Separation Logic

Part 2

Arthur Charguéraud

February 2014

Last week

- Type of heap predicates: Hprop $\equiv$ Heap $\rightarrow$ Prop
- Basic heap predicates: $[\,]$, $[P]$, $l \hookrightarrow v$, $H * H'$ and $\exists x.\, H$
- Heap implication: $H_1 \rhd H_2 \;\equiv\; \forall m.\, H_1\, m \;\Rightarrow\; H_2\, m$
- Frame rule:

$$\frac{\{H_1\}\; t\; \{Q_1\}}{\{H_1 * H_2\}\; t\; \{Q_1 \star H_2\}}$$

- Proof of specification by induction

$$\forall n. \quad \{[n \geqslant 0]\}\; (f\, n)\; \{\lambda r.\, [r = 2n]\}$$

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

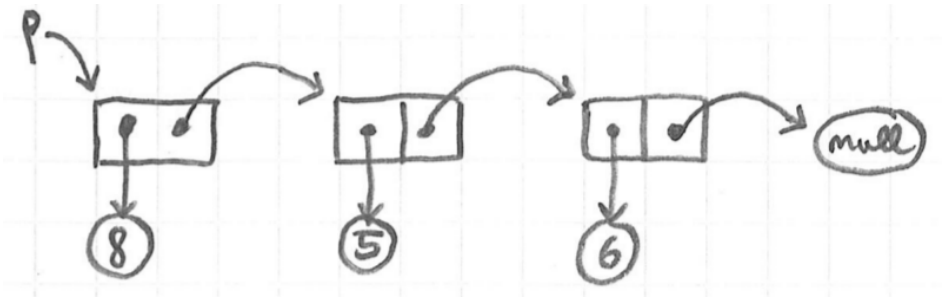
List segments

Trees

Polymorphic containers

Null-terminated lists

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



$$p \rightsquigarrow \mathsf{Mlist}\,(8 :: 5 :: 6 :: \mathsf{nil})$$

Null pointers in OCaml

Interface:

```
val null : 'a
```

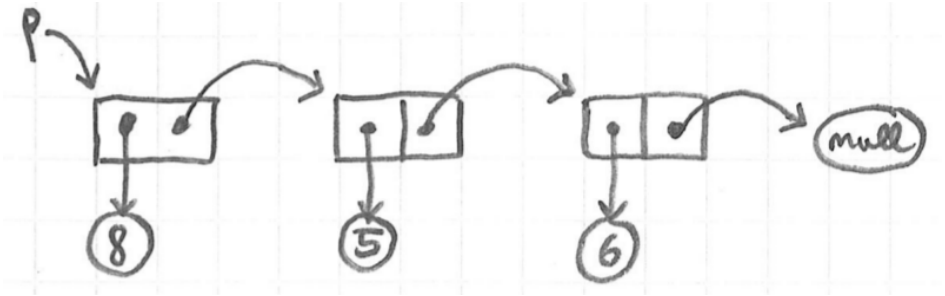
Example:

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

Implementation:

```
let null = Obj.magic ()
```

Construction of mutable lists



Representation of mutable lists:

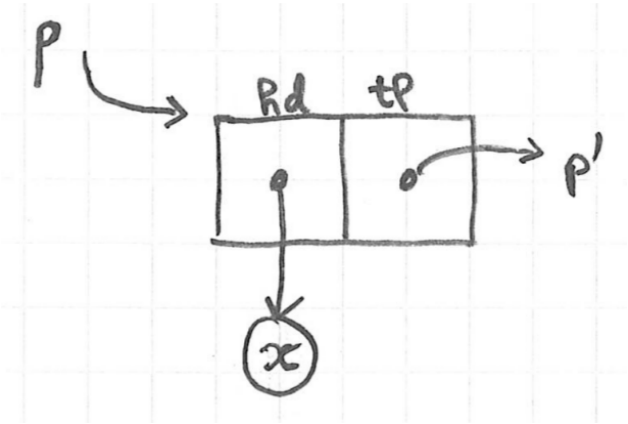
```
type 'a cell = { mutable hd : 'a;
                 mutable tl : 'a cell }
```

Building a list of 3 elements:

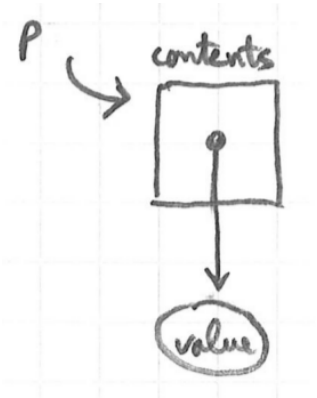
```
let p0 = null in
let p1 = { hd = 6; tl = p0 } in
let p2 = { hd = 5; tl = p1 } in
let p  = { hd = 8; tl = p2 } in
p
```

Heap predicates for records

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

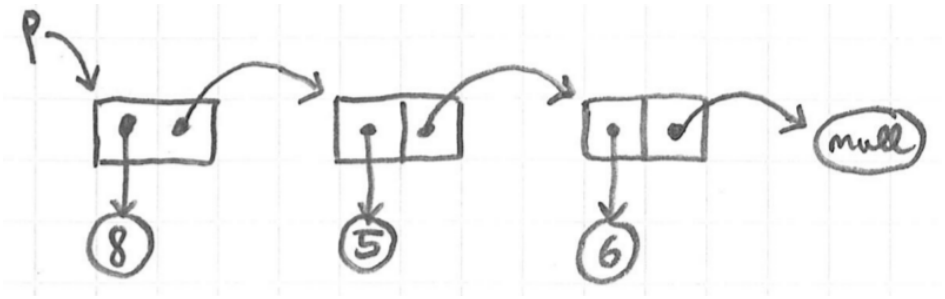


$p \mapsto \{|\mathsf{contents}{=}v|\}$



Shorthand: $p \hookrightarrow v$

Representation of mutable lists



First goal: define a heap predicate for mutable lists.

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

The definition should be such that:

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

Construction of a mutable list

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

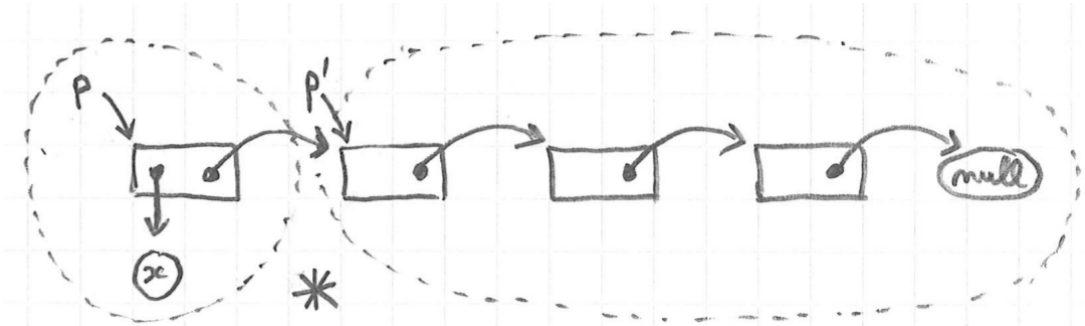
Example:

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

Specification:

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

Representation predicate for mutable lists



Definition:

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

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

Unfolding the representation predicate

Representation predicate

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

Application:

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

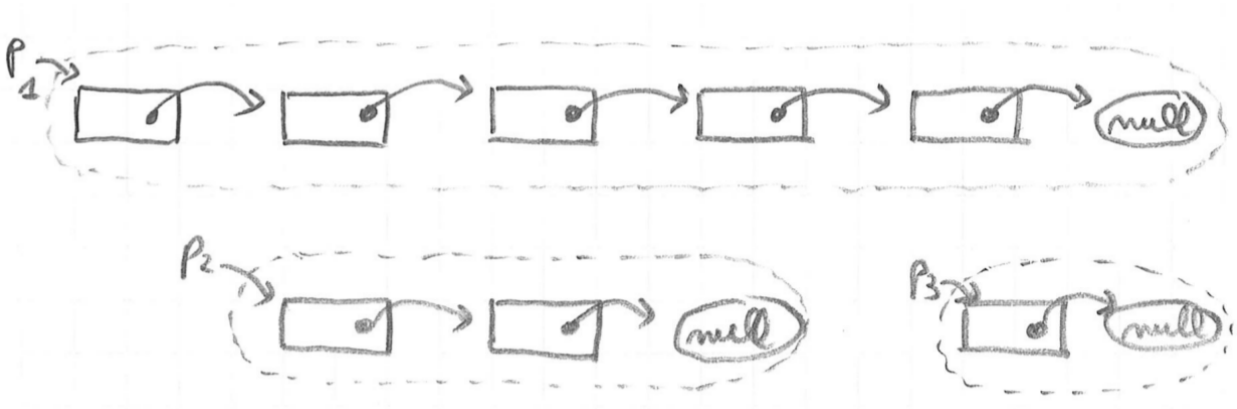
Summary of arrows

list cell:	$p \mapsto \{\!\|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'\|\!\}$
ref cell:	$p \mapsto \{\!\|\mathsf{contents}{=}v\|\!\}$
ref cell:	$p \hookrightarrow v$
full list:	$p \rightsquigarrow \mathsf{Mlist}\, L$

Remark: $p \mapsto \{\!\|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'\|\!\}$ entails $p \neq \mathsf{null}$,
and $\mathsf{null} \rightsquigarrow \mathsf{Mlist}\, L$ entails $L = \mathsf{nil}$.

Separation of lists

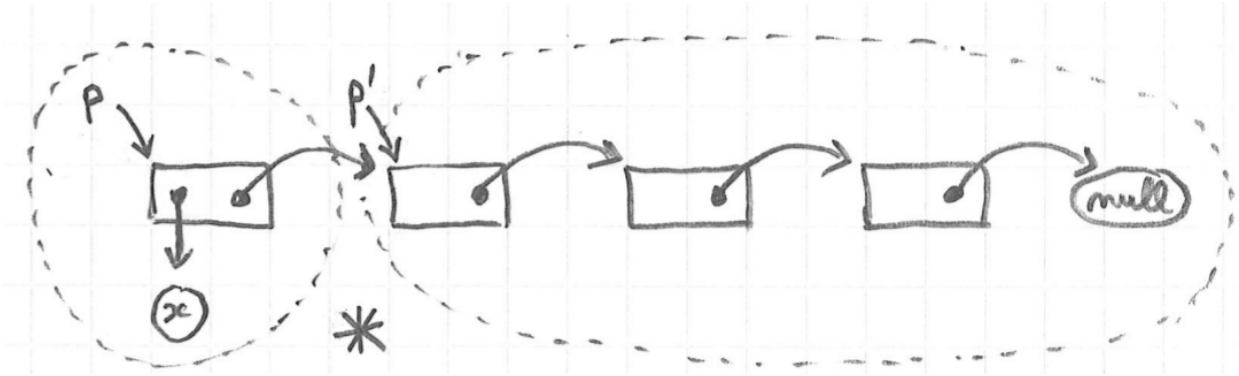
$$p_1 \rightsquigarrow \mathsf{Mlist}\, L_1 \quad * \quad p_2 \rightsquigarrow \mathsf{Mlist}\, L_2 \quad * \quad p_3 \rightsquigarrow \mathsf{Mlist}\, L_3$$



Separation Logic enforces:

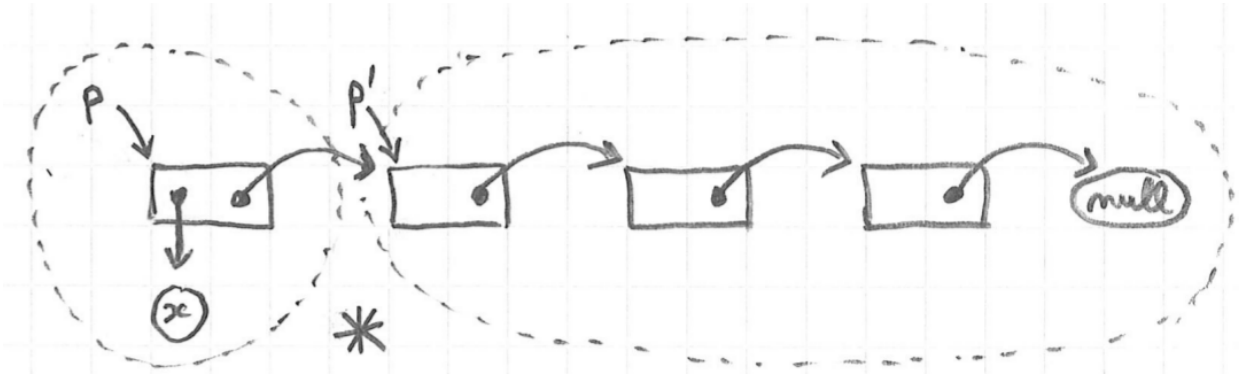
- no cycles
- no sharing

Function mlength




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

Verification of mlength



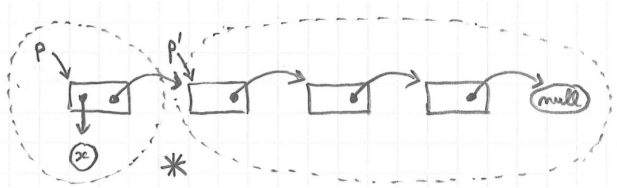
Specification:

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

Verification:

- Case $p = \mathsf{null}$. Pre-condition is "$\mathsf{null} \rightsquigarrow \mathsf{Mlist}\, L$", therefore L is nil. The code returns 0, which matches "length nil".

Verification of mlength, continued



- Case $p \neq \mathsf{null}$. The list L decomposes as $x :: L'$ and there exists p' such that the state is: $p \mapsto \{\!|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\!\} * p' \rightsquigarrow \mathsf{Mlist}\, L'$.

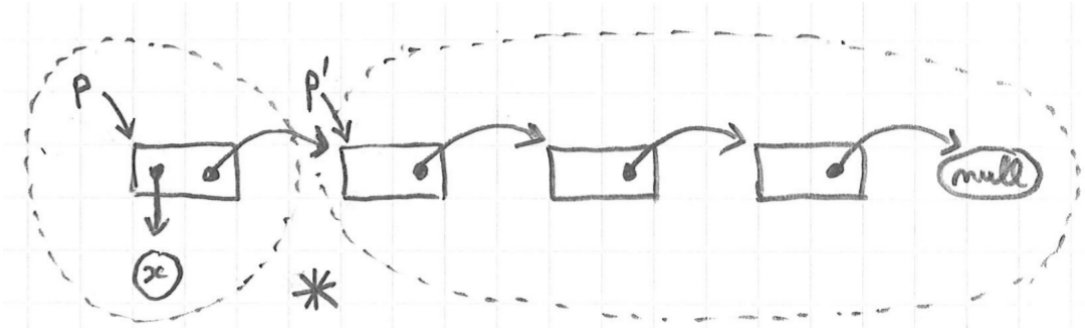
$$\frac{\{p' \rightsquigarrow \mathsf{Mlist}\, L'\}\ (\texttt{mlength p'})\ \{\lambda n.\, [n = |L'|] * p' \rightsquigarrow \mathsf{Mlist}\, L'\}}{\begin{array}{l}\{\ p \mapsto \{\!|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\!\}\ \}\ (\texttt{mlength p'})\ \{\lambda n.\ \ [n = |L'|] \qquad\qquad \}\\ \quad *p' \rightsquigarrow \mathsf{Mlist}\, L' \qquad\qquad\qquad\qquad\qquad\qquad *p \mapsto \{\!|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\!\}\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad *p' \rightsquigarrow \mathsf{Mlist}\, L'\end{array}}$$

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

where $L = x :: L'$ and $|L| = \mathsf{length}\, L$.

Focus and defocus rules for cons

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



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

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

Focus and defocus rules for nil

Derivable from the definition:

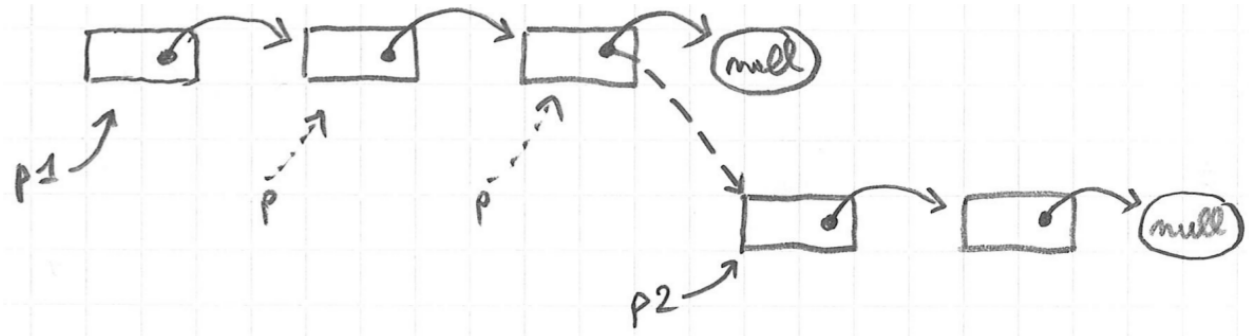
$$
\begin{array}{lcl}
p \rightsquigarrow \mathsf{Mlist}\, L & \equiv & \mathsf{match}\, L\, \mathsf{with} \\
& & \quad \mid \mathsf{nil} \;\Rightarrow\; [p = \mathsf{null}] \\
& & \quad \mid x :: L' \;\Rightarrow\; \exists p'.\; p \mapsto \{\!|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\!\} \;*\; p' \rightsquigarrow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

Focus and defocus for empty lists:

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

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

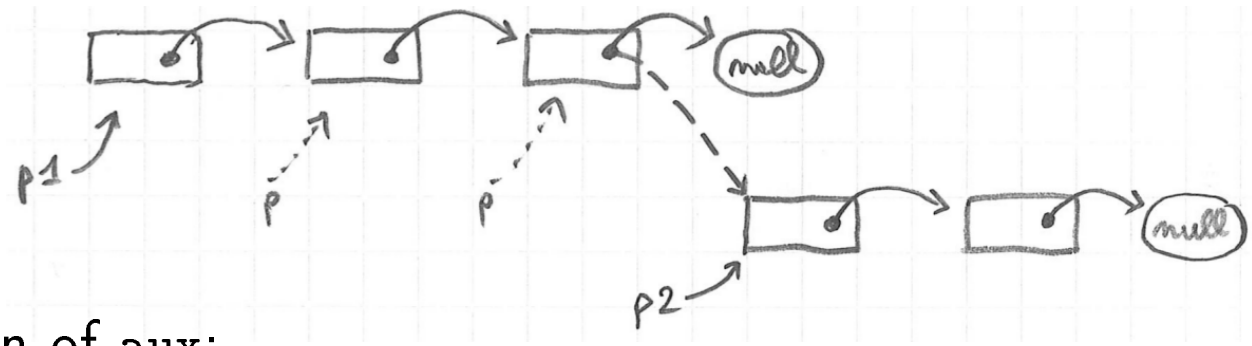
The function mappend




```
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



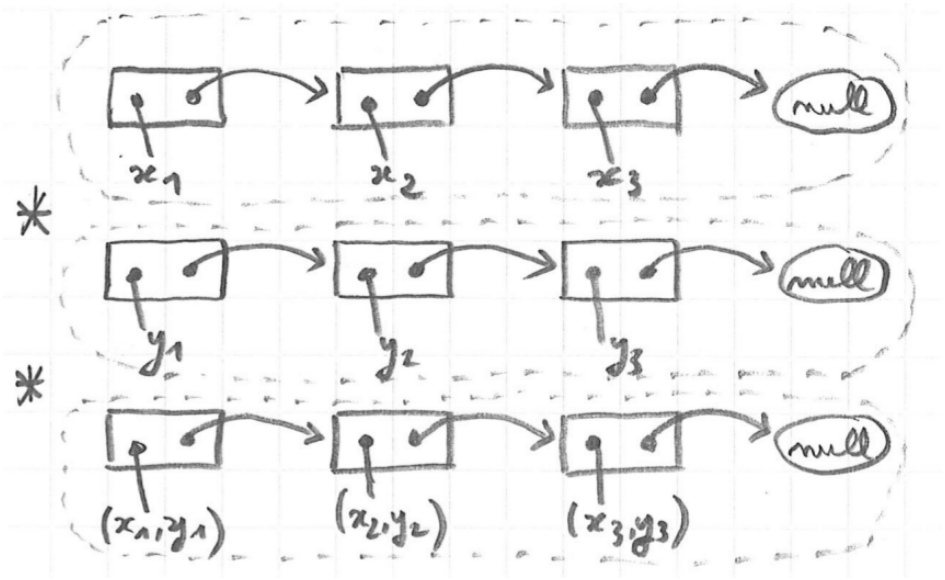
Specification of `aux`:

$$\forall p p_2 L L_2. \quad \{[p \neq \mathsf{null}] * p \rightsquigarrow \mathsf{Mlist}\, L * p_2 \rightsquigarrow \mathsf{Mlist}\, L_2\}$$
$$(\texttt{aux p})$$
$$\{\lambda tt.\ p \rightsquigarrow \mathsf{Mlist}\,(L +\!\!+ L_2)\}$$

Specification of `mappend`:

$$\forall p_1 p_2 L_1 L_2. \quad \{p_1 \rightsquigarrow \mathsf{Mlist}\, L_1 * p_2 \rightsquigarrow \mathsf{Mlist}\, L_2\}$$
$$(\texttt{mappend p1 p2})$$
$$\{\lambda q.\ q \rightsquigarrow \mathsf{Mlist}\,(L_1 +\!\!+ L_2)\}$$

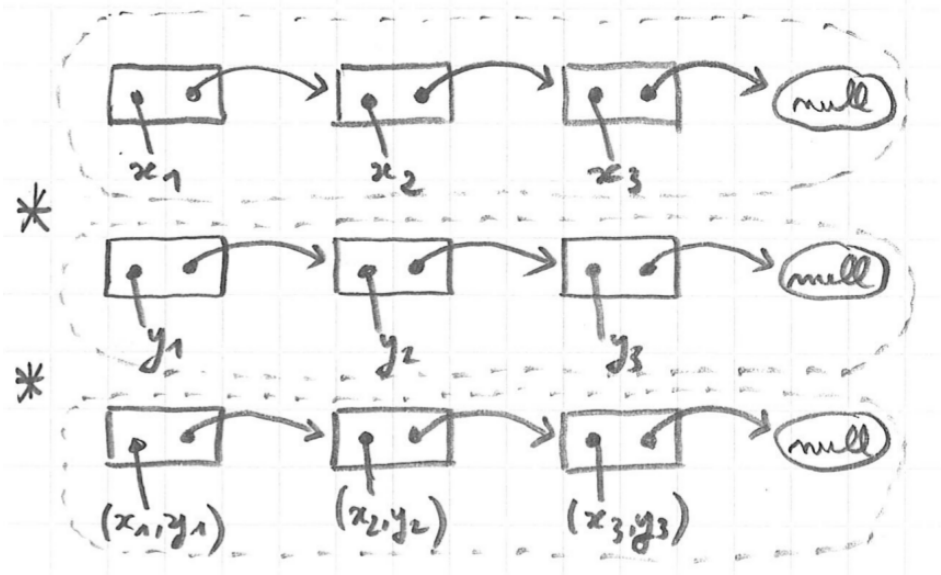
The function combine




```
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



$\forall p_1 p_2 L_1 L_2.$

$\{p_1 \rightsquigarrow \mathsf{Mlist}\, L_1 \ * \ p_2 \rightsquigarrow \mathsf{Mlist}\, L_2 \ * \ [\mathsf{length}\, L_1 = \mathsf{length}\, L_2]\}$

$(\texttt{combine p1 p2})$

$\{\lambda q.\ q \rightsquigarrow \mathsf{Mlist}\,(\mathsf{combine}\, L_1\, L_2) \ * \ p_1 \rightsquigarrow \mathsf{Mlist}\, L_1 \ * \ p_2 \rightsquigarrow \mathsf{Mlist}\, L_2\}$

Content

Mutable lists

List segments

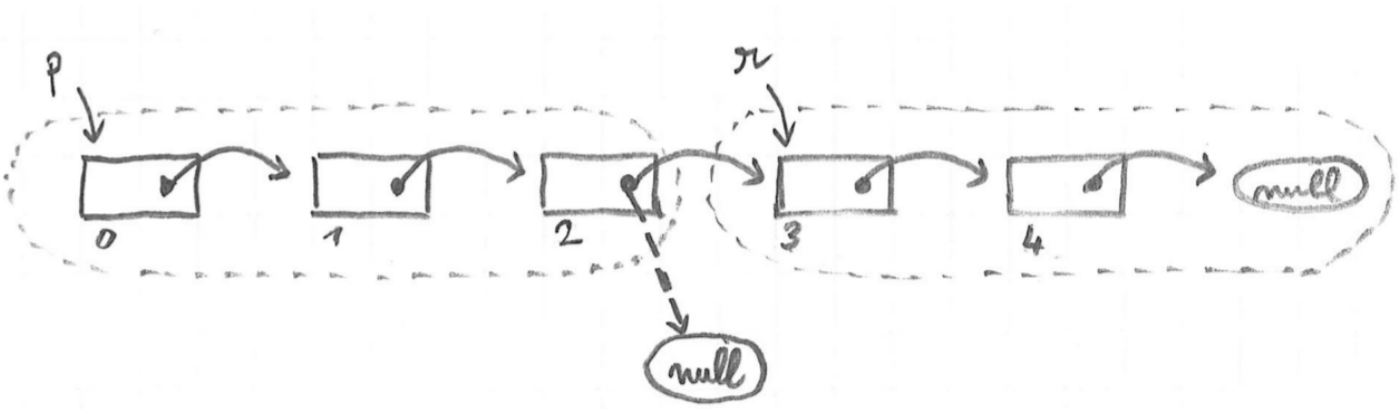
- Motivating example: split
- Representation of list segments
- Focus and defocus rules
- Application: mutable queues

Trees

Polymorphic containers

The function split-after

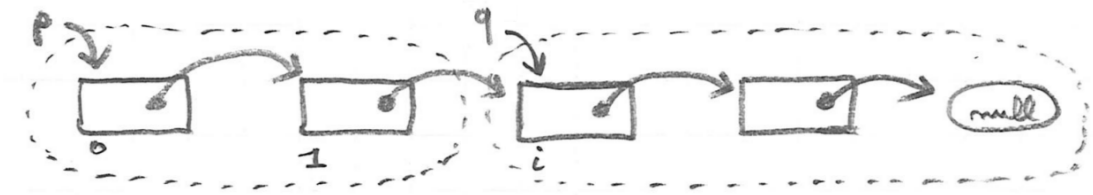
Splits a list p after the item at index i, and returns the chopped sublist:




```
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 nth-cell

Returns the i-th cell of a list:

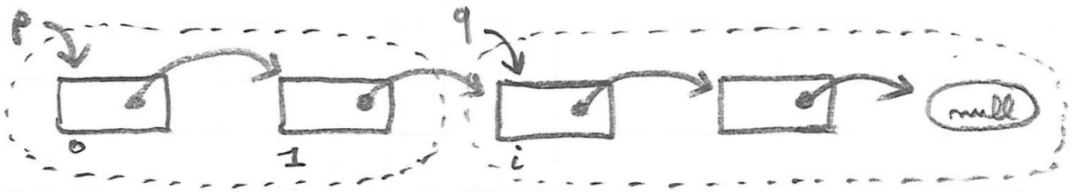



```
let rec nth_cell (i:int) (p:'a cell) =
  if i = 0
    then p
    else nth_cell (i-1) (p.tl)
```

How to complete the specification?

$$\{p \rightsquigarrow \mathsf{Mlist}\, L \ast [0 \leqslant i < |L|]\}\ (\texttt{split_after i p})\ \{\lambda q.\ \ldots\}$$

Specification of nth-cell



Specification of nth-cell:

$$
\begin{array}{ll}
\forall p L i. & \{p \rightsquigarrow \mathsf{Mlist}\, L \ * \ [0 \leqslant i < |L|]\} \\
& (\texttt{nth_cell i p}) \\
& \{\lambda q.\ \exists L_1 L_2.\ p \rightsquigarrow \mathsf{MlistSeg}\, q\, L_1 \ * \ q \rightsquigarrow \mathsf{Mlist}\, L_2 \\
& \qquad\qquad * \, [L = L_1 +\!\!+ L_2 \ \wedge \ |L_1| = i]\}
\end{array}
$$

Representation predicate for list segments

From the definition of Mlist:

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

to that of MlistSeg:

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

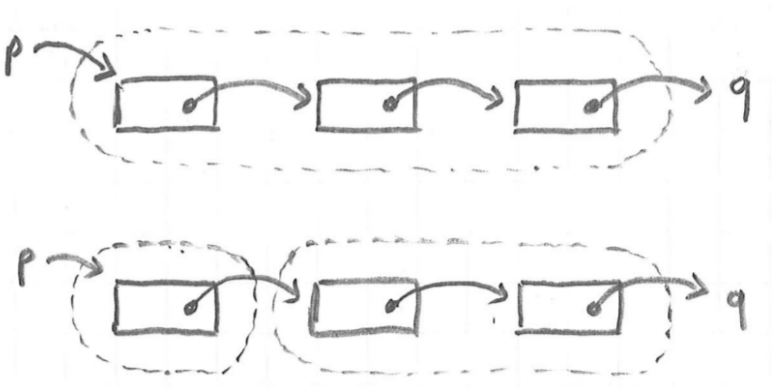
Lists as null-terminated list segments:

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

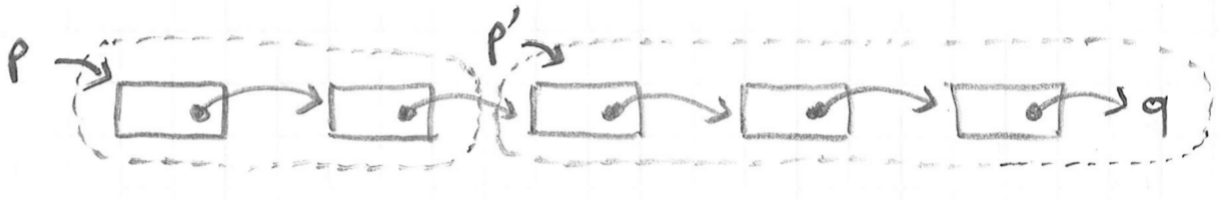
Focus rules for segments

For cons at head, defocus rule:

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



Focus and defocus rules for segments, continued



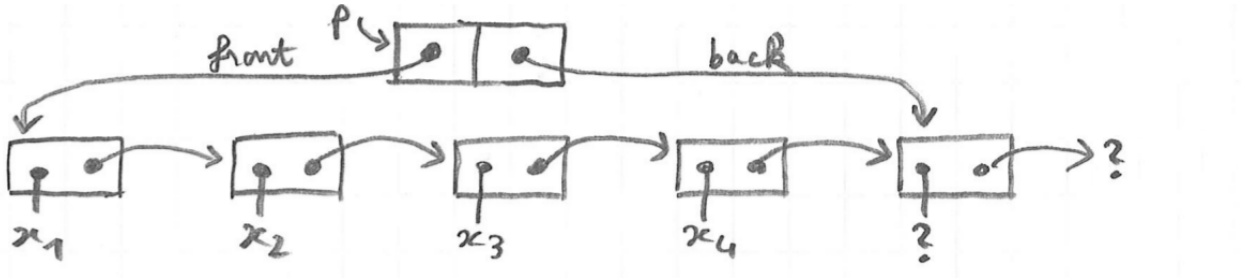
For glueing list segments, defocus rule:

$$
\begin{array}{ll}
 & p \rightsquigarrow \mathsf{MlistSeg}\, p'\, L_1 \;*\; p' \rightsquigarrow \mathsf{MlistSeg}\, q\, L_2 \\
\rhd & p \rightsquigarrow \mathsf{MlistSeg}\, q\, (L_1 +\!\!+ L_2)
\end{array}
$$

For splitting list segments, focus rule:

$$
\begin{array}{ll}
 & p \rightsquigarrow \mathsf{MlistSeg}\, q\, L \;*\; [L = L_1 +\!\!+ L_2] \\
\rhd & \exists p'.\; p \rightsquigarrow \mathsf{MlistSeg}\, p'\, L_1 \;*\; p' \rightsquigarrow \mathsf{MlistSeg}\, q\, L_2
\end{array}
$$

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?

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

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:

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

Interface specification for mutable queues

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

$\{q \rightsquigarrow \mathsf{Queue}\,L\}\ (\texttt{is_empty q})\ \{\lambda b.\ [b = \mathsf{true} \Leftrightarrow L = \mathsf{nil}] * q \rightsquigarrow \mathsf{Queue}\,L\}$

$\{q \rightsquigarrow \mathsf{Queue}\,L\}\ (\texttt{push x q})\ \{\lambda tt.\ q \rightsquigarrow \mathsf{Queue}\,(x :: L)\}$

$\{q \rightsquigarrow \mathsf{Queue}\,L * [L \neq \mathsf{nil}]\}\ (\texttt{pop q})\ \{\lambda x.\ \exists L'.\,[L = L' \& x] * q \rightsquigarrow \mathsf{Queue}\,L'\}$

$\{q \rightsquigarrow \mathsf{Queue}\,(L \& x)\}\ (\texttt{pop q})\ \{\lambda r.\ [r = x] * q \rightsquigarrow \mathsf{Queue}\,L\}$

$$\begin{array}{l}\{q_1 \rightsquigarrow \mathsf{Queue}\,L_1\ *\ q_2 \rightsquigarrow \mathsf{Queue}\,L_2\}\\ (\texttt{transfer q1 q2})\\ \{\lambda tt.\ q_1 \rightsquigarrow \mathsf{Queue}\,\mathsf{nil}\ *\ q_2 \rightsquigarrow \mathsf{Queue}\,(L_1 +\!\!+ L_2)\}\end{array}$$

where $L \& x = L +\!\!+ x :: \mathsf{nil}$.

Summary

- $p \rightsquigarrow \mathsf{MlistSeg}\, q\, L$
- $p \rightsquigarrow \mathsf{Mlist}\, L \;\equiv\; p \rightsquigarrow \mathsf{MlistSeg}\, \mathsf{null}\, q$
- focus/defocus rules for splitting/glueing segments

Content

Mutable lists

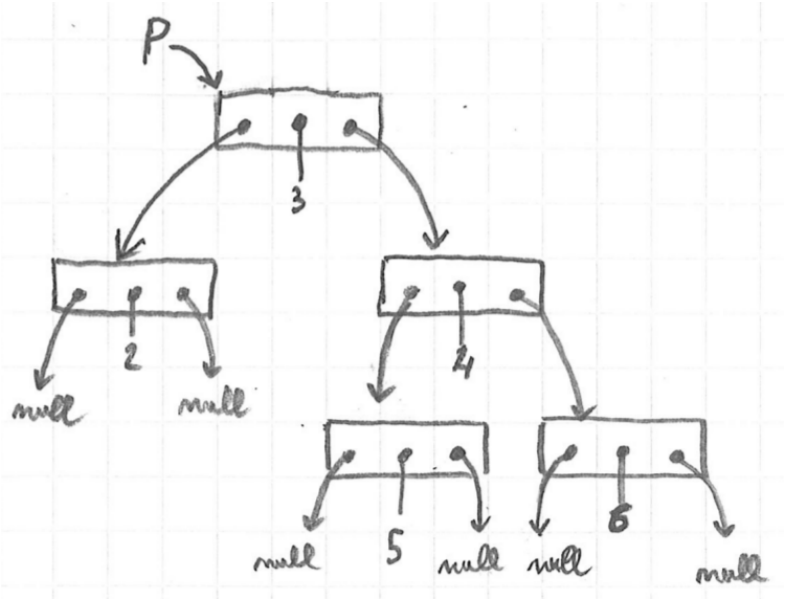
List segments

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

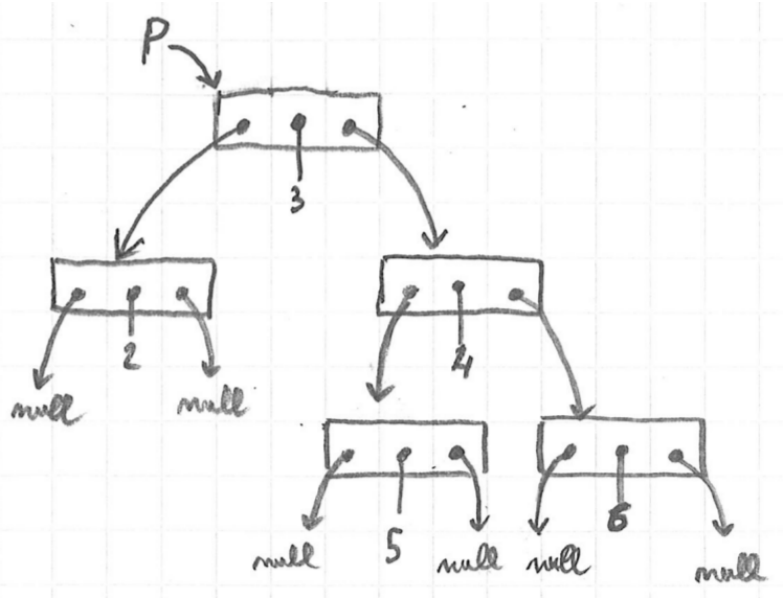


Representation of binary trees, with null for empty trees:

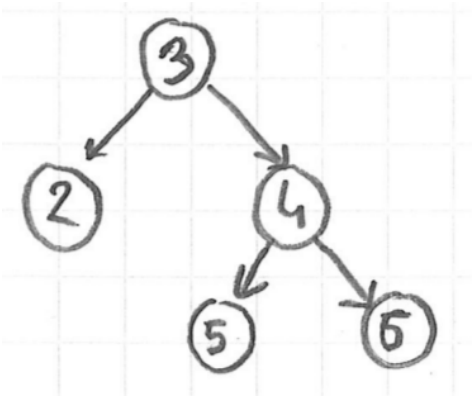
```
type node = {
  mutable item : int;
  mutable left : node;
  mutable right : node; }
```

Representation of a binary tree

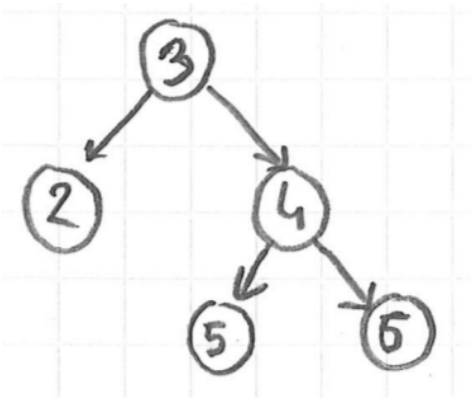
$p \rightsquigarrow \mathsf{Mtree}\, T$



T



Representation of pure trees



Type of pure binary trees in Coq:

```
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))
```

Copy of a binary tree

Construction of a copy of a tree:

```
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?

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

Binary trees: specification

Recall the definition:

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

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

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

Binary trees: focus and defocus

For nonempty trees:

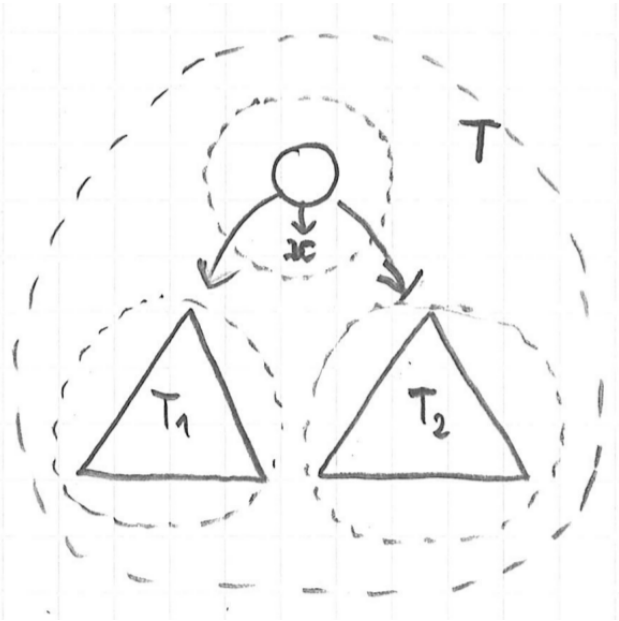
$$p \rightsquigarrow \mathsf{Mtree}\, T * [p \neq \mathsf{null}] \;\; \rhd \;\; \exists x\, T_1\, T_2\, p_1\, p_2.\; \ldots$$

$$p \mapsto \{\!|\mathsf{item}{=}x;\ \mathsf{left}{=}p_1;\ \mathsf{right}{=}p_2|\!\} * p_1 \rightsquigarrow \mathsf{Mtree}\, T_1 * p_2 \rightsquigarrow \mathsf{Mtree}\, T_2$$
$$\rhd \;\; p \rightsquigarrow \mathsf{Mtree}\,(\mathsf{Node}\, x\, T_1\, T_2)$$

For empty trees:

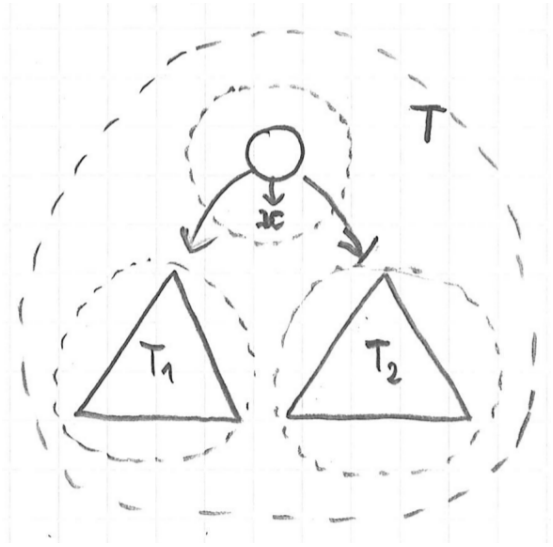
$$\mathsf{null} \rightsquigarrow \mathsf{Mtree}\, T \;\; \rhd \;\; [T = \mathsf{Leaf}]$$

$$[\,] \;\; \rhd \;\; \mathsf{null} \rightsquigarrow \mathsf{Mtree}\, \mathsf{Leaf}$$



Verification of tree copy

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

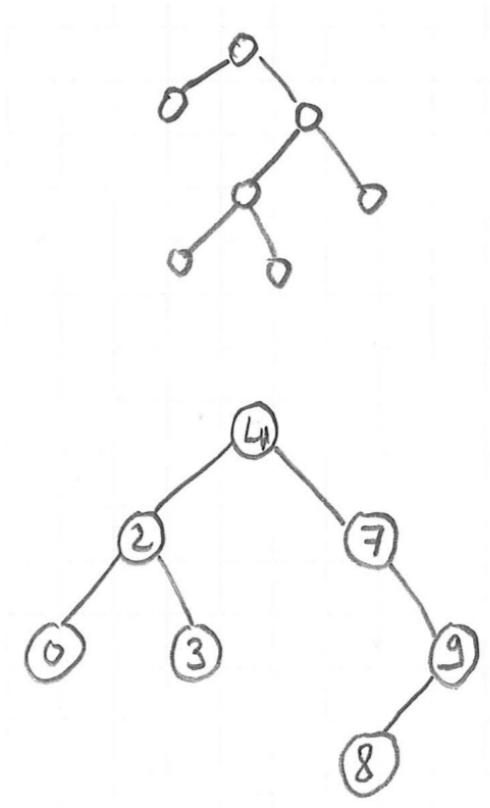


Verification of the copy function:

$$p \mapsto \{|\mathsf{item}{=}x;\ \mathsf{left}{=}p_1;\ \mathsf{right}{=}p_2|\}\ *\ p_1 \rightsquigarrow \mathsf{Mtree}\,T_1\ *\ p_2 \rightsquigarrow \mathsf{Mtree}\,T_2$$
$$p' \mapsto \{|\mathsf{item}{=}x;\ \mathsf{left}{=}p'_1;\ \mathsf{right}{=}p'_2|\}\ *\ p'_1 \rightsquigarrow \mathsf{Mtree}\,T_1\ *\ p'_2 \rightsquigarrow \mathsf{Mtree}\,T_2$$

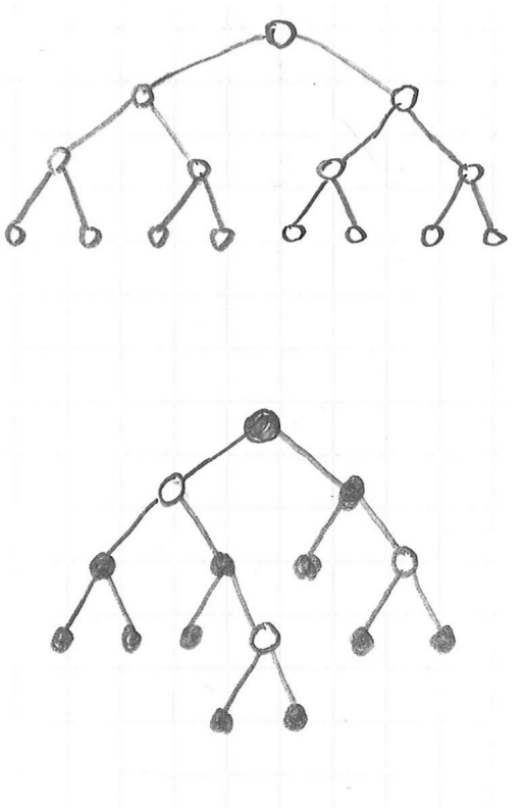
Invariant on trees

tree with 0 or 2 children



binary search tree

complete binary tree



red-black tree

Invariant on binary trees: zero or two children

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

$$
\begin{array}{rcl}
p \rightsquigarrow \mathsf{Mtree2}\, T & \equiv & \mathsf{match}\, T\, \mathsf{with} \\
& & \quad \mid \mathsf{Leaf} \Rightarrow [p = \mathsf{null}] \\
& & \quad \mid \mathsf{Node}\, x\, T_1\, T_2 \Rightarrow \exists p_1 p_2. \\
& & \qquad\qquad p \mapsto \{\!|\mathsf{item}{=}x;\ \mathsf{left}{=}p_1;\ \mathsf{right}{=}p_2|\!\} \\
& & \qquad\qquad * \, p_1 \rightsquigarrow \mathsf{Mtree2}\, T_1 \\
& & \qquad\qquad * \, p_2 \rightsquigarrow \mathsf{Mtree2}\, T_2 \\
& & \qquad\qquad * \, [p_1 = \mathsf{null} \Leftrightarrow p_2 = \mathsf{null}]
\end{array}
$$

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

Problem with modified representation predicate

Specification of copy:

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

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

$$\{p \rightsquigarrow \mathsf{Mtree2}\, T\}\ (\texttt{copy}\ \ \texttt{p})\ \{\lambda p'.\ p \rightsquigarrow \mathsf{Mtree2}\, T\ *\ p' \rightsquigarrow \mathsf{Mtree2}\, T\}$$

Invariants expressed on the pure representation

Better definition, reusing the existing representation predicate:

$$p \rightsquigarrow \mathsf{Mtree2}\, T \quad \equiv \quad p \rightsquigarrow \mathsf{Mtree}\, T \ * \ [\mathsf{nounary}\, T]$$

Inductive properties over pure trees:

```
Inductive nounary : tree → Prop :=
  | nounary_leaf :
      nounary Leaf
  | nounary_node : ∀ x T1 T2,
      nounary T1 →
      nounary T2 →
      (T1 = Leaf ↔ T2 = Leaf) →
      nounary (Node x T1 T2)
```

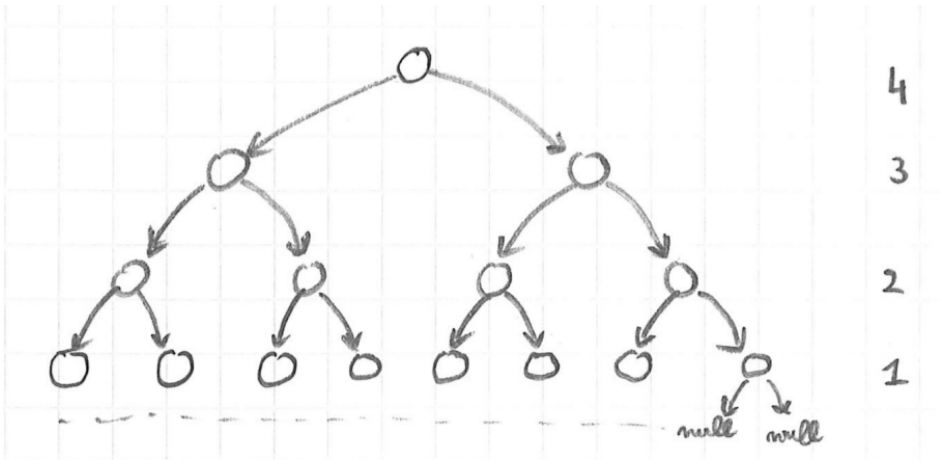
Copying a tree with invariants

Specification of copy:

$$\frac{\{p \rightsquigarrow \mathsf{Mtree}\,T\}\ (\texttt{copy}\ \ \texttt{p})\ \{\lambda p'.\ p \rightsquigarrow \mathsf{Mtree}\,T * p' \rightsquigarrow \mathsf{Mtree}\,T\}}{\begin{array}{l} \{\ \ p \rightsquigarrow \mathsf{Mtree}\,T\ \}\ (\texttt{copy}\ \ \texttt{p})\ \{\lambda p'.\ \ p \rightsquigarrow \mathsf{Mtree}\,T\ \} \\ \quad * \,[\mathsf{nounary}\,T] \qquad\qquad\qquad\qquad\qquad * \,[\mathsf{nounary}\,T] \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad * \,p' \rightsquigarrow \mathsf{Mtree}\,T \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad * \,[\mathsf{nounary}\,T] \end{array}}$$

$$\{p \rightsquigarrow \mathsf{Mtree2}\,T\}\ (\texttt{copy}\ \ \texttt{p})\ \{\lambda p'.\ p \rightsquigarrow \mathsf{Mtree2}\,T * p' \rightsquigarrow \mathsf{Mtree2}\,T\}$$

Complete binary trees



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

$$p \rightsquigarrow \mathsf{MtreeComplete}\, T \quad \equiv \quad p \rightsquigarrow \mathsf{Mtree}\, T \ * \ [\mathsf{complete}\, T]$$

Complete binary trees

Tree completeness as an inductive property over pure trees:

```
Definition complete T :=
  ∃ n, 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).
```

Exposing invariants in representation predicates

Definition from previous slide:

$$p \rightsquigarrow \mathsf{MtreeComplete}\, T \quad \equiv \quad p \rightsquigarrow \mathsf{Mtree}\, T \;*\; [\exists n.\, \mathsf{depth}\, n\, T]$$

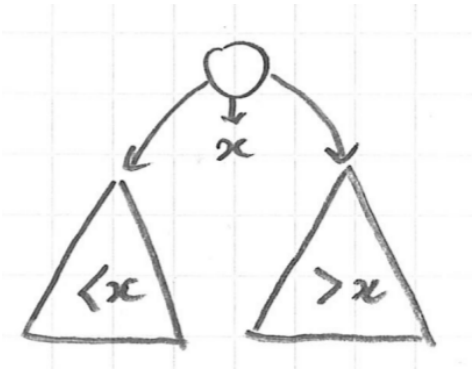
Another useful definition:

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

Example of a "smart constructor":

$$\begin{aligned}\forall x p_1 p_2 T_1 T_2 n.\ & \{p_1 \rightsquigarrow \mathsf{MtreeDepth}\, n\, T_1 \;*\; p_2 \rightsquigarrow \mathsf{MtreeDepth}\, n\, T_2\} \\ & (\texttt{\{ item = x; left = p1; right = p2 \}}) \\ & \{\lambda p.\ p \rightsquigarrow \mathsf{MtreeDepth}\, (n+1)\, (\mathsf{Node}\, x\, T_1\, T_2)\}\end{aligned}$$

Binary search trees



Exercise: define "$p \rightsquigarrow \mathsf{Msearchtree}\, E$" to describe a binary search tree that represents the set E.

$$p \rightsquigarrow \mathsf{Msearchtree}\, E \quad \equiv \quad \exists T.\ p \rightsquigarrow \mathsf{Mtree}\, T \ * \ [\mathsf{search}\, T\, E]$$

Binary search trees

Characterization of purely-functional search trees:

```
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).
```

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 :=
  match T with
  | Leaf ⇒ Black
  | Node c x T1 T2 ⇒ c
  end
```

Representation predicate:

$$p \rightsquigarrow \mathsf{Mrbtree}\, E \;\equiv\; \exists T.\; p \rightsquigarrow \mathsf{Mtree}\, T \;*\; \left[\begin{array}{l} \mathsf{search}\, T\, E \\ \wedge\; \mathsf{color}\, T = \mathsf{Black} \\ \wedge\; \exists n.\; \mathsf{rbtree}\, n\, T \end{array}\right]$$

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

"$p \rightsquigarrow$ Mrbtree E" is an instance of a mutable set "$p \rightsquigarrow$ Mset E".

Specify the functions from the OCaml interface below:

```
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 $p \rightsquigarrow$ Mset E.

Interface specification for mutable sets

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

$\{p \rightsquigarrow \mathsf{Mset}\,E\}\ (\texttt{is_empty p})\ \{\lambda b.\ [b = \mathsf{true} \Leftrightarrow E = \varnothing] * p \rightsquigarrow \mathsf{Mset}\,E\}$

$\{p \rightsquigarrow \mathsf{Mset}\,E\}\ (\texttt{mem x p})\ \{\lambda b.\ [b = \mathsf{true} \Leftrightarrow x \in E] * p \rightsquigarrow \mathsf{Mset}\,E\}$

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

$\{p \rightsquigarrow \mathsf{Mset}\,E\}\ (\texttt{rem x p})\ \{\lambda tt.\ p \rightsquigarrow \mathsf{Mset}\,(E \setminus \{x\})\}$

Summary

Specification of mutable sets in terms of pure sets:

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

Implementation as mutable binary trees with pure invariants:

$$\begin{aligned} p \rightsquigarrow \mathsf{Mrbtree}\, E \quad \equiv \quad \exists T.\ p \rightsquigarrow \mathsf{Mtree}\, T \ *\ [\ & \mathsf{search}\, T\, E \\ & \wedge\ \mathsf{color}\, T = \mathsf{Black} \\ & \wedge\ \exists n.\ \mathsf{rbtree}\, n\, T \ ] \end{aligned}$$

Separation Logic specification of binary trees:

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

Content

Mutable lists

List segments

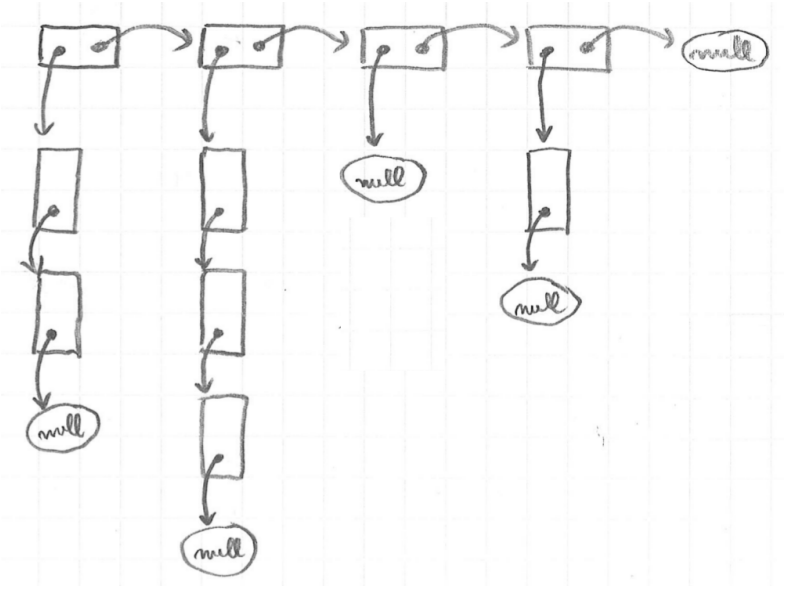
Trees

Polymorphic containers

- Mutable lists of mutable lists
- Polymorphic list representation predicate
- Application to arrays, queues, ...
- Application to records
- Examples with binary trees

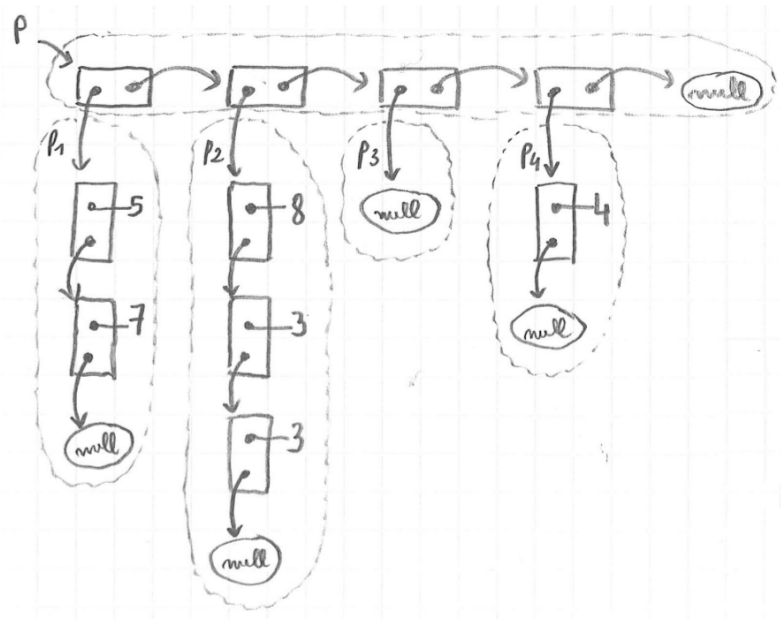
Mutable lists of mutable lists

The goal is to specify a mutable lists of mutable lists:



Representation as a list of list

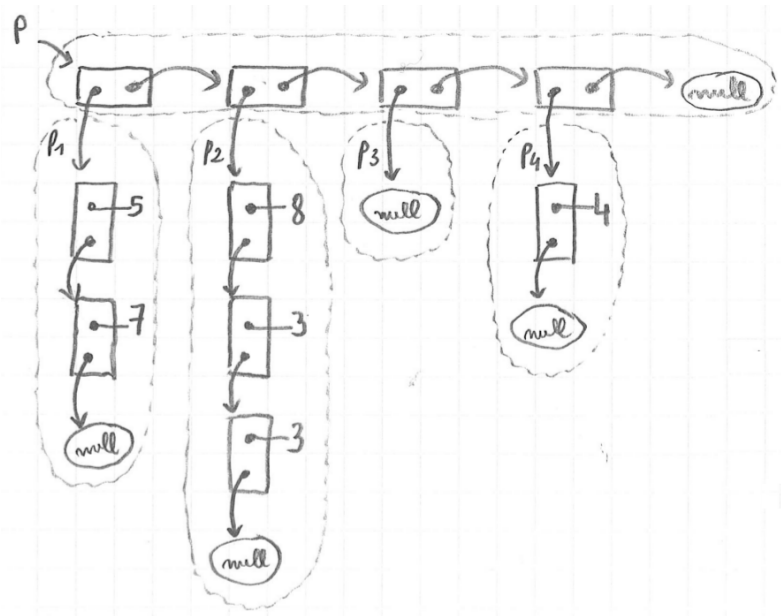
Specify it with respect to a pure list of pure lists:



$p \rightsquigarrow$ MlistOfMlist $((5 :: 7 :: \text{nil}) :: (8 :: 3 :: 3 :: \text{nil}) :: (\text{nil}) :: (4 :: \text{nil}) :: \text{nil})$

Mutable lists of mutable lists

A first attempt:



$$p \rightsquigarrow \mathsf{MlistOfMlist}\, K \;\equiv\; \exists L.\; p \rightsquigarrow \mathsf{Mlist}\, L \;*\; \mathop{\Big\circledast}_{p_i \text{ at index } i \text{ in} L} p_i \rightsquigarrow \mathsf{Mlist}\,(K[i])$$

Problem: involves reasoning about indices.

Representation predicate for lists of lists

Representation of lists:

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

Generalization to lists of lists:

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

Example: flatten

For pure lists, we have:

```
flatten : ('a list) list -> 'a list
```

Specification of the equivalent function for mutable lists:

$$\{p \rightsquigarrow \mathsf{MlistOfMlist}\, L\}\ (\texttt{mflatten p})\ \{\lambda q.\ q \rightsquigarrow \mathsf{Mlist}\,(\mathsf{flatten}\, L)\}$$

Generalization to polymorphic lists

Representation of lists of lists:

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

Representation of lists of anything:

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

$$
p \rightsquigarrow \mathsf{MlistOfMlist}\, K \quad \equiv \quad p \rightsquigarrow \mathsf{Mlistof}\, \mathsf{Mlist}\, K
$$

Polymorphic list representation predicate

Polymorphic representation predicate for lists

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

Remark:

$$p \rightsquigarrow \mathsf{Mlist}\, L \quad \equiv \quad p \rightsquigarrow \mathsf{Mlistof}\, \mathsf{Id}\, L$$

where $x \rightsquigarrow \mathsf{Id}\, X$ is defined as $[x = X]$.

Checking the types

$$p \rightsquigarrow \mathsf{Mlistof}\, R\, L$$

- 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: $x \rightsquigarrow R\, X$

Implementation of the arrow notation:

$$\begin{aligned} p \rightsquigarrow Q &\equiv Q\, p \\ p \rightsquigarrow \mathsf{Mlistof}\, R\, L &\equiv \mathsf{Mlistof}\, R\, L\, p \end{aligned}$$

```
Mlistof (a:Type) (A:Type) (R:A→a→Hprop) (L:list A) (p:loc)
  : Hprop
```

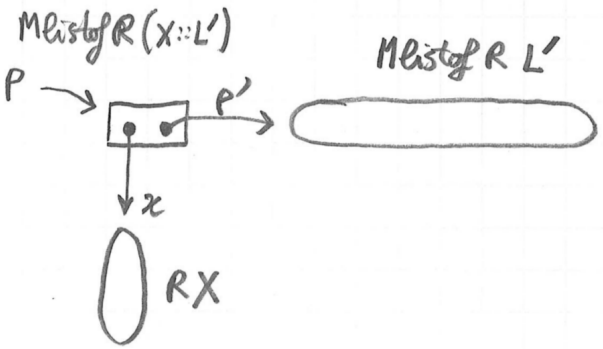
Application to arrays, queues, ...

An array is represented using a finite map M:

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

array of arrays:	$p \rightsquigarrow \mathsf{Arrayof}\,(\mathsf{Arrayof}\,\mathsf{Id})K$
array of lists:	$p \rightsquigarrow \mathsf{Arrayof}\,(\mathsf{Mlistof}\,\mathsf{Id})K$
queue of arrays:	$p \rightsquigarrow \mathsf{Queueof}\,(\mathsf{Arrayof}\,\mathsf{Id})K$
list of trees:	$p \rightsquigarrow \mathsf{Mlistof}\,\mathsf{Mtree}\,K$
array of arrays of trees:	$p \rightsquigarrow \mathsf{Arrayof}\,(\mathsf{Arrayof}\,\mathsf{Mtree})K$

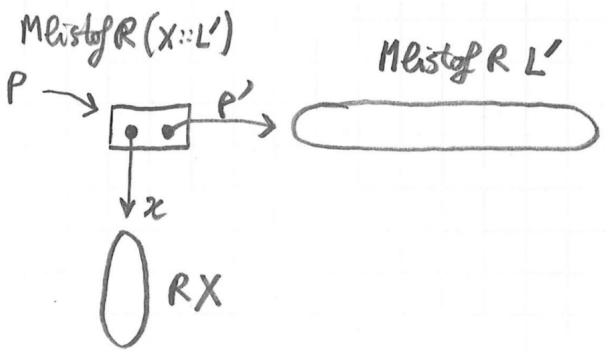
Representation predicate for records



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

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

Representation predicate for records



Representation predicate for list cells:

$$p \rightsquigarrow \mathsf{Cellof}\, R_h\, H\, R_t\, T \quad \equiv \quad \exists ht. \quad \begin{array}{l} p \mapsto \{|\mathsf{hd}{=}h;\ \mathsf{tl}{=}t|\} \\ *\ x \rightsquigarrow R_h\, H \\ *\ y \rightsquigarrow R_t\, T \end{array}$$

Application to binary trees

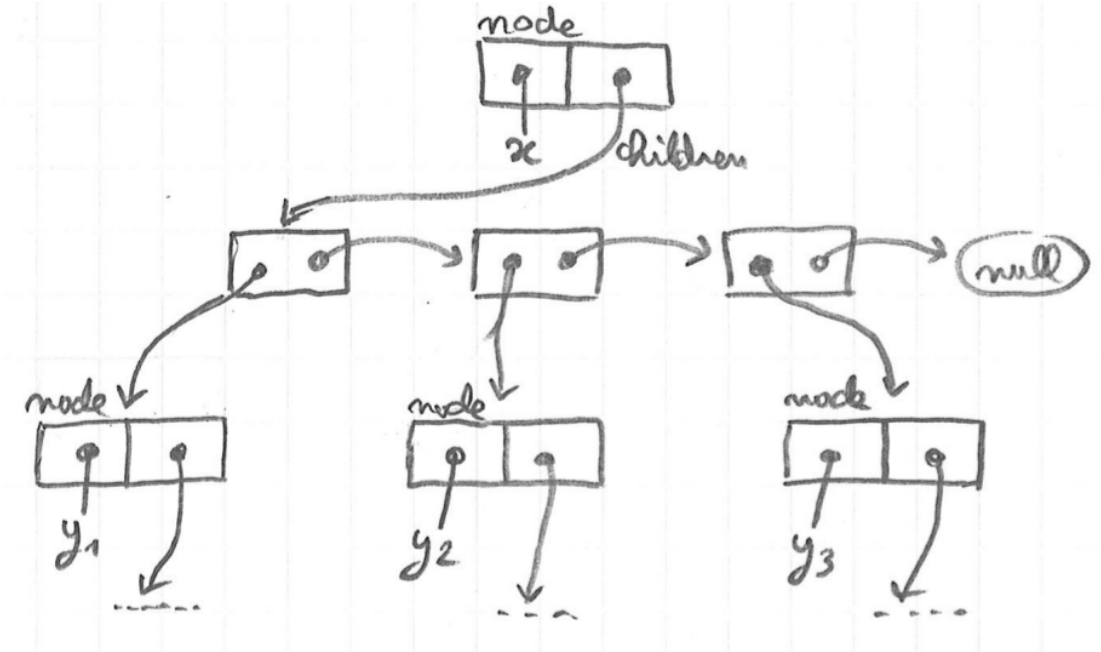
Example: definition of Mtree (for integer leaves, without holes) using Nodeof.

$$
\begin{array}{lcl}
p \rightsquigarrow \mathsf{Mtree}\, T & \equiv & \mathsf{match}\, T\, \mathsf{with} \\
& & \quad \mid \mathsf{Leaf} \;\Rightarrow\; [p = \mathsf{null}] \\
& & \quad \mid \mathsf{Node}\, x\, T_1\, T_2 \;\Rightarrow\; p \rightsquigarrow \mathsf{Nodeof}\, \mathsf{Id}\, x\, \mathsf{Mtree}\, T_1\, \mathsf{Mtree}\, T_2
\end{array}
$$

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:

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

Exercise: define Mtree, using Nodeof.

Representation predicate:

$$
\begin{aligned}
p \rightsquigarrow \mathsf{Mtree}\, T \quad \equiv \quad & \mathsf{match}\, T\, \mathsf{with} \\
& \mid \mathsf{Leaf} \Rightarrow [p = \mathsf{null}] \\
& \mid \mathsf{Node}\, x\, C \Rightarrow p \rightsquigarrow \mathsf{Nodeof}\, \mathsf{Id}\, x\, (\mathsf{Mlistof}\, \mathsf{Mtree})\, C
\end{aligned}
$$