

Separation Logic

2/4

Jean-Marie Madiot

Inria Paris

January 31, 2018

Chapter 7

The Frame Rule

Preservation of independent state

We have:

$$\{r \mapsto 2\} (\text{incr } r) \{\lambda_. r \mapsto 3\}$$

We also have:

$$\{r \mapsto 2 \star s \mapsto 7\} (\text{incr } r) \{\lambda_. r \mapsto 3 \star s \mapsto 7\}$$

Preservation of independent state

We have:

$$\{r \mapsto 2\} (\text{incr } r) \{\lambda_. r \mapsto 3\}$$

We also have:

$$\{r \mapsto 2 \star s \mapsto 7\} (\text{incr } r) \{\lambda_. r \mapsto 3 \star s \mapsto 7\}$$

More generally:

$$\{r \mapsto 2 \star H\} (\text{incr } r) \{\lambda_. r \mapsto 3 \star H\}$$

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:

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

By the frame rule, we have:

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

Frame rule and allocation

We have:

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

By the frame rule, we have:

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

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:

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

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:

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

We prove this specification by induction on L .

Verification of mlength: nil case

Case $L = \mathbf{nil}$. Then $p = \mathbf{null}$. Goal is:

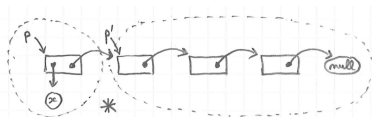
$$\{p \rightsquigarrow \mathbf{Mlist\ nil}\} (0) \{\lambda n. [n = \mathbf{length\ nil}] \star p \rightsquigarrow \mathbf{Mlist\ nil}\}$$

Same as:

$$\{[p = \mathbf{null}]\} (0) \{\lambda n. [n = 0] \star [p = \mathbf{null}]\}$$

Verification of mlength: frame process

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



Assume $L = x :: L'$.

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

Instantiation of the frame rule

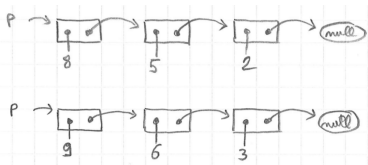
Induction hypothesis:

$$\begin{aligned} & \{p' \rightsquigarrow \text{Mlist } L'\} \\ & (\text{mlength } p') \\ & \{\lambda n'. [n = \text{length } L'] \star p' \rightsquigarrow \text{Mlist } L'\} \end{aligned}$$

By the frame rule:

$$\begin{aligned} & \{p' \rightsquigarrow \text{Mlist } L' \star p \rightsquigarrow \{\text{hd}=x; \text{tl}=p'\}\} \\ & (\text{mlength } p') \\ & \{\lambda n. [n = \text{length } L'] \star p' \rightsquigarrow \text{Mlist } L' \star p \rightsquigarrow \{\text{hd}=x; \text{tl}=p'\}\} \end{aligned}$$

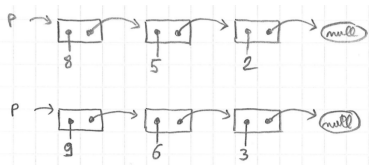
Verification of in-place increment



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

$$\forall pL. \{p \rightsquigarrow \text{Mlist } L\} (\text{list_incr } p) \{\lambda_. p \rightsquigarrow \text{Mlist}(\text{map}(+1) L)\}$$

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
```

$$\forall pL. \{p \rightsquigarrow \text{Mlist } L\} (\text{list_incr } p) \{\lambda_. p \rightsquigarrow \text{Mlist}(\text{map}(+1) L)\}$$

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

Verification of in-place increment: frame process

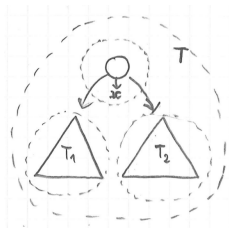
$\forall pL. \{p \rightsquigarrow \text{Mlist } L\} (\text{list_incr } p) \{\lambda_. p \rightsquigarrow \text{Mlist } (\text{map } (+1) L)\}$

Assume $L = x :: L'$.

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

Specification of tree copy

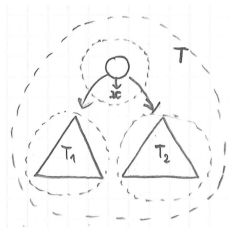
```
let rec copy (p:node) : node =  
  if p == null then null else  
  let p1' = copy p.left in  
  let p2' = copy p.right in  
  { item = p.item;  
    left = p1';  
    right = p2' }
```



Exercise: specify the tree copy function.

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' }
```

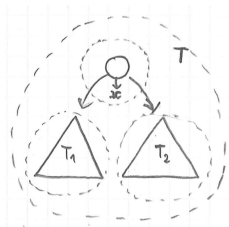


Exercise: specify the tree copy function.

$$\forall pT. \{p \rightsquigarrow \text{Mtree } T\} (\text{copy } p) \{ \lambda p'. p \rightsquigarrow \text{Mtree } T \star p' \rightsquigarrow \text{Mtree } T \}$$

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.

$$\forall pT. \{p \rightsquigarrow \text{Mtree } T\} (\text{copy } p) \{\lambda p'. p \rightsquigarrow \text{Mtree } T \star p' \rightsquigarrow \text{Mtree } T\}$$

Exercise: describe the frame process for tree copy.

Verification of tree copy: frame process

| | |
|---|------------------|
| $p \rightsquigarrow \text{Mtree } T$ | by pre-condition |
| $p \mapsto \{x; p_1; p_2\} \star p_1 \rightsquigarrow \text{Mtree } T_1 \star p_2 \rightsquigarrow \text{Mtree } T_2$ | by unfolding |
| $p_1 \rightsquigarrow \text{Mtree } T_1$ | frame begins |
| $p_1 \rightsquigarrow \text{Mtree } T_1$ $\star p'_1 \rightsquigarrow \text{Mtree } T_1$ | by induction |
| $p \mapsto \{x; p_1; p_2\} \star p_1 \rightsquigarrow \text{Mtree } T_1 \star p_2 \rightsquigarrow \text{Mtree } T_2$ $\star p'_1 \rightsquigarrow \text{Mtree } T_1$ | frame ends |
| $p_2 \rightsquigarrow \text{Mtree } T_2$ | frame begins |
| $p_2 \rightsquigarrow \text{Mtree } T_2$ $\star p'_2 \rightsquigarrow \text{Mtree } T_2$ | by induction |
| $p \mapsto \{x; p_1; p_2\} \star p_1 \rightsquigarrow \text{Mtree } T_1 \star p_2 \rightsquigarrow \text{Mtree } T_2$ $\star p'_1 \rightsquigarrow \text{Mtree } T_1 \star p'_2 \rightsquigarrow \text{Mtree } T_2$ | frame ends |
| $p \mapsto \{x; p_1; p_2\} \star p_1 \rightsquigarrow \text{Mtree } T_1 \star p_2 \rightsquigarrow \text{Mtree } T_2$ $\star p' \mapsto \{x; p'_1; p'_2\} \star p'_1 \rightsquigarrow \text{Mtree } T_1 \star p'_2 \rightsquigarrow \text{Mtree } T_2$ | by allocation |
| $p \rightsquigarrow \text{Mtree } T$ $\star p' \rightsquigarrow \text{Mtree } T$ | 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 $L = x :: L'$.

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

Chapter 8

Small footprint specifications

Small footprint access to records

$$p \rightsquigarrow \{\text{hd}=v; \text{tl}=q\} \equiv p.\text{hd} \mapsto v \star p.\text{tl} \mapsto q$$

Specification of a write on the head field:

$$\{p \rightsquigarrow \{\text{hd}=w; \text{tl}=q\}\} (p.\text{hd} \leftarrow v) \{\lambda_. p \rightsquigarrow \{\text{hd}=v; \text{tl}=q\}\}$$

Small footprint access to records

$$p \rightsquigarrow \{\text{hd}=v; \text{tl}=q\} \equiv p.\text{hd} \mapsto v \star p.\text{tl} \mapsto q$$

Specification of a write on the head field:

$$\{p \rightsquigarrow \{\text{hd}=w; \text{tl}=q\}\} (p.\text{hd} \leftarrow v) \{\lambda_. p \rightsquigarrow \{\text{hd}=v; \text{tl}=q\}\}$$

Same, but with a smaller footprint:

$$\{p.\text{hd} \mapsto w\} (p.\text{hd} \leftarrow v) \{\lambda_. p.\text{hd} \mapsto v\}$$

or $\{p.\text{hd} \mapsto -\} (p.\text{hd} \leftarrow v) \{\lambda_. p.\text{hd} \mapsto v\}$

Small footprint access to records

$$p \rightsquigarrow \{\text{hd}=v; \text{tl}=q\} \equiv p.\text{hd} \mapsto v \star p.\text{tl} \mapsto q$$

Specification of a write on the head field:

$$\{p \rightsquigarrow \{\text{hd}=w; \text{tl}=q\}\} (p.\text{hd} \leftarrow v) \{\lambda_. p \rightsquigarrow \{\text{hd}=v; \text{tl}=q\}\}$$

Same, but with a smaller footprint:

$$\begin{aligned} & \{p.\text{hd} \mapsto w\} (p.\text{hd} \leftarrow v) \{\lambda_. p.\text{hd} \mapsto v\} \\ \text{or } & \{p.\text{hd} \mapsto -\} (p.\text{hd} \leftarrow v) \{\lambda_. p.\text{hd} \mapsto v\} \end{aligned}$$

From small to large footprint using frame:

$$\{p.\text{hd} \mapsto w \star p.\text{tl} \mapsto q\} (p.\text{hd} \leftarrow v) \{\lambda_. p.\text{hd} \mapsto v \star p.\text{tl} \mapsto q\}$$

Representation predicate for arrays

Representation predicate for C arrays:

$$p \rightsquigarrow \text{Array } L \quad \equiv \quad \bigotimes_{v \text{ at index } i \text{ in } L} p[i] \mapsto v$$

where:

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

Representation predicate for arrays

Representation predicate for C arrays:

$$p \rightsquigarrow \text{Array } L \quad \equiv \quad \bigotimes_{v \text{ at index } i \text{ in } L} p[i] \mapsto v$$

where:

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

Representation predicate for ML arrays:

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

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

Small footprint specifications of array operations

$i \in \text{dom } L \Rightarrow$

$\{p \rightsquigarrow \text{Array } L\} (\text{Array.get } i \ p) \{\lambda x. [x = L[i]] \star p \rightsquigarrow \text{Array } L\}$

$\{p \rightsquigarrow \text{Array } L\} (\text{Array.set } i \ p \ v) \{\lambda_. p \rightsquigarrow \text{Array } (L[i := v])\}$

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

Exercise: give small footprint specifications for array operations.

How to derive the large footprint specifications from them?

Small footprint specifications of array operations

$i \in \text{dom } L \Rightarrow$

$\{p \rightsquigarrow \text{Array } L\} (\text{Array.get } i \text{ } p) \{\lambda x. [x = L[i]] \star p \rightsquigarrow \text{Array } L\}$

$\{p \rightsquigarrow \text{Array } L\} (\text{Array.set } i \text{ } p \text{ } v) \{\lambda_. p \rightsquigarrow \text{Array } (L[i := v])\}$

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

Exercise: give small footprint specifications for array operations.

How to derive the large footprint specifications from them?

$\{p[i] \mapsto v\} (\text{Array.get } i \text{ } p) \{\lambda x. [x = v] \star p[i] \mapsto v\}$

$\{p[i] \mapsto -\} (\text{Array.set } i \text{ } p \text{ } v) \{\lambda_. p[i] \mapsto v\}$

$\{p.\text{length} \mapsto n\} (\text{Array.length } p) \{\lambda x. [x = n] \star p.\text{length} \mapsto n\}$

Small footprint specifications of array operations

$i \in \text{dom } L \Rightarrow$

$\{p \rightsquigarrow \text{Array } L\} (\text{Array.get } i \ p) \{\lambda x. [x = L[i]] \star p \rightsquigarrow \text{Array } L\}$

$\{p \rightsquigarrow \text{Array } L\} (\text{Array.set } i \ p \ v) \{\lambda_. p \rightsquigarrow \text{Array } (L[i := v])\}$

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

Exercise: give small footprint specifications for array operations.

How to derive the large footprint specifications from them?

$\{p[i] \mapsto v\} (\text{Array.get } i \ p) \{\lambda x. [x = v] \star p[i] \mapsto v\}$

$\{p[i] \mapsto -\} (\text{Array.set } i \ p \ v) \{\lambda_. p[i] \mapsto v\}$

$\{p.\text{length} \mapsto n\} (\text{Array.length } p) \{\lambda x. [x = n] \star p.\text{length} \mapsto n\}$

$p \rightsquigarrow \text{Array } L = p[i] \mapsto L[i]$

$\star p.\text{length} \mapsto |L|$

$\star \textcircled{*} v \text{ at index } j \text{ in } L \quad p[j] \mapsto v$
with $j \neq i$

Dynamic access of ML arrays

Dynamic checks in ocaml:

```
# let v = Array.make 5 0 in Array.get v 7;;
```

Dynamic access of ML arrays

Dynamic checks in ocaml:

```
# let v = Array.make 5 0 in Array.get v 7;;
```

```
Exception: Invalid_argument "index out of bounds".
```

This means preconditions must retain some header information.

Dynamic access of ML arrays

Dynamic checks in ocaml:

```
# let v = Array.make 5 0 in Array.get v 7;;
```

```
Exception: Invalid_argument "index out of bounds".
```

This means preconditions must retain some header information.

When running programs proved in separation logic, one can disable those dynamic checks `ocamlopt -unsafe` and get faster code.

(Same story with `null`.)

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`

Access to a memory cell

In C, record and array accesses are treated uniformly:

| | | |
|---------------------------|-------------|-----------------------------|
| <code>p->hd = v</code> | compiles to | <code>*(p+hd)=v</code> |
| <code>p[i] = v</code> | compiles to | <code>*(p+i)=v</code> |
| <code>i[p] = v</code> | compiles to | <code>*(i+p)=v (...)</code> |

Access to a memory cell

In C, record and array accesses are treated uniformly:

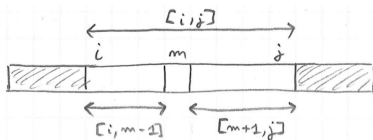
| | | |
|-------------------------------|-------------|------------------------------|
| $p \rightarrow \text{hd} = v$ | compiles to | $*(p + \text{hd}) = v$ |
| $p[i] = v$ | compiles to | $*(p + i) = v$ |
| $i[p] = v$ | compiles to | $*(i + p) = v \quad (\dots)$ |

Common small footprint specification for accessing a memory cell:

$$\{p \mapsto -\} (*p = v) \quad \{\lambda_. p \mapsto v\}$$
$$\{p \mapsto v\} (*p) \quad \{\lambda x. [x = v] \star p \mapsto v\}$$

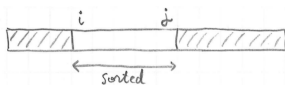
All other specifications for read and write operations are derivable.

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



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

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

Group of array cells

Definition

$$p \rightsquigarrow \text{Cells } M \quad \equiv \quad \bigotimes_{(i,v) \in M} p[i] \mapsto v$$

where M has type “map int A ”, for some A .

Group of array cells

Definition

$$p \rightsquigarrow \text{Cells } M \equiv \bigotimes_{(i,v) \in M} p[i] \mapsto v$$

where M has type “map int A ”, for some A .

Conversions:

$$p \rightsquigarrow \text{Array } L = p.\text{length} \mapsto |L| \star p \rightsquigarrow \text{Cells } (\text{mapOfList } L)$$

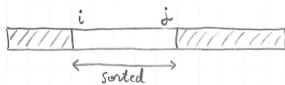
$$p \rightsquigarrow \text{Cells } \emptyset = []$$

$$p \rightsquigarrow \text{Cells } \{(i, v)\} = p[i] \mapsto v$$

$$p \rightsquigarrow \text{Cells } (M_1 \uplus M_2) = p \rightsquigarrow \text{Cells } M_1 \star p \rightsquigarrow \text{Cells } M_2 \quad (\text{when } M_1 \perp M_2)$$

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

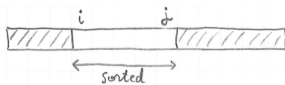
Small-footprint specification of quicksort



Exercise: give a small-footprint specification for quicksort.

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

Small-footprint specification of quicksort

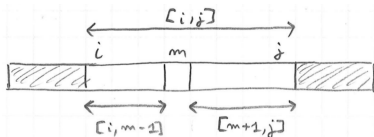


Exercise: give a small-footprint specification for quicksort.

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

$$\begin{aligned} \forall p M i j. \quad \text{dom } M = [i, j] \Rightarrow \\ \{ p \rightsquigarrow \text{Cells } M \} \\ (\text{qsort } i \ j \ p) \\ \{ \lambda _ . \exists M'. \quad p \rightsquigarrow \text{Cells } M' \quad \} \\ \star [\text{sortedSeg } i \ j \ M'] \\ \star [\text{permut } M \ M'] \end{aligned}$$

Segment splitting in Quicksort



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

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

Note that:

$$p \rightsquigarrow \text{Cells } (M|_{[m, m]}) = p[m] \mapsto M[m]$$

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:

$i \in \text{dom } M \Rightarrow$

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

$\{p \rightsquigarrow \text{Cells } M\} (\text{Array.set } i \ p \ v) \{\lambda_. p \rightsquigarrow \text{Cells } (M[i := v])\}$

Summary

Representation of a full array using a list:

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

Small footprint accesses:

$$\begin{aligned} \{p[i] \mapsto v\} & \quad (\text{Array.get } i \text{ } p) \quad \{\lambda_. p[i] \mapsto v\} \\ \{p[i] \mapsto -\} & \quad (\text{Array.set } i \text{ } p \text{ } v) \quad \{\lambda x. [x = v] \star p[i] \mapsto v\} \\ \{p.\text{length} \mapsto n\} & \quad (\text{Array.length } p) \quad \{\lambda x. [x = n] \star p.\text{length} \mapsto n\} \end{aligned}$$

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

$$p \rightsquigarrow \text{Cells } M \quad \equiv \quad \bigotimes_{(i,v) \in M} p[i] \mapsto v$$

Chapter 9

Heap entailment

Definition of Hprop

Let:

$$\text{Hprop} \equiv \text{Heap} \rightarrow \text{Prop}$$

For example:

$$[] : \text{Hprop}$$

$$l \mapsto v : \text{Hprop}$$

$$H_1 \star H_2 : \text{Hprop}$$

In particular:

$$(\star) : \text{Hprop} \rightarrow \text{Hprop} \rightarrow \text{Hprop}$$

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:

$$(\forall x. f x = g x) \Rightarrow (f = g) \quad (\text{if } f, g: A \rightarrow B)$$

Propositional extensionality:

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

Predicate extensionality:

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

Heap predicate extensionality:

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

Heap entailment

Definition:

$$H_1 \triangleright H_2 \quad \equiv \quad \forall m. H_1 m \Rightarrow H_2 m$$

For example:

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

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

Heap entailment as a partial order

$$H_1 \triangleright H_2 \quad \equiv \quad \forall m. H_1 m \Rightarrow H_2 m$$

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

REFLEXIVITY

$$\frac{}{H \triangleright H}$$

TRANSITIVITY

$$\frac{H_1 \triangleright H_2 \quad H_2 \triangleright H_3}{H_1 \triangleright H_3}$$

ANTISYMMETRY

$$\frac{H_1 \triangleright H_2 \quad H_2 \triangleright H_1}{H_1 = H_2}$$

Frame property for heap entailment

$$\frac{H_1 \triangleright H'_1}{H_1 \star H_2 \triangleright H'_1 \star H_2} \text{ENTAIL-FRAME}$$

For example, to prove:

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

it suffices and prove:

$$(s \mapsto 3) \triangleright (t \mapsto n).$$

Heap implications: true or false?

1. $(r \mapsto 3) \star (s \mapsto 4) \triangleright (s \mapsto 4) \star (r \mapsto 3)$
2. $(r \mapsto 3) \triangleright (s \mapsto 4) \star (r \mapsto 3)$
3. $(s \mapsto 4) \star (r \mapsto 3) \triangleright (r \mapsto 4)$
4. $(s \mapsto 4) \star (r \mapsto 3) \triangleright (r \mapsto 3)$
5. $[\text{False}] \star (r \mapsto 3) \triangleright (s \mapsto 4) \star (r \mapsto 4)$
6. $(r \mapsto 4) \star (s \mapsto 3) \triangleright [\text{False}]$
7. $(r \mapsto 4) \star (r \mapsto 3) \triangleright [\text{False}]$
8. $(r \mapsto 3) \star (r \mapsto 3) \triangleright [\text{False}]$

Heap implications: true or false?

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

Heap implications: true or false?

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

Instantiation of existentials and propositions

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

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

Rules:

$$\frac{H_1 \triangleright ([x \rightarrow v] H_2)}{H_1 \triangleright (\exists x. H_2)} \text{ EXISTS-R}$$

$$\frac{(H_1 \triangleright H_2) \quad P}{H_1 \triangleright (H_2 \star [P])} \text{ PROP-R}$$

Instantiation of existentials and propositions

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

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

Rules:

$$\frac{H_1 \triangleright ([x \rightarrow v] H_2)}{H_1 \triangleright (\exists x. H_2)} \text{ EXISTS-R} \qquad \frac{(H_1 \triangleright H_2) \quad P}{H_1 \triangleright (H_2 \star [P])} \text{ PROP-R}$$

Example:

$$\frac{\frac{\frac{\overline{(r \mapsto 6) \triangleright (r \mapsto 6)}}{\text{REFL}} \quad \frac{\overline{\text{even } 6}}{\text{MATH}}}{(r \mapsto 6) \triangleright (r \mapsto 6) \star [\text{even } 6]} \text{ PROP-R}}{(r \mapsto 6) \triangleright [n \rightarrow 6] ((r \mapsto n) \star [\text{even } n])} \text{ SUBST}}{(r \mapsto 6) \triangleright \exists n. (r \mapsto n) \star [\text{even } n]} \text{ EXISTS-R}$$

Extraction of existentials and propositions

$$(\exists n. [\text{even } n] \star (r \mapsto n)) \triangleright (\exists m. [\text{even } m] \star (r \mapsto m + 2))$$

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

$$(r \mapsto n) \triangleright \exists m. [\text{even } m] \star (r \mapsto m + 2)$$

Extraction of existentials and propositions

$$(\exists n. [\text{even } n] \star (r \mapsto n)) \triangleright (\exists m. [\text{even } m] \star (r \mapsto m + 2))$$

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

$$(r \mapsto n) \triangleright \exists m. [\text{even } m] \star (r \mapsto m + 2)$$

Reasoning rules:

$$\frac{\forall x. (H_1 \triangleright H_2)}{(\exists x. H_1) \triangleright H_2} \text{ EXISTS-L}$$

$$\frac{P \Rightarrow (H_1 \triangleright H_2)}{([P] \star H_1) \triangleright H_2} \text{ PROP-L}$$

Extraction of existentials and propositions

$$(\exists n. [\text{even } n] \star (r \mapsto n)) \triangleright (\exists m. [\text{even } m] \star (r \mapsto m + 2))$$

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

$$(r \mapsto n) \triangleright \exists m. [\text{even } m] \star (r \mapsto m + 2)$$

Reasoning rules:

$$\frac{\forall x. (H_1 \triangleright H_2)}{(\exists x. H_1) \triangleright H_2} \text{ EXISTS-L}$$

$$\frac{P \Rightarrow (H_1 \triangleright H_2)}{([P] \star H_1) \triangleright H_2} \text{ PROP-L}$$

Same with explicit proof contexts:

$$\frac{\Gamma, x : A \vdash H_1 \triangleright H_2}{\Gamma \vdash (\exists(x : A). H_1) \triangleright H_2} (x \notin H_2)$$

$$\frac{\Gamma, h : P \vdash H_1 \triangleright H_2}{\Gamma \vdash ([P] \star H_1) \triangleright H_2}$$

Heap implications: true or false?

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

Heap implications: true or false?

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

$$\frac{}{a : \text{int}, a > 5 \vdash (r \mapsto 3) \star (s \mapsto a) \triangleright (r \mapsto 3) \star (s \mapsto a)}$$
$$\frac{}{a : \text{int}, a > 5 \vdash (r \mapsto 3) \star (s \mapsto a) \triangleright (r \mapsto 3) \star (s \mapsto 3 + (a - 3))}$$
$$\frac{}{a : \text{int}, a > 5 \vdash (r \mapsto 3) \star (s \mapsto a) \triangleright \exists m. (r \mapsto 3) \star (s \mapsto 3 + m)}$$
$$\frac{}{a : \text{int}, a > 5 \vdash (r \mapsto 3) \star (s \mapsto a) \triangleright \exists nm. (r \mapsto n) \star (s \mapsto n + m)}$$
$$\frac{}{\emptyset \vdash \exists a. [a > 5] \star (r \mapsto 3) \star (s \mapsto a) \triangleright \exists nm. (r \mapsto n) \star (s \mapsto n + m)}$$
$$\frac{}{\emptyset \vdash (r \mapsto 3) \star \exists a. [a > 5] \star (s \mapsto a) \triangleright \exists nm. (s \mapsto n + m) \star (r \mapsto n)}$$

Summary

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

Existentials and pure facts may be extruded from stars.

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

“ $[\text{False}] \triangleright H$ ” 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:

$$\frac{\{H_1\} t \{Q_1\}}{\{H_1 \star H_2\} t \{Q_1 \star H_2\}} \text{FRAME}$$

with the overloading:

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

Consequence rule

$$\frac{H \triangleright H' \quad \{H'\} t \{Q'\} \quad Q' \triangleright Q}{\{H\} t \{Q\}} \text{CONSEQUENCE}$$

with the overloading:

$$Q' \triangleright Q \equiv \forall x. (Q' x \triangleright Q x)$$

Consequence rule

$$\frac{H \triangleright H' \quad \{H'\} t \{Q'\} \quad Q' \triangleright Q}{\{H\} t \{Q\}} \text{ CONSEQUENCE}$$

with the overloading:

$$Q' \triangleright Q \equiv \forall x. (Q' x \triangleright Q x)$$

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:

$$\{\{\}\} (\text{myref } x) \{\lambda r. r \mapsto x \star \exists s. s \mapsto r\}$$

To:

$$\{\{\}\} (\text{myref } x) \{\lambda r. r \mapsto x\}$$

Recall the need for garbage collection

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

From:

$$\{\{\}\} (\text{myref } x) \{\lambda r. r \mapsto x \star \exists s. s \mapsto r\}$$

To:

$$\{\{\}\} (\text{myref } x) \{\lambda r. r \mapsto x\}$$

Can the following rule be used?

$$\frac{\{H\} t \{Q \star H'\}}{\{H\} t \{Q\}} \text{GC-POST'}$$

Garbage collection rule

$$\frac{\{H\} t \{Q \star \text{GC}\}}{\{H\} t \{Q\}} \text{GC-POST} \quad \text{where: } \text{GC} \equiv \exists H'. H'$$

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

$$\frac{\{H\} t \{Q\}}{\{H \star GC\} t \{Q\}} \text{GC-PRE} \qquad \frac{\{H\} t \{Q\}}{\{H \star H'\} t \{Q\}} \text{GC-PRE}'$$

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

$$\frac{\{H\} t \{Q \star GC\}}{\{H\} t \{Q\}} \text{GC-POST} \qquad \frac{\{H_1\} t \{Q_1\}}{\{H_1 \star H_2\} t \{Q_1 \star H_2\}} \text{FRAME}$$

Garbage collection in the pre-condition

$$\frac{\{H\} t \{Q\}}{\{H \star GC\} t \{Q\}} \text{GC-PRE} \qquad \frac{\{H\} t \{Q\}}{\{H \star H'\} t \{Q\}} \text{GC-PRE}'$$

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

$$\frac{\{H\} t \{Q \star GC\}}{\{H\} t \{Q\}} \text{GC-POST} \qquad \frac{\{H_1\} t \{Q_1\}}{\{H_1 \star H_2\} t \{Q_1 \star H_2\}} \text{FRAME}$$

Proof:

$$\frac{\frac{\{H\} t \{Q\}}{\{H \star GC\} t \{Q \star GC\}} \text{FRAME}}{\{H \star GC\} t \{Q\}} \text{GC-POST}$$

Combined structural rule

$$\frac{H \triangleright H' \quad \{H'\} t \{Q'\} \quad Q' \triangleright Q}{\{H\} t \{Q\}} \text{CONSEQUENCE}$$

$$\frac{\{H\} t \{Q \star GC\}}{\{H\} t \{Q\}} \text{GC-POST}$$

$$\frac{\{H_1\} t \{Q_1\}}{\{H_1 \star H_2\} t \{Q_1 \star H_2\}} \text{FRAME}$$

Combined structural rule

$$\frac{H \triangleright H' \quad \{H'\} t \{Q'\} \quad Q' \triangleright Q}{\{H\} t \{Q\}} \text{CONSEQUENCE}$$

$$\frac{\{H\} t \{Q \star GC\}}{\{H\} t \{Q\}} \text{GC-POST}$$

$$\frac{\{H_1\} t \{Q_1\}}{\{H_1 \star H_2\} t \{Q_1 \star H_2\}} \text{FRAME}$$

$$\frac{H = H_1 \star H_2 \quad \{H_1\} t \{Q_1\} \quad Q_1 \star H_2 = Q}{\{H\} t \{Q\}} \text{FRAME'}$$

Combined structural rule

$$\frac{H \triangleright H' \quad \{H'\} t \{Q'\} \quad Q' \triangleright Q}{\{H\} t \{Q\}} \text{ CONSEQUENCE}$$

$$\frac{\{H\} t \{Q \star \text{GC}\}}{\{H\} t \{Q\}} \text{ GC-POST} \qquad \frac{\{H_1\} t \{Q_1\}}{\{H_1 \star H_2\} t \{Q_1 \star H_2\}} \text{ FRAME}$$

$$\frac{H = H_1 \star H_2 \quad \{H_1\} t \{Q_1\} \quad Q_1 \star H_2 = Q}{\{H\} t \{Q\}} \text{ FRAME'}$$

$$\frac{H \triangleright H_1 \star H_2 \quad \{H_1\} t \{Q_1\} \quad Q_1 \star H_2 \triangleright Q \star \text{GC}}{\{H\} t \{Q\}} \text{ COMBINED}$$

Extraction of existentials and propositions

$$\{\exists n. (r \mapsto n) \star [\text{even } n]\} (!r) \{\lambda x. \dots\}$$

To prove the above, we need to show that:

$$\forall n. \text{even } n \Rightarrow \{r \mapsto n\} (!r) \{\lambda x. \dots\}$$

Rules:

$$\frac{\forall x. \{H\} t \{Q\}}{\{\exists x. H\} t \{Q\}} \text{ EXISTS}$$

$$\frac{P \Rightarrow \{H\} t \{Q\}}{\{[P] \star H\} t \{Q\}} \text{ PROP}$$

Application: copying a tree with invariants

Specification of copy for binary trees:

$$\{p \rightsquigarrow \text{Mtree } T\} (\text{copy } p) \{\lambda p'. p \rightsquigarrow \text{Mtree } T \star p' \rightsquigarrow \text{Mtree } T\}$$

Description of complete binary trees:

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

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

Application: copying a tree with invariants

Specification of copy for binary trees:

$$\{p \rightsquigarrow \text{Mtree } T\} (\text{copy } p) \{ \lambda p'. p \rightsquigarrow \text{Mtree } T \star p' \rightsquigarrow \text{Mtree } T \}$$

Description of complete binary trees:

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

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

$$\{p \rightsquigarrow \text{MtreeComplete } T\} (\text{copy } p) \{ \lambda p'. \begin{array}{l} p \rightsquigarrow \text{MtreeComplete } T \\ \star p' \rightsquigarrow \text{MtreeComplete } T \end{array} \}$$

Proof of the derived specification

(1) By unfolding of MtreeComplete:

$$\{\exists n. (p \rightsquigarrow \text{Mtree } T) \star [\text{depth } n T]\}$$

(copy p)

$$\{\lambda p'. \quad \exists n. (p \rightsquigarrow \text{Mtree } T) \star [\text{depth } n T] \quad \} \\ \star \exists n. (p' \rightsquigarrow \text{Mtree } T) \star [\text{depth } n T]$$

Proof of the derived specification

(1) By unfolding of MtreeComplete:

$$\begin{aligned} & \{ \exists n. (p \rightsquigarrow \text{Mtree } T) \star [\text{depth } n T] \} \\ & \text{(copy p)} \\ & \{ \lambda p'. \exists n. (p \rightsquigarrow \text{Mtree } T) \star [\text{depth } n T] \} \\ & \quad \star \exists n. (p' \rightsquigarrow \text{Mtree } T) \star [\text{depth } n T] \end{aligned}$$

(2) By the EXISTS and PROP rules:

$$\begin{aligned} \forall n. \text{depth } n T \Rightarrow \{ p \rightsquigarrow \text{Mtree } T \} \\ & \text{(copy p)} \\ & \{ \lambda p'. \exists n. (p \rightsquigarrow \text{Mtree } T) \star [\text{depth } n T] \} \\ & \quad \star \exists n. (p' \rightsquigarrow \text{Mtree } T) \star [\text{depth } n T] \end{aligned}$$

Proof of the derived specification

(1) By unfolding of MtreeComplete:

$$\begin{aligned} & \{ \exists n. (p \rightsquigarrow \text{Mtree } T) \star [\text{depth } n T] \} \\ & \text{(copy p)} \\ & \{ \lambda p'. \exists n. (p \rightsquigarrow \text{Mtree } T) \star [\text{depth } n T] \} \\ & \star \{ \exists n. (p' \rightsquigarrow \text{Mtree } T) \star [\text{depth } n T] \} \end{aligned}$$

(2) By the EXISTS and PROP rules:

$$\begin{aligned} \forall n. \text{depth } n T \Rightarrow \{ p \rightsquigarrow \text{Mtree } T \} \\ & \text{(copy p)} \\ & \{ \lambda p'. \exists n. (p \rightsquigarrow \text{Mtree } T) \star [\text{depth } n T] \} \\ & \star \{ \exists n. (p' \rightsquigarrow \text{Mtree } T) \star [\text{depth } n T] \} \end{aligned}$$

(3) By the CONSEQUENCE rule:

$$p \rightsquigarrow \text{Mtree } T \star p' \rightsquigarrow \text{Mtree } T \triangleright \begin{aligned} & \exists n. (p \rightsquigarrow \text{Mtree } T) \star [\text{depth } n T] \\ & \star \exists n. (p' \rightsquigarrow \text{Mtree } T) \star [\text{depth } n T] \end{aligned}$$

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

Summary

Structural rules:

$$\frac{H \triangleright H_1 \star H_2 \quad \{H_1\} t \{Q_1\} \quad Q_1 \star H_2 \triangleright Q \star GC}{\{H\} t \{Q\}} \text{ COMBINED}$$

$$\frac{\forall x. \{H\} t \{Q\}}{\{\exists x. H\} t \{Q\}} \text{ EXISTS} \qquad \frac{P \Rightarrow \{H\} t \{Q\}}{\{[P] \star H\} t \{Q\}} \text{ PROP}$$

Other structural rules are derivable.

Chapter 11

Reasoning rules for terms

Reasoning rule for sequences

Example:

$$\frac{\{r \mapsto n\} (\text{incr } r) \{\lambda_. r \mapsto n + 1\} \quad \{r \mapsto n + 1\} (!r) \{\lambda x. [x = n + 1] \star r \mapsto n + 1\}}{\{r \mapsto n\} (\text{incr } r; !r) \{\lambda x. [x = n + 1] \star r \mapsto n + 1\}}$$

Reasoning rule for sequences

Example:

$$\frac{\{r \mapsto n\} (\text{incr } r) \{\lambda_. r \mapsto n + 1\} \quad \{r \mapsto n + 1\} (!r) \{\lambda x. [x = n + 1] \star r \mapsto n + 1\}}{\{r \mapsto n\} (\text{incr } r; !r) \{\lambda x. [x = n + 1] \star r \mapsto n + 1\}}$$

Exercise: complete the rule for sequences.

$$\frac{\{\dots\} t_1 \{\dots\} \quad \{\dots\} t_2 \{\dots\}}{\{H\} (t_1; t_2) \{Q\}}$$

Reasoning rule for sequences

Solution 1:

$$\frac{\{H\} t_1 \{\lambda_{-}. H'\} \quad \{H'\} t_2 \{Q\}}{\{H\} (t_1 ; t_2) \{Q\}}$$

Solution 2:

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

Remark: $Q' = \lambda_{-}. H'$ is equivalent to $Q' () = H'$.

Reasoning rule for let-bindings

Exercise: complete the reasoning rule for let-bindings.

$$\frac{\{\dots\} t_1 \{\dots\} \quad \forall x. (\{\dots\} t_2 \{\dots\})}{\{H\} (\text{let } x = t_1 \text{ in } t_2) \{Q\}}$$

Reasoning rule for let-bindings

Exercise: complete the reasoning rule for let-bindings.

$$\frac{\{\dots\} t_1 \{\dots\} \quad \forall x. (\{\dots\} t_2 \{\dots\})}{\{H\} (\text{let } x = t_1 \text{ in } t_2) \{Q\}}$$

Solution:

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

Example of let-binding

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

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

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

Example of let-binding

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

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

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

Solution:

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

Reasoning rule for values

Example:

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

Rule:

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

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

$$\frac{\dots \triangleright \dots}{\{H\} v \{Q\}}$$

Reasoning rule for values

Example:

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

Rule:

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

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

$$\frac{\dots \triangleright \dots}{\{ H \} v \{ Q \}}$$

Solution:

$$\frac{H \triangleright Q v}{\{ H \} v \{ Q \}} \text{VAL-FRAME}$$

Derivability of the val-frame rule

$$\frac{H \triangleright Q v}{\{H\} v \{Q\}} \text{ VAL-FRAME}$$

Proof:

$$\frac{\frac{\frac{\overline{\{[]\} v \{\lambda x. [x = v]\}} \text{ VAL}}{\{H\} v \{\lambda x. [x = v] \star H\}} \text{ FRAME} \quad \frac{\frac{\frac{\overline{H \triangleright Q v} \text{ HYPOTHESIS}}{\forall x. x = v \Rightarrow (H \triangleright Q x)} \text{ SUBST}}{\forall x. ([x = v] \star H) \triangleright (Q x)} \text{ PROP-L}}{(\lambda x. [x = v] \star H) \triangleright Q} \text{ DEF OF } \triangleright}{\{H\} v \{Q\}} \text{ CONSEQ}$$

Reasoning rule for conditionals

Rule:

$$\frac{(v = \text{true} \Rightarrow \{H\} t_1 \{Q\}) \quad (v = \text{false} \Rightarrow \{H\} t_2 \{Q\})}{\{H\} (\text{if } v \text{ then } t_1 \text{ else } t_2) \{Q\}} \text{IF}$$

Reasoning rule for conditionals

Rule:

$$\frac{(v = \text{true} \Rightarrow \{H\} t_1 \{Q\}) \quad (v = \text{false} \Rightarrow \{H\} t_2 \{Q\})}{\{H\} (\text{if } v \text{ then } t_1 \text{ else } t_2) \{Q\}} \text{IF}$$

Transformation to A-normal form:

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

Reasoning rule for top-level functions

Rule:

$$\frac{v_1 = \lambda x. t \quad \{H\} ([x \rightarrow v_2] t) \{Q\}}{\{H\} (v_1 v_2) \{Q\}} \text{TOP-FUN}$$

Transformation to A-normal form:

$$(t_1 t_2) = (\text{let } f = t_1 \text{ in let } v = t_2 \text{ in } (f v))$$

Verification of a simple function

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

Specification:

$$\forall rn. \{r \mapsto n\} (\text{incr } r) \{\lambda_. r \mapsto n + 1\}$$

Verification:

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

$$\{r \mapsto n\} (\text{let } a = !r \text{ in } r := a+1) \{\lambda_. r \mapsto n + 1\}$$

Verification of a simple function

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

Specification:

$$\forall rn. \{r \mapsto n\} (\text{incr } r) \{\lambda_. r \mapsto n + 1\}$$

Verification:

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

$$\{r \mapsto n\} (\text{let } a = !r \text{ in } r := a+1) \{\lambda_. r \mapsto n + 1\}$$

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:

$$\frac{v_1 = \mu f. \lambda x. t \quad \{H\} ([f \rightarrow v_1] [x \rightarrow v_2] t) \{Q\}}{\{H\} (v_1 v_2) \{Q\}} \text{TOP-FIX}$$

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:

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

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:

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

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

Verification of mlength: nil case

Case $p = \text{null}$. Goal is:

$$\{p \rightsquigarrow \text{Mlist } L\} (0) \{\lambda n. [n = |L|] \star p \rightsquigarrow \text{Mlist } L\}$$

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

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

- Replace L with nil .

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

- Apply the VAL-FRAME rule.

$$[] \triangleright [0 = 0] \star [\text{nil} = \text{nil}]$$

Verification of mlength: cons case (1/2)

Case $p \neq \text{null}$. Goal is:

$$\begin{aligned} & \{p \rightsquigarrow \text{Mlist } L\} \\ & (\text{let } p' = p.\text{tl} \text{ in let } n' = \text{mlength } p' \text{ in } 1 + n') \\ & \{\lambda n. [n = |L|] \star p \rightsquigarrow \text{Mlist } L\} \end{aligned}$$

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

$$p' \rightsquigarrow \text{Mlist } L' \star p \rightsquigarrow \{\{\text{hd}=x; \text{tl}=p'\}\}$$

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

$$\begin{aligned} & \{p' \rightsquigarrow \text{Mlist } L' \star p \rightsquigarrow \{\{\text{hd}=x; \text{tl}=p'\}\}\} \\ & (\text{let } n' = \text{mlength } p' \text{ in } 1 + n') \\ & \{\lambda n. [n = |L|] \star p' \rightsquigarrow \text{Mlist } L' \star p \rightsquigarrow \{\{\text{hd}=x; \text{tl}=p'\}\}\} \end{aligned}$$

– Apply the frame rule to remove: $p \rightsquigarrow \{\{\text{hd}=x; \text{tl}=p'\}\}$.

– Apply the let-binding rule with : $Q \equiv \lambda n'. [n' = |L'|] \star p' \rightsquigarrow \text{Mlist } L'$.

Verification of mlength: cons case (2/2)

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

– First branch, exploit the induction hypothesis:

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

– Second branch:

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

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

$$n' = |L'| \quad \Rightarrow \quad p' \rightsquigarrow \text{Mlist } L' \triangleright [1 + n' = |L|] \star p' \rightsquigarrow \text{Mlist } L'$$

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

Reasoning rule for local functions

Rule template:

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

Hypothesis about f :

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

Rule:

$$\frac{P f = (\forall x H' Q'. \{H'\} t_1 \{Q'\} \Rightarrow \{H'\} (f x) \{Q'\}) \quad \forall f. P f \Rightarrow \{H\} t_2 \{Q\}}{\{H\} (\text{let rec } f x = t_1 \text{ in } t_2) \{Q\}} \text{ FIX}$$

Summary

$$\overline{\{[]\} v \{\lambda x. [x = v]\}}$$

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

$$\frac{v = \text{true} \Rightarrow \{H\} t_1 \{Q\} \quad v = \text{false} \Rightarrow \{H\} t_2 \{Q\}}{\{H\} (\text{if } v \text{ then } t_1 \text{ else } t_2) \{Q\}}$$

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

Summary of Course 2

Summary of chapter 7

$$\frac{\{H_1\} t \{\lambda x. H'_1\}}{\{H_1 \star H_2\} t \{\lambda x. H'_1 \star H_2\}} \text{FRAME}$$

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

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

Summary of chapter 8

Small footprint specification for C-style memory accesses:

$$\begin{aligned} & \{p \mapsto w\} (*p = v) \{ \lambda_. p \mapsto v \} \\ & \{p \mapsto v\} (*p) \quad \{ \lambda x. [x = v] \star p \mapsto v \} \end{aligned}$$

Representation of a full array using a list:

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

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

$$p \rightsquigarrow \text{Cells } M \quad \equiv \quad \bigotimes_{(i,v) \in M} p[i] \mapsto v$$

Summary of chapter 9

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

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

“ $[\text{False}] \triangleright H$ ” 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:

$$\frac{H \triangleright H_1 \star H_2 \quad \{H_1\} t \{Q_1\} \quad Q_1 \star H_2 \triangleright Q \star GC}{\{H\} t \{Q\}} \text{ COMBINED}$$

$$\frac{\forall x. \{H\} t \{Q\}}{\{\exists x. H\} t \{Q\}} \text{ EXISTS} \qquad \frac{P \Rightarrow \{H\} t \{Q\}}{\{[P] \star H\} t \{Q\}} \text{ PROP}$$

Other structural rules are derivable.

Summary of chapter 11

$$\overline{\{[]\} v \{\lambda x. [x = v]\}}$$

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

$$\frac{v = \text{true} \Rightarrow \{H\} t_1 \{Q\} \quad v = \text{false} \Rightarrow \{H\} t_2 \{Q\}}{\{H\} (\text{if } v \text{ then } t_1 \text{ else } t_2) \{Q\}}$$

$$\frac{\forall f. Pf \Rightarrow \{H\} t_2 \{Q\} \quad Pf = (\forall x H' Q'. \{H'\} t_1 \{Q'\} \Rightarrow \{H'\} (f x) \{Q'\})}{\{H\} (\text{let rec } f x = t_1 \text{ in } t_2) \{Q\}}$$

Exercises

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

Available from the webpage of the course.

The end!