Separation Logic
Part 2

Arthur Charguéraud

February 2014
Last week

- Type of heap predicates: $\text{Hprop} \equiv \text{Heap} \rightarrow \text{Prop}$
- Basic heap predicates: $[\_], [P], l \leftarrow v, H \ast H'$ and $\exists x. H$
- Heap implication: $H_1 \triangleright H_2 \equiv \forall m. H_1 m \Rightarrow H_2 m$
- Frame rule:

\[
\frac{\{H_1\} t \{Q_1\}}{\{H_1 \ast H_2\} t \{Q_1 \ast H_2\}}
\]

- Proof of specification by induction

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

Today’s course:

- lists
- list segments
- binary trees
- nested structures (e.g., lists of trees)
Content

Mutable lists
- Representation of lists
- Representation predicate for lists
- Example: mlength
- Examples: append and combine

List segments
Trees
Polymorphic containers
Null-terminated lists

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

\[ p \mapsto \text{Mlist}(8 :: 5 :: 6 :: \text{nil}) \]
Null pointers in OCaml

Interface:

    val null : 'a

Example:

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

Implementation:

    let null = Obj.magic ()
Construction of mutable lists

Representation of mutable lists:

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

Building a list of 3 elements:

```ocaml
let p0 = null in
let p1 = { hd = 6; tl = p0 } in
let p2 = { hd = 5; tl = p1 } in
let p = { hd = 8; tl = p2 } in
p
```
Heap predicates for records

\[ p \mapsto \{ \text{hd}=x; \; \text{tl}=p' \} \]

Shorthand: \( p \mapsto v \)

\[ p \mapsto \{ \text{contents}=v \} \]
First goal: define a heap predicate for mutable lists.

\[ p \rightsquigarrow \text{Mlist} \ L \]

The definition should be such that:

\[ p \rightsquigarrow \text{Mlist} (8 :: 5 :: 6 :: \text{nil}) \equiv \exists p_2 p_1. \]

\[ p \leftrightarrow \{\text{hd}=8; \text{tl}=p_2\} \]

* \[ p_2 \leftrightarrow \{\text{hd}=5; \text{tl}=p_1\} \]

* \[ p_1 \leftrightarrow \{\text{hd}=6; \text{tl}=\text{null}\} \]
Construction of a mutable list

Second goal: specify functions on mutable lists in terms of pure lists.

Example:

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

Specification:

\[ \forall pL. \{ p \leadsto \text{Mlist } L \} (\text{mlength } p) \{ \lambda n. [n = \text{length } L] \ast p \leadsto \text{Mlist } L \} \]
Representation predicate for mutable lists

Definition:

\[ p \leadsto \text{Mlist} \; L \quad \equiv \quad \text{match} \; L \; \text{with} \]
\[ \begin{align*}
| \text{nil} & \Rightarrow [p = \text{null}] \\
| x :: L' & \Rightarrow \exists p'. \; p \mapsto \{\text{hd} = x; \; \text{tl} = p'\} \ast p' \leadsto \text{Mlist} \; L'
\end{align*} \]

where \((L : \text{list} A)\) for some type \(A\), and \((p : \text{loc})\).
Unfolding the representation predicate

Representation predicate

\[ p \rightsquigarrow \text{Mlist } L \equiv \text{match } L \text{ with } \]

\[ \begin{align*}
| \text{nil} & \Rightarrow [p = \text{null}] \\
| x :: L' & \Rightarrow \exists p'. p \mapsto \{\text{hd}=x; \text{tl}=p'\} \ast p' \rightsquigarrow \text{Mlist } L'
\end{align*} \]

Application:

\[ p \rightsquigarrow \text{Mlist } (8 :: 5 :: 6 :: \text{nil}) \equiv \begin{align*}
\exists p_1. & \ p \mapsto \{\text{hd}=8; \text{tl}=p_1\} \\
\ast \exists p_2. & \ p_1 \mapsto \{\text{hd}=5; \text{tl}=p_2\} \\
\ast \exists p_3. & \ p_2 \mapsto \{\text{hd}=6; \text{tl}=p_3\} \\
\ast [p_3 = \text{null}] 
\end{align*} \]
Summary of arrows

list cell: \[ p \mapsto \{ \text{hd}=x; \text{tl}=p' \} \]

ref cell: \[ p \mapsto \{ \text{contents}=v \} \]

ref cell: \[ p \mapsto v \]

full list: \[ p \leadsto \text{Mlist } L \]

Remark: \[ p \mapsto \{ \text{hd}=x; \text{tl}=p' \} \] entails \( p \neq \text{null} \), and \( \text{null} \leadsto \text{Mlist } L \) entails \( L = \text{nil} \).
Separation of lists

\[ p_1 \rightsquigarrow \text{Mlist } L_1 \, \, \, * \, \, \, p_2 \rightsquigarrow \text{Mlist } L_2 \, \, \, * \, \, \, p_3 \rightsquigarrow \text{Mlist } L_3 \]

Separation Logic enforces:

- no cycles
- no sharing
Function `mlength`

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

Specification:

$$\forall p L. \{ p \leadsto Mlist L \} (m\text{length } p) \{ \lambda n. [n = \text{length } L] \ast p \leadsto Mlist L \}$$

Verification:

- Case $p = \text{null}$. Pre-condition is “null $\leadsto Mlist L$”, therefore $L$ is nil. The code returns 0, which matches “length nil”.
Verification of mlength, continued

- Case $p \neq \text{null}$. The list $L$ decomposes as $x :: L'$ and there exists $p'$ such that the state is: $p \mapsto \{ \text{hd}=x; \text{tl}=p' \} \ast p' \leadsto \text{Mlist } L'$.

\[
\frac{
\{ p' \leadsto \text{Mlist } L' \} \ (\text{mlength } p') \ \{ \lambda n. [n = |L'|] \ast p' \leadsto \text{Mlist } L' \}
}{
\{ p \mapsto \{ \text{hd}=x; \text{tl}=p' \} \} \ (\text{mlength } p') \ \{ \lambda n. [n = |L'|] \}
\ast p' \leadsto \{ \text{hd}=x; \text{tl}=p' \}
\ast p' \leadsto \text{Mlist } L'
\}
\]

\[
\frac{
\{ p \leadsto \text{Mlist } L \} \ (1 + \text{mlength } p.\text{tl}) \ \{ \lambda n. [n = |L|] \ast p \leadsto \text{Mlist } L \}
}{
\}
\]

where $L = x :: L'$ and $|L| = \text{length } L$.
Focus and defocus rules for cons

\[ p \leadsto \text{Mlist } L \equiv \text{match } L \text{ with } \]
\[ \begin{align*}
| \text{nil} & \Rightarrow [p = \text{null}] \\
| x :: L' & \Rightarrow \exists p'. p \mapsto \{|\text{hd=}x; \text{tl=}p'| \} \ast p' \leadsto \text{Mlist } L'
\end{align*} \]

\[ p \leadsto \text{Mlist } L \ast [p \neq \text{null}] \triangleright \exists x L' p'. p \mapsto \{|\text{hd=}x; \text{tl=}p'| \} \ast p' \leadsto \text{Mlist } L' \ast [L = x :: L'] \]

\[ p \mapsto \{|\text{hd=}x; \text{tl=}p'| \} \ast p' \leadsto \text{Mlist } L' \triangleright p \leadsto \text{Mlist } (x :: L') \]
Focus and defocus rules for nil

Derivable from the definition:

\[ p \leadsto \text{Mlist } L \equiv \text{match } L \text{ with} \]
\[ | \text{nil} \Rightarrow [p = \text{null}] \]
\[ | x :: L' \Rightarrow \exists p'. p \mapsto \{|\text{hd}=x; \text{tl}=p'| \} \ast p' \leadsto \text{Mlist } L' \]

Focus and defocus for empty lists:

\[ \text{null} \leadsto \text{Mlist } L \triangleright [L = \text{nil}] \]

\[ [] \triangleright \text{null} \leadsto \text{Mlist } \text{nil} \]
The function `mappend`

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

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

Specification of aux:

\[
\forall p_1 p_2 L_1 L_2. \quad \{ [p \neq \text{null} \land p \rightsquigarrow \text{Mlist}\ L \land p_2 \rightsquigarrow \text{Mlist}\ L_2] \\
\quad \quad \quad \quad \quad (\text{aux} \ p) \\
\quad \quad \quad \quad \quad \{ \lambda tt. \ p \rightsquigarrow \text{Mlist}\ (L \uplus L_2)\}\}
\]

Specification of mappend:

\[
\forall p_1 p_2 L_1 L_2. \quad \{ p_1 \rightsquigarrow \text{Mlist}\ L_1 \land p_2 \rightsquigarrow \text{Mlist}\ L_2 \} \\
\quad \quad \quad \quad \quad (\text{mappend}\ p_1 \ p_2) \\
\quad \quad \quad \quad \quad \{ \lambda q. \ q \rightsquigarrow \text{Mlist}\ (L_1 \uplus L_2)\}\}
\]
The function combine

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

What is the specification of mcombine (in terms of List.combine)?
The function combine: specification

∀p₁p₂L₁L₂.
\{p₁ \leadsto \text{Mlist } L₁ * p₂ \leadsto \text{Mlist } L₂ * [\text{length } L₁ = \text{length } L₂]\}
(combine p₁ p₂)
\{\lambda q. \, q \leadsto \text{Mlist } (\text{combine } L₁ \, L₂) * p₁ \leadsto \text{Mlist } L₁ * p₂ \leadsto \text{Mlist } L₂\}
Mutable lists

List segments
   Motivating example: split
   Representation of list segments
   Focus and defocus rules
   Application: mutable queues

Trees

Polymorphic containers
The function split-after

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

```ocaml
let split_after (i:int) (p:'a cell) =
  let q = nth_cell i p in
  let r = q.tl in
  q.tl <- null
  r
```
The function \texttt{nth-cell}

Returns the \textit{i}-th cell of a list:

\begin{verbatim}
let rec nth_cell (i:int) (p:'a cell) =
  if i = 0
  then p
  else nth_cell (i-1) (p.tl)
\end{verbatim}

How to complete the specification?

\begin{verbatim}
{p ~\rightarrow \text{Mlist} L * [0 \leq i < |L|]} (\text{split_after i} p) \{\lambda q. ...\}
\end{verbatim}
Specification of nth-cell:

\[\forall pL_i. \quad \{p \rightsquigarrow \text{Mlist } L \; \ast \; [0 \leq i < |L|]\}\]

\((\text{nth\_cell } i \; p)\)

\(\{\lambda q. \exists L_1L_2. \; p \rightsquigarrow \text{MlistSeg } q \; L_1 \; \ast \; q \rightsquigarrow \text{Mlist } L_2 \; \ast \; [L = L_1 \; \ast \; L_2 \; \land \; |L_1| = i]\}\)
Representation predicate for list segments

From the definition of Mlist:

\[ p \leadsto \text{Mlist}\ L \equiv \text{match } L \text{ with } \]
\[ \begin{align*}
| \text{nil} & \Rightarrow [p = \text{null}] \\
| x :: L' & \Rightarrow \exists p'. p \mapsto \{ \text{hd}=x; \text{tl}=p' \} \\
\end{align*} \]
\[ \ast p' \leadsto \text{Mlist}\ L' \]

to that of MlistSeg:

\[ p \leadsto \text{MlistSeg}\ q\ L \equiv \text{match } L \text{ with } \]
\[ \begin{align*}
| \text{nil} & \Rightarrow [p = q] \\
| x :: L' & \Rightarrow \exists p'. p \mapsto \{ \text{hd}=x; \text{tl}=p' \} \\
\end{align*} \]
\[ \ast p' \leadsto \text{MlistSeg}\ q\ L' \]

Lists as null-terminated list segments:

\[ p \leadsto \text{Mlist}\ L \equiv p \leadsto \text{MlistSeg}\ \text{null}\ L \]
Focus rules for segments

For cons at head, defocus rule:

\[ p \rightarrow \{ \text{hd}=x; \; \text{tl}=p' \} \; \ast \; p' \leadsto \text{MlistSeg } q \; L' \; \triangleright \; p \leadsto \text{MlistSeg } q \; (x :: L') \]
Focus and defocus rules for segments, continued

For glueing list segments, defocus rule:

\[ p \bowtie \text{MlistSeg} p' \ L_1 \ast \ p' \bowtie \text{MlistSeg} \ q \ L_2 \]

\[ \triangleright \quad p \bowtie \text{MlistSeg} \ q \ (L_1 \uplus L_2) \]

For splitting list segments, focus rule:

\[ p \bowtie \text{MlistSeg} \ q \ L \ast \ [L = L_1 \uplus L_2] \]

\[ \triangleright \quad \exists p'. \ p \bowtie \text{MlistSeg} \ p' \ L_1 \ast \ p' \bowtie \text{MlistSeg} \ q \ L_2 \]
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;
}
```

Representation predicate?

\[
p \rightsquigarrow \text{Queue} \ L \iff \exists fb. \ p \rightarrow \{\text{front} = f; \text{back} = b\}
\]

- \(f \rightsquigarrow \text{MlistSeg } b \ L\)
- \(\exists yq. \ b \rightarrow \{\text{hd} = y; \text{tl} = q\}\)
Abstract interface for mutable queues

Specify the function from the OCaml interface below:

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

in terms of an abstract representation predicate written:

\[ q \leadsto \text{Queue } L \]
Interface specification for mutable queues

\[
\begin{align*}
\text{create}() & \Rightarrow \lambda q. \text{Queue} \text{nil} \\
\text{is_empty} q & \Rightarrow \lambda b. [b = \text{true} \iff L = \text{nil}] \Rightarrow \text{Queue} L \\
\text{push} x q & \Rightarrow \lambda tt. \text{Queue} (x :: L) \\
\text{pop} q & \Rightarrow \lambda x. \exists L'. [L = L' \& x] \Rightarrow \text{Queue} L' \\
\text{transfer} q1 q2 & \Rightarrow \lambda tt. \text{Queue} \text{nil} \Rightarrow \text{Queue} (L1 ++ L2)
\end{align*}
\]

where \( L \& x = L ++ x :: \text{nil} \).
Summary

- $p \leadsto \text{MlistSeg} \ q \ \text{L}$
- $p \leadsto \text{Mlist} \ \equiv \ p \leadsto \text{MlistSeg} \ \text{null} \ q$
- focus/defocus rules for splitting/glueing segments
Content

Mutable lists
List segments

Trees
  Representation of binary trees
  Specification of binary trees
  Focus and defocus rules
  Enforcing invariants on the tree
  Application to binary search trees
  Application to red-black trees
  Interface: mutable set

Polymorphic containers
Implementation of a binary tree

Representation of binary trees, with null for empty trees:

```plaintext
type node = {
    mutable item : int;
    mutable left : node;
    mutable right : node;
}
```
Representation of a binary tree

$p \rightsquigarrow \text{Mtree } T$

$T$
Representation of pure trees

Type of pure binary trees in Coq:

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

Example:

```coq
Node 3 (Node 2 Leaf Leaf) (Node 4 (Node 5 Leaf Leaf) (Node 6 Leaf Leaf))
```
Copy of a binary tree

Construction of a copy of a tree:

```plaintext
let rec copy (t:node) : node =
    if t == null then null else
    let l = copy t.left;
    let r = copy t.right;
    { item = t.item; left = l; right = r }
```

Specification?

\[ \forall p T. \{ p \rightsquigarrow Mtree T \} (\text{copy } p) \{ \lambda p'. p \rightsquigarrow Mtree T \land p' \rightsquigarrow Mtree T \} \]
Binary trees: specification

Recall the definition:

\[ p \leadsto M\text{list } L \equiv \text{match } L \text{ with} \]
\[ \quad | \text{nil} \Rightarrow [p = \text{null}] \]
\[ \quad | x :: L' \Rightarrow \exists p'. p \mapsto \{\text{hd}=x; \text{tl}=p'\} \ast p' \leadsto M\text{list } L' \]

Exercise: define \( p \leadsto M\text{tree } T \).

\[ p \leadsto M\text{tree } T \equiv \text{match } T \text{ with} \]
\[ \quad | \text{Leaf} \Rightarrow [p = \text{null}] \]
\[ \quad | \text{Node } x \ T_1 \ T_2 \Rightarrow \exists p_1 p_2. \]
\[ \quad \quad p \mapsto \{\text{item}=x; \text{left}=p_1; \text{right}=p_2\} \]
\[ \quad \ast p_1 \leadsto M\text{tree } T_1 \]
\[ \quad \ast p_2 \leadsto M\text{tree } T_2 \]
Binary trees: focus and defocus

For nonempty trees:

\[ p \leadsto \text{Mtree} \, T \star [p \neq \text{null}] \rightarrow \exists x \, T_1 \, T_2 \, p_1 \, p_2. \ldots \]

\[ p \mapsto \{\text{item}=x; \, \text{left}=p_1; \, \text{right}=p_2\} \star p_1 \leadsto \text{Mtree} \, T_1 \star p_2 \leadsto \text{Mtree} \, T_2 \]
\[ \rightarrow p \leadsto \text{Mtree} \, (\text{Node} \, x \, T_1 \, T_2) \]

For empty trees:

\[ \text{null} \leadsto \text{Mtree} \, T \rightarrow [T = \text{Leaf}] \]
\[ [] \rightarrow \text{null} \leadsto \text{Mtree} \, \text{Leaf} \]
Verification of tree copy

\[ \forall pT. \ {p \rightsquigarrow Mtree T} \ (\text{copy} \ p) \ \{\lambda p'. \ p \rightsquigarrow Mtree T * p' \rightsquigarrow Mtree T\} \]

Verification of the copy function:

\[
p \mapsto \{\text{item}=x; \ \text{left}=p_1; \ \text{right}=p_2\} \ * \ p_1 \rightsquigarrow Mtree T_1 \ * \ p_2 \rightsquigarrow Mtree T_2
\]

\[
p' \mapsto \{\text{item}=x; \ \text{left}=p'_1; \ \text{right}=p'_2\} \ * \ p'_1 \rightsquigarrow Mtree T_1 \ * \ p'_2 \rightsquigarrow Mtree T_2
\]
Invariant on trees

tree with 0 or 2 children

complete binary tree

binary search tree

red-black tree
Invariant on binary trees: zero or two children

Exercise: define $p \leadsto \text{Mtree2 } T$ to enforce that every node has exactly zero or two non-null children.

$p \leadsto \text{Mtree2 } T \iff \text{match } T \text{ with }$

$\begin{align*}
| \text{Leaf} \Rightarrow [p = \text{null}] \\
| \text{Node } x & T_1 T_2 \Rightarrow \exists p_1 p_2. \\
& p \mapsto \{ \text{item} = x; \text{left} = p_1; \text{right} = p_2 \} \\
* p_1 \leadsto \text{Mtree2 } T_1 \\
* p_2 \leadsto \text{Mtree2 } T_2 \\
* [p_1 = \text{null} \iff p_2 = \text{null}] 
\end{align*}$

Remark: last condition could also be $[T_1 = \text{Leaf} \iff T_2 = \text{Leaf}]$. 

Problem with modified representation predicate

Specification of copy:

\[ \{ p \leadsto \text{Mtree} T \} (\text{copy } p) \{ \lambda p'. \ p \leadsto \text{Mtree} T \ast p' \leadsto \text{Mtree} T \} \]

How to apply it to a tree with zero or two children?

\[ \{ p \leadsto \text{Mtree}2 T \} (\text{copy } p) \{ \lambda p'. \ p \leadsto \text{Mtree}2 T \ast p' \leadsto \text{Mtree}2 T \} \]
Invariants expressed on the pure representation

Better definition, reusing the existing representation predicate:

\[ p \mapsto \text{Mtree2 } T \equiv p \mapsto \text{Mtree } T \ast [\text{nounary } T] \]

Inductive properties over pure trees:

\textbf{Inductive nounary : }\text{tree }\to \text{Prop } :=

\begin{align*}
| \text{nounary_leaf :} \\
& \text{nounary Leaf} \\
| \text{nounary_node : } \forall x \ T1 \ T2,
& \text{nounary } T1 \to \\
& \text{nounary } T2 \to \\
& (T1 = \text{Leaf } \leftrightarrow T2 = \text{Leaf}) \to \\
& \text{nounary } (\text{Node } x \ T1 \ T2)
\end{align*}
Copying a tree with invariants

Specification of copy:

\[
\begin{align*}
\{ p \rightsquigarrow \text{Mtree } T \} \text{ (copy } p \text{) } \{ \lambda p'. \ p \rightsquigarrow \text{Mtree } T \ast p' \rightsquigarrow \text{Mtree } T \} \\
\{ p \rightsquigarrow \text{Mtree } T \} \text{ (copy } p \text{) } \{ \lambda p'. \ p \rightsquigarrow \text{Mtree } T \} \\
\ast [\text{nounary } T] \\
\ast [\text{nounary } T] \\
\ast p' \rightsquigarrow \text{Mtree } T \\
\ast [\text{nounary } T] \\
\{ p \rightsquigarrow \text{Mtree2 } T \} \text{ (copy } p \text{) } \{ \lambda p'. \ p \rightsquigarrow \text{Mtree2 } T \ast p' \rightsquigarrow \text{Mtree2 } T \}
\end{align*}
\]
Complete binary trees

Exercise: define \( p \rightsquigarrow \text{MtreeComplete } T \) (reusing \( \text{Mtree} \)) to describe a binary tree such that all the leaves are exactly at the same depth.

\[
p \rightsquigarrow \text{MtreeComplete } T \equiv p \rightsquigarrow \text{Mtree } T \ast [\text{complete } T]
\]
Complete binary trees

Tree completeness as an inductive property over pure trees:

**Definition** complete T :=
\[ \exists n, \text{depth } n \ T. \]

**Inductive** depth : int \to tree \to Prop :=
\[ \begin{align*}
| \ & \text{depth}_\text{leaf} : \\
& \text{depth } 0 \ \text{Leaf} \\
| \ & \text{depth}_\text{node} : \forall n \ x \ T_1 \ T_2, \\
& \text{depth } n \ T_1 \ \to \\
& \text{depth } n \ T_2 \ \to \\
& \text{depth } (n+1) \ (\text{Node } x \ T_1 \ T_2).
\end{align*} \]
Exposing invariants in representation predicates

Definition from previous slide:

\[ p \circ M\text{treeComplete} T \equiv p \circ M\text{tree} T \land \exists n. \text{depth } n T \]

Another useful definition:

\[ p \circ M\text{treeDepth} n T \equiv p \circ M\text{tree} T \land \left[ \text{depth } n T \right] \]

Example of a “smart constructor”:

\[ \forall xp_1p_2T_1T_2n. \{ p_1 \circ M\text{treeDepth} n T_1 \land p_2 \circ M\text{treeDepth} n T_2 \} \]
\[ \{ \text{item} = x; \text{left} = p_1; \text{right} = p_2 \} \]
\[ \{ \lambda p. p \circ M\text{treeDepth} (n + 1) (\text{Node } x T_1 T_2) \} \]
Exercise: define \( p \leadsto \text{Msearchtree } E \) to describe a binary search tree that represents the set \( E \).

\[
p \leadsto \text{Msearchtree } E \quad \equiv \quad \exists T. \ p \leadsto \text{Mtree } T \ast \text{[search } T \ E]\]
Binary search trees

Characterization of purely-functional search trees:

**Inductive search**: $\text{tree} \rightarrow \text{set int} \rightarrow \text{Prop} :=$

$\mid \text{search_leaf} :$

$\quad \text{search Leaf } \emptyset$

$\mid \text{search_node} : \forall x \ T1 \ T2,$

$\quad \text{search T1 E1 } \rightarrow$

$\quad \text{search T2 E2 } \rightarrow$

$\quad \text{foreach (is_lt x) E1 } \rightarrow$

$\quad \text{foreach (is_gt x) E2 } \rightarrow$

$\quad \text{search (Node x T1 T2) (\{x\} } \cup E1 \cup E2).$
Red-black trees

Invariants on red-black-trees:

- Every node has color either red or black.
- The root must be black.
- Empty subtrees are considered to be black.
- Every red node must have two black children.
- Every path from a given node to any of its descendant leaves contains the same number of black nodes.
Representation of red-black trees

Assume that Mtree and search are extended to handle the color field.

**Definition color T:**

\[
\text{match } T \text{ with } \\
| \text{Leaf } \Rightarrow \text{Black} \\
| \text{Node } c \times T_1 T_2 \Rightarrow c
\]

Representation predicate:

\[
p \rightsquigarrow \text{Mrbtree } E \equiv \exists T. \; p \rightsquigarrow \text{Mtree } T \ast [\; \text{search } T E \;] \\
\land \; \text{color } T = \text{Black} \\
\land \; \exists n. \; \text{rbtree } n \; T
\]

where “rbtree n T” formalizes the red-black tree invariants.
Red-black trees

Predicate “rbtree n T” asserts that T is a red-black tree that has n black nodes in every path.

**Inductive rbtree : int → tree → Prop :=**

| rbtree_leaf : |
| rbtree 0 Leaf |
| rbtree_node : ∀ n m c x T1 T2, |
| (c = Red → color T1 = Black ∧ color T2 = Black) → |
| (m = if (c = Black) then n–1 else n) → |
| rbtree m T1 → |
| rbtree m T2 → |
| rbtree n (Node c x T1 T2) |
Abstract interface for mutable sets

“$p \rightsquigarrow \text{Mrbtree } E$” is an instance of a mutable set “$p \rightsquigarrow \text{Mset } E$”.

Specify the functions from the OCaml interface below:

```ocaml
create : unit -> 'a set
is_empty : 'a set -> bool
mem : 'a -> 'a set -> bool
add : 'a -> 'a set -> unit
rem : 'a -> 'a set -> unit
```

in terms of an abstract representation predicate $p \rightsquigarrow \text{Mset } E$. 
Interface specification for mutable sets

\[
\begin{align*}
\{[]\} \ (\text{create}()) & \{\lambda p.\ p \rightsquigarrow \text{Mset} \emptyset\} \\
\{p \rightsquigarrow \text{Mset} E\} \ (\text{is\_empty} \ p) & \{\lambda b.\ [b = \text{true} \iff E = \emptyset] \ast p \rightsquigarrow \text{Mset} E\} \\
\{p \rightsquigarrow \text{Mset} E\} \ (\text{mem} \ x \ p) & \{\lambda b.\ [b = \text{true} \iff x \in E] \ast p \rightsquigarrow \text{Mset} E\} \\
\{p \rightsquigarrow \text{Mset} E\} \ (\text{add} \ x \ p) & \{\lambda tt.\ p \rightsquigarrow \text{Mset} (E \cup \{x\})\} \\
\{p \rightsquigarrow \text{Mset} E\} \ (\text{rem} \ x \ p) & \{\lambda tt.\ p \rightsquigarrow \text{Mset} (E \setminus \{x\})\}
\end{align*}
\]
Summary

Specification of mutable sets in terms of pure sets:

\[ \{ p \leadsto \text{Mset } E \} \ (\text{add } x \ p) \ \{ \lambda t t. \ p \leadsto \text{Mset } (E \cup \{x\}) \} \]

Implementation as mutable binary trees with pure invariants:

\[ p \leadsto \text{Mrbtree } E \equiv \exists T. \ p \leadsto \text{Mtree } T \ast \left[ \begin{array}{c} \text{search } T E \\ \wedge \text{color } T = \text{Black} \\ \wedge \exists n. \ \text{rbtree } n \ T \end{array} \right] \]

Separation Logic specification of binary trees:

\[ p \leadsto \text{Mtree } T \equiv \text{match } T \text{ with} \]

\[ \begin{array}{c} \text{Leaf } \Rightarrow [p = \text{null}] \\ \text{Node } x \ T_1 \ T_2 \Rightarrow \exists p_1 p_2. \\
 p \mapsto \{ \text{item}=x; \ \text{left}=p_1; \ \text{right}=p_2 \} \end{array} \]

\[ \ast p_1 \leadsto \text{Mtree } T_1 \ast p_2 \leadsto \text{Mtree } T_2 \]
Mutable lists
List segments
Trees
Polymorphic containers
  Mutable lists of mutable lists
  Polymorphic list representation predicate
  Application to arrays, queues, ...
  Application to records
  Examples with binary trees
Mutable lists of mutable lists

The goal is to specify a mutable lists of mutable lists:
Representation as a list of list

Specify it with respect to a pure list of pure lists:

\[
p \leadsto \text{MlistOfMlist} \ ((5 :: 7 :: \text{n}il) :: (8 :: 3 :: 3 :: \text{n}il) :: (n\text{il}) :: (4 :: \text{n}il) :: \text{n}il)\]
Mutable lists of mutable lists

A first attempt:

\[ p \leadsto \text{MlistOfMlist} K \equiv \exists L. \ p \leadsto \text{Mlist} L \ast \bigodot p_i \leadsto \text{Mlist} (K[i]) \]

Problem: involves reasoning about indices.
Representation predicate for lists of lists

Representation of lists:

\[ p \rightsquigarrow \text{Mlist}\ L \quad \equiv \quad \text{match}\ L\ \text{with} \]
\[ \quad |\ \text{nil} \Rightarrow [p = \text{null}] \]
\[ \quad |\ x :: L' \Rightarrow \exists p'. \ p \mapsto \{|\text{hd}=x;\ \text{tl}=p'|\} * p' \rightsquigarrow \text{Mlist}\ L' \]

Generalization to lists of lists:

\[ p \rightsquigarrow \text{MlistOfMlist}\ L \quad \equiv \quad \text{match}\ L\ \text{with} \]
\[ \quad |\ \text{nil} \Rightarrow [p = \text{null}] \]
\[ \quad |\ K :: L' \Rightarrow \exists xp'. \ p \mapsto \{|\text{hd}=x;\ \text{tl}=p'|\} * p' \rightsquigarrow \text{MlistOfMlist}\ L' \]
\[ \quad * \ x \rightsquigarrow \text{Mlist}\ K \]
Example: flatten

For pure lists, we have:

\[
\text{flatten} : \text{('a list) list} \rightarrow \text{'a list}
\]

Specification of the equivalent function for mutable lists:

\[
\{ p \leadsto \text{MlistOfMlist} L \} \ (\text{mflatten} \ p) \ \{ \lambda q. \ q \leadsto \text{Mlist} (\text{flatten} L) \}
\]
Generalization to polymorphic lists

Representation of lists of lists:

\[ p \leadsto \text{MlistOfMlist } L \equiv \text{match } L \text{ with} \]
\[ | \text{nil} \Rightarrow [p = \text{null}] \]
\[ | K :: L' \Rightarrow \exists x p'. \quad p \mapsto \{\text{hd} = x; \text{tl} = p'\} \]
\[ * \quad p' \leadsto \text{MlistOfMlist } L' \]
\[ * \quad x \leadsto \text{Mlist } K \]

Representation of lists of anything:

\[ p \leadsto \text{Mlistof } R \ L \equiv \text{match } L \text{ with} \]
\[ | \text{nil} \Rightarrow [p = \text{null}] \]
\[ | X :: L' \Rightarrow \exists x p'. \quad p \mapsto \{\text{hd} = x; \text{tl} = p'\} \]
\[ * \quad p' \leadsto \text{Mlistof } R \ L' \]
\[ * \quad x \leadsto R \ X \]

\[ p \leadsto \text{MlistOfMlist } K \equiv p \leadsto \text{Mlistof Mlist } K \]
Polymorphic list representation predicate

Polymorphic representation predicate for lists

\[ p \rightsquigarrow \text{Mlistof } R L \equiv \text{match } L \text{ with} \]
\[ \begin{cases} \text{nil} \Rightarrow [p = \text{null}] \\ X :: L' \Rightarrow \exists x p'. \quad p \mapsto \{ \text{hd}=x; \text{tl}=p' \} \end{cases} \]

* \( p' \rightsquigarrow \text{Mlistof } R L' \)
* \( x \rightsquigarrow R X \)

Remark:

\[ p \rightsquigarrow \text{Mlist } L \equiv p \rightsquigarrow \text{Mlistof } \text{id } L \]

where \( x \rightsquigarrow \text{id } X \) is defined as \( [x = X] \).
Checking the types

\[ p \leadsto \text{Mlistof } R L \]

- \( p \) points to cells storing values of type \( a \)
- \( L \) has type “list \( A \)”
- an item \( x \) of type \( a \) is represented as \( X \) of type \( A \)
- the corresponding heap predicate is: \( x \leadsto R X \)

Implementation of the arrow notation:

\[ p \leadsto Q \quad \equiv \quad Q p \]
\[ p \leadsto \text{Mlistof } R L \quad \equiv \quad \text{Mlistof } R L p \]

\text{Mlistof } (a:\text{Type})(A:\text{Type})(R:A\rightarrow a\rightarrow \text{Hprop})(L:\text{list } A)(p:\text{loc}) : \text{Hprop}
Application to arrays, queues, ...

An array is represented using a finite map $M$:

- $p \mapsto [\| M[0], M[1], \ldots, M[n - 1] |]$  
- $p \rightsquigarrow \text{Array } M$ describes the same array  
- $p \rightsquigarrow \text{Arrayof } RM$ describes the array with items represented using $R$

array of arrays: $\quad p \rightsquigarrow \text{Arrayof } (\text{Arrayof } \text{Id}) K$
array of lists: $\quad p \rightsquigarrow \text{Arrayof } (\text{Mlistof } \text{Id}) K$
queue of arrays: $\quad p \rightsquigarrow \text{Queueof } (\text{Arrayof } \text{Id}) K$
list of trees: $\quad p \rightsquigarrow \text{Mlistof } \text{Mtree } K$
array of arrays of trees: $\quad p \rightsquigarrow \text{Arrayof } (\text{Arrayof } \text{Mtree}) K$
Representation predicate for records

\[ p \rightsquigarrow \text{Mlistof } R \ L \ \equiv \ \text{match } L \text{ with} \]
\[ | \text{nil} \Rightarrow [p = \text{null}] \]
\[ | X :: L' \Rightarrow \exists x p'. \quad p \mapsto \{ \text{hd}=x; \text{tl}=p' \} \]
\[ * \ p' \rightsquigarrow \text{Mlistof } R \ L' \]
\[ * \ x \rightsquigarrow R \ X \]

\[ p \rightsquigarrow \text{Mlistof } R \ L \ \equiv \ \text{match } L \text{ with} \]
\[ | \text{nil} \Rightarrow [p = \text{null}] \]
\[ | X :: L' \Rightarrow p \rightsquigarrow \text{Cellof } R \ X \ (\text{Mlistof } R) \ L' \]
Representation predicate for records

\[
\begin{align*}
p \leadsto \text{Cellof } R_h H R_t T & \quad \equiv \quad \exists h t. \quad p \mapsto \{ \text{hd=}h; \text{tl=}t\} \quad \text{\textcopyright{}} \quad R_h H \\
\quad \ast x \leadsto R_h H \\
\quad \ast y \leadsto R_t T
\end{align*}
\]
Application to binary trees

Example: definition of Mtree (for integer leaves, without holes) using Nodeof.

\[
p \mapsto \text{Mtree } T \equiv \text{match } T \text{ with} \\
| \text{Leaf} \Rightarrow [p = \text{null}] \\
| \text{Node } x \ T_1 \ T_2 \Rightarrow p \mapsto \text{Nodeof } \text{id } x \ \text{Mtree } T_1 \ \text{Mtree } T_2
\]
Application to n-ary trees

Definition of n-ary trees, where children is a mutable list of subtrees.

type node = {
    mutable item : int;
    mutable children : node cell
}
Application to n-ary trees

Consider the pure representation of n-ary trees:

\[
\text{Inductive tree :=} \\
\quad \mid \text{Leaf : tree} \\
\quad \mid \text{Node : int} \to \text{list tree} \to \text{tree}.
\]

Exercise: define Mtree, using Nodeof.

Representation predicate:

\[
p \rightsquigarrow \text{Mtree } T \equiv \text{match } T \text{ with} \\
\quad \mid \text{Leaf } \Rightarrow [p = \text{null}] \\
\quad \mid \text{Node } x C \Rightarrow p \rightsquigarrow \text{Nodeof Id } x (\text{Mlistof Mtree}) C
\]