotes the remaining bytes to read.

$$\begin{aligned}
\{[\,]\}\ \ (\texttt{fopen s})\ \ \{\lambda f.\ \exists L.\ f \rightsquigarrow \mathsf{File}\, L\} \\
\{f \rightsquigarrow \mathsf{File}\,(c :: L)\}\ \ (\texttt{fread f})\ \ \{\lambda x.\ [x = c] \star f \rightsquigarrow \mathsf{File}\, L\} \\
\{f \rightsquigarrow \mathsf{File}\, L\}\ (\texttt{fclose f})\ \{\lambda\_.\ [\,]\}
\end{aligned}$$

Complexity analysis

Time credits:

$$\$\,x \;:\; \text{Hprop} \qquad \text{where } x \in \mathbb{R}^+$$

Properties:

$$\$(x+y) \;=\; \$\,x \;\star\; \$\,y \quad \text{and} \quad \$\,0 \;=\; [\,]$$

Principle:

The execution of every instruction costs $\$1$.

Simplification:

Entering the body of a function or a loop costs $\$1$.

Time credits in pre-conditions

Constant-time:

$$\{t \leadsto \mathsf{Array}\, M \star \$\, c\}\ (\texttt{Array.length t})\ \{\lambda n.\, [n = |M|] \star t \leadsto \mathsf{Array}\, M\}$$

Linear-time:

$$\{\$(c_1 n + c_2)\}\ (\texttt{Array.make n v})\ \{\lambda t.\ \exists L.\ t \leadsto \mathsf{Array}\, L \star [\ldots]\}$$

Superlinear-time:

$$\{t \leadsto \mathsf{Array}\, L \star \$(c_1 |L| \log |L| + c_2)\}$$
$$(\texttt{Array.sort t})$$
$$\{\lambda t.\ \exists L'.\ t \leadsto \mathsf{Array}\, L' \star [\ldots]\}$$

Amortized analysis

Stack of unbounded size with amortized constant-time operations:

$$\{\$\, c\}\ (\texttt{Stack.create()})\ \{\lambda\_.\, s \rightsquigarrow \mathsf{Stack\,nil}\}$$

$$\{s \rightsquigarrow \mathsf{Stack}\, L \star \$\, c\}\ (\texttt{Stack.push s x})\ \{\lambda\_.\, s \rightsquigarrow \mathsf{Stack}\,(x :: L)\}$$

$$\{s \rightsquigarrow \mathsf{Stack}\,(x :: L) \star \$\, c\}\ (\texttt{Stack.pop s})\ \{\lambda y.\ [y = x] \star s \rightsquigarrow \mathsf{Stack}\, L\}$$

Representation predicate with a potential function:

$$\begin{aligned} s \rightsquigarrow \mathsf{Stack}\, L \quad \equiv \quad \exists ntMk. \quad & s \mapsto \{\!|\mathsf{size}{=}n;\ \mathsf{data}{=}t|\!\} \\ & \star\ t \rightsquigarrow \mathsf{Array}\, M \\ & \star\ [n = |L| \leqslant |M| = 2^k] \\ & \star\ [\forall i \in [0,n).\ M[i] = L[i]] \\ & \star\ \$(c' \cdot \mathsf{abs}(n - |M|/2)) \end{aligned}$$

Chapter 25

Read-only permissions

Motivation for read-only permissions

What we currently need to write:

$$\{a_1 \rightsquigarrow \mathsf{Array}\, L_1 \,\star\, a_2 \rightsquigarrow \mathsf{Array}\, L_2\}$$
$$(\texttt{concat}\, a_1\, a_2)$$
$$\{\lambda a_3.\ a_3 \rightsquigarrow \mathsf{Array}\,(L_1 +\!\!+ L_2) \,\star\, a_1 \rightsquigarrow \mathsf{Array}\, L_1 \,\star\, a_2 \rightsquigarrow \mathsf{Array}\, L_2\}$$

What we wish to write:

$$\{a_1 \overset{\mathsf{ro}}{\rightsquigarrow} \mathsf{Array}\, L_1 \,\star\, a_2 \overset{\mathsf{ro}}{\rightsquigarrow} \mathsf{Array}\, L_2\}$$
$$(\texttt{concat}\, a_1\, a_2)$$
$$\{\lambda a_3.\ a_3 \rightsquigarrow \mathsf{Array}\,(L_1 +\!\!+ L_2)\}$$

More than syntactic sugar:
– we wish "ro" to enforce no write operations,
– we wish to allow aliasing of read-only arguments.

Fractional permissions

$$(r \overset{\alpha}{\mapsto} v) \qquad \text{with} \;\; 0 < \alpha \leqslant 1$$

Splitting and merging:

$$(r \mapsto v) \;=\; (r \overset{1}{\mapsto} v) \;=\; (r \overset{1/2}{\mapsto} v) \star (r \overset{1/2}{\mapsto} v)$$

More generally:

$$(r \overset{\alpha+\beta}{\mapsto} v) \;=\; (r \overset{\alpha}{\mapsto} v) \star (r \overset{\beta}{\mapsto} v) \qquad \text{with} \;\; 0 < \alpha, \beta \leqslant 1$$

Operations:

$$\{[\,]\} \;\; (\texttt{ref v}) \;\; \{\lambda r.\; r \overset{1}{\mapsto} v\}$$

$$\{r \overset{1}{\mapsto} v'\} \; (\texttt{r := v}) \; \{\lambda \_ .\; r \overset{1}{\mapsto} v\}$$

$$\forall \alpha. \qquad \{r \overset{\alpha}{\mapsto} v\} \quad (\texttt{!r}) \quad \{\lambda x.\; [x = v] \star (r \overset{\alpha}{\mapsto} v)\}$$

Fractional permissions in practice

$$\begin{aligned}\forall \alpha\beta.\ & \{a_1 \overset{\alpha}{\rightsquigarrow} \mathsf{Array}\, L_1 \star a_2 \overset{\beta}{\rightsquigarrow} \mathsf{Array}\, L_2\} \\ & (\mathtt{concat}\, a_1\, a_2) \\ & \{\lambda a_3.\ a_1 \overset{\alpha}{\rightsquigarrow} \mathsf{Array}\, L_1 \star a_2 \overset{\beta}{\rightsquigarrow} \mathsf{Array}\, L_2 \star a_3 \overset{1}{\rightsquigarrow} \mathsf{Array}\, (L_1 +\!\!+ L_2)\}\end{aligned}$$

Limitations:

– need to quantify fractions explicitly,

– need to syntactic sugar to avoid copy-pasting,

– need to re-establish post-conditions,

– a fraction $\frac{1}{2}H$ cannot be defined for arbitrary H.

Generic read-only modifier

Extension of the logic with a modifier $\mathsf{RO}(H)$ that applies to any H.

$$a \overset{\mathsf{ro}}{\rightsquigarrow} \mathsf{Array}\, L \;\equiv\; \mathsf{RO}(a \rightsquigarrow \mathsf{Array}\, L)$$

$\mathsf{RO}(H)$ is duplicatable and never mentioned in post-conditions.

$$\frac{}{\mathsf{RO}(H) \;\;\rhd\;\; \mathsf{RO}(H) \,\star\, \mathsf{RO}(H)}\ \textsc{dup-ro}$$

$$\frac{}{\{\mathsf{RO}(l \mapsto v)\}\ (\mathsf{get}\, l)\ \{\lambda x.\ [x = v]\}}\ \textsc{get-ro}$$

Read-only frame rule

$\mathsf{RO}(H)$ is introduced on frame:

$$\frac{\{H \star \mathsf{RO}(H')\}\ t\ \{Q\} \qquad \text{no-ro-in}\ H'}{\{H \star H'\}\ t\ \{Q \star H'\}}\ \textsc{frame-ro}$$

Read-only sequencing rule

$$\frac{\{H\}\ t_1\ \{Q'\} \qquad \{Q'()\}\ t_2\ \{Q\}}{\{H\}\ (t_1\ ;\ t_2)\ \{Q\}}\ \textsc{seq}$$

$$\frac{\{H \star \mathsf{RO}(H')\}\ t_1\ \{Q'\} \qquad \{Q'() \star \mathsf{RO}(H')\}\ t_2\ \{Q\}}{\{H \star \mathsf{RO}(H')\}\ (t_1\ ;\ t_2)\ \{Q\}}\ \textsc{seq-ro}$$

$$\frac{\{H\}\ t_1\ \{Q'\} \qquad \{Q'() \star H'\}\ t_2\ \{Q\}}{\{H \star H'\}\ (t_1\ ;\ t_2)\ \{Q\}}\ \textsc{seq-frame}$$

RO in practice

$$\{\mathsf{RO}(a_1 \rightsquigarrow \mathsf{Array}\, L_1) \star \mathsf{RO}(a_2 \rightsquigarrow \mathsf{Array}\, L_2)\}$$
$$(\texttt{concat}\, a_1\, a_2)$$
$$\{\lambda a_3.\; a_3 \rightsquigarrow \mathsf{Array}\, (L_1 \mathbin{+\!\!+} L_2)\}$$

Chapter 26

Parallelism and Concurrency

Parallel pairs

A parallel pair, written $(|t_1, t_2|)$, for evaluating two subterms in parallel.

Computing: $a[i] + a[i+1] + ... + a[j-1]$.

```
let rec sum a i j =
  if j - i = 1 then a.(i) else begin
    let m = (i+j) / 2 in
    let (s1,s2) = (| sum a i m, sum a m j |) in
    s1 + s2
 end
```

Efficient use of parallel pairs with granularity control

```
let rec sum a i j =
  if j - i < sequential_cutoff then begin
    let r = ref 0 in
    for k = i to j-1 do
      r := !r + a.(k)
    done;
    !r
  end else begin
    let m = (i+j) / 2 in
    let (s1,s2) = (| sum a i m, sum a m j |) in
    s1 + s2
  end
```

Generalizable to map-reduce: $f(t[0]) \oplus f(a[1]) \oplus ... \oplus f(a[n-1])$.

Reasoning rule for parallel pairs

$$\frac{\{H_1\}\ t_1\ \{Q_1\} \qquad \{H_2\}\ t_2\ \{Q_2\}}{\{H_1 \star H_2\}\ (|t_1, t_2|)\ \{Q_1 \star Q_2\}}\ \textsc{parallel}$$

where $Q_1 \star Q_2 \ \equiv\ \lambda(x_1, x_2).\ Q_1\, x_1 \star Q_2\, x_2$

This rule restricts parallel threads to act on disjoint parts of memory.

Parallel rule needs read-only permissions

$$\frac{\{H_1\}\ t_1\ \{Q_1\} \qquad \{H_2\}\ t_2\ \{Q_2\}}{\{H_1 \star H_2\}\ (|t_1, t_2|)\ \{Q_1 \star Q_2\}}\ \textsc{parallel}$$

Compute: $u[a[0]] + u[a[1]] + ... + u[a[n-1]]$.

```
map_reduce (fun x -> u.(x)) 0 (+) 0 n
```

The ownership of the array `u` is needed in both branches.

$$\frac{\{H_1 \star \mathsf{RO}(H_3)\}\ t_1\ \{Q_1\} \qquad \{H_2 \star \mathsf{RO}(H_3)\}\ t_2\ \{Q_2\}}{\{H_1 \star H_2 \star \mathsf{RO}(H_3)\}\ (|t_1, t_2|)\ \{Q_1 \star Q_2\}}\ \textsc{parallel-ro}$$

Concurrent locks: example

```
let r = ref 0
let s = ref n
let p = create_lock()

let concurrent_step () =
  let () = acquire_lock p in
  incr r;
  decr s;
  release_lock p
```

Heap predicate $p \leadsto \mathsf{Lock}\, H$ asserts that lock p protects an invariant H.
Here:

$$p \leadsto \mathsf{Lock}\,(\exists i.\ (r \mapsto i) \star (s \mapsto n - i))$$

Concurrent locks: specification of operations

Duplicatable representation predicate:

$$p \overset{\mathsf{ro}}{\leadsto} \mathsf{Lock}\, H$$

Operations:

$$\forall H. \quad \{H\}\ (\texttt{create\_lock ()})\ \{\lambda p.\ p \overset{\mathsf{ro}}{\leadsto} \mathsf{Lock}\, H\}$$

$$\forall pH. \quad \{p \overset{\mathsf{ro}}{\leadsto} \mathsf{Lock}\, H\}\ (\texttt{acquire\_lock p})\ \{\lambda\_.\ H\}$$

$$\forall pH. \ \{H \star p \overset{\mathsf{ro}}{\leadsto} \mathsf{Lock}\, H\}\ (\texttt{release\_lock p})\ \{\lambda\_.\ [\,]\}$$

Concurrent locks: exercise

Describe the state at the front of each lines (except 5 and 6).
Explicit the instantiation of the existential in the invariant.

```
1    let r = ref 0
2    let s = ref n
3    let p = create_lock()
4
5    let concurrent_step () =
6       let () = acquire_lock p in
7       incr r;
8       decr s;
9       release_lock p
```

1: $[]$. 2: $r \mapsto 0$. 3: $r \mapsto 0 \star s \mapsto n$.
4: $p \overset{\text{ro}}{\rightsquigarrow} \mathsf{Lock}\,(\exists i.\ (r \mapsto i) \star (s \mapsto n - i))$.
7: $(r \mapsto i) \star (s \mapsto n - i)$. 8: $(r \mapsto i + 1) \star (s \mapsto n - i)$.
9: $(r \mapsto i + 1) \star (s \mapsto n - i - 1)$. Instantiate the invariant with $i + 1$.

Conclusion

Program verification using Separation Logic gives you:

- Expressiveness: tree-shaped structures, and structures with sharing
- Expressiveness: effectful, first-class functions, with local state
- Modularity: most-general specifications
- Modularity: composable representation predicates
- Abstraction: existential quantification of intermediate pointers
- Abstraction: existential quantification of invariants
- Practice: formalization in Coq of all heap predicates
- Practice: characteristic formulae for reasoning rules