Separation Logic 1/4

Jean-Marie Madiot

Inria Paris

January 20, 2021

slides (mostly) from Arthur Charguéraud

Separation Logic

Separation Logic:
a set of rules for reasoning about programs that mutate the heap.

Separation Logic

Separation Logic:
a set of rules for reasoning about programs that mutate the heap.

Separation Logic with interactive proofs:
a successful approach to the verification of imperative programs.

Where does it come from?

- John Reynolds (2000)
  - Intuitionistic Reasoning about Shared Mutable Data Structure
  - —building on ideas from Burstall (1972).
- John Reynolds, Peter O'Hearn, Hongseok Yang (2001)
  - Local reasoning about programs that alter data structures
- John Reynolds (2002)
  - Separation Logic: A logic for shared mutable data structure.

How successful?

Micro-controller	Klein et al	NICTA	Isabelle
Assembly language	Chlipala et al	MIT	Coq
Operating system	Shao et al	Yale	Coq
C (drivers)	Yang et al	Oxford	Other
C-light (concurrent)	Appel et al	Princeton	Coq
C11 (concurrent)	Vafeiadis et al	MPI and MSR	Paper
Java	Parkinson et al	MSR and Cambridge	Other
Java	Jacobs et al	Leuven	Verifast
Javascript	Gardner et al	Imperial College	Paper
ML	Morisset et al	Harvard	Coq
OCaml	Charguéraud	Inria	Coq
SML	Myreen et al	U. of Cambridge	HOL
Multicore OCaml	Mével et al	Inria	Coq-Iris

and many more...

Why successful?

- Specifications that one can read: concise and intuitive.
- Proofs that one can understand: step-by-step analysis.
- Modularity that enable scaling up: $>$ 10k loc.

Why successful?

- Specifications that one can read: concise and intuitive.
- Proofs that one can understand: step-by-step analysis.
- Modularity that enable scaling up: $>$ 10k loc.

Separation Logic $+$ interactive proofs $=$ very expressive

Purpose of the course

Goal: make *you* an expert in Separation Logic for sequential programs.

You will learn how Separation Logic handles:

- Tree-shaped structures
- Structures with sharing
- Polymorphic functions
- Higher-order functions
- Abstraction
- Modularity
- Interactive proofs

Choice of the logic

We will *define* Separation Logic in terms of a standard higher-order logic.

Our definitions will be given in Coq, because:

1. most formalizations of SL are in Coq;
2. my own work belong to this majority.

Choice of the source language

Separation Logic can be adapted to pretty much any language.

To best present all its aspects, we need a language with:

- a clean and concise syntax,
- null pointers,
- pointer arithmetic,
- types and polymorphism,
- higher-order functions.

Choice of the source language

Separation Logic can be adapted to pretty much any language.

To best present all its aspects, we need a language with:

- a clean and concise syntax,
- null pointers,
- pointer arithmetic,
- types and polymorphism,
- higher-order functions.

We will use a virtual language: OCaml extended with null pointers and pointers arithmetic; we ignore the existence of headers used by the GC.

(see Coq definitions)

Chapter 1

Separation Logic Operators

Heap predicates

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

A heap predicate H characterizes memory stores of a particular shape.

Heap predicates

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

A heap predicate H characterizes memory stores of a particular shape.

A heap predicate has type "$\texttt{heap} \rightarrow \texttt{Prop}$", i.e. "$H\,m$" is a proposition.

Heap predicates

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

A heap predicate H characterizes memory stores of a particular shape.

A heap predicate has type "$\texttt{heap} \rightarrow \texttt{Prop}$", i.e. "$H\,m$" is a proposition.

Primitive heap predicates:

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

Empty heap and pure facts

Definition:

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

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

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

Empty heap and pure facts

Definition:

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

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

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

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

Singleton heap

Definition:

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

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

Before: $[\,]$

After: $r \mapsto 3$

Singleton heap

Definition:

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

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

Before: $[\,]$

After: $r \mapsto 3$

Example: specification of "`incr s`".

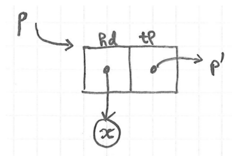
Before:	$s \mapsto n$	for some n
After:	$s \mapsto (n+1)$	

Record fields

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

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

Example:



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

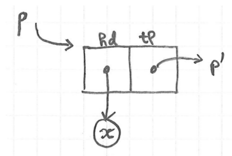
$$p.\mathsf{tl} \mapsto p'$$

Record fields

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

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

Example:



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

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

In the C memory model:

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

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

Separating conjunction

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

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

Separating conjunction

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

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

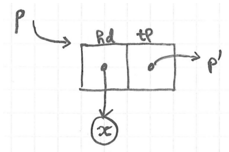
Definition:

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

where:

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

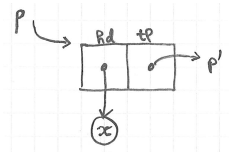
Representation of list cells



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

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

Representation of list cells



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

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

Remark: the new arrow symbol will be overloaded later.

Separating conjunction: aliasing

Exercise:

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

Separating conjunction: aliasing

Exercise:

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

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

Separating conjunction: aliasing

Exercise:

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

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

Correct answer:

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

Existential quantification

Definition:

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

Compare:

$$(\exists x.\, P) \quad : \quad \texttt{Prop} \qquad\qquad \text{when} \quad (P : \texttt{Prop})$$

$$(\exists x.\, H) \quad : \quad \texttt{heap} \to \texttt{Prop} \qquad \text{when} \quad (H : \texttt{heap} \to \texttt{Prop})$$

Summary

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

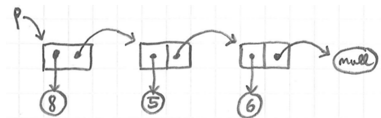
See coq definitions

Chapter 2

Representation Predicate for Lists

Implementation of mutable lists

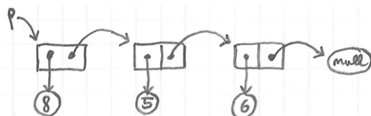
Mutable lists (C-style), expressed in OCaml extended with null pointers.




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

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

Representation of mutable lists

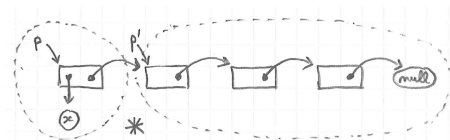


$$L \;=\; 8 :: 5 :: 6 :: \mathsf{nil}$$

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

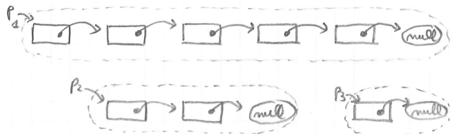
Remark: in Coq, $p \rightsquigarrow \mathsf{MList}\, L$ is just a convenient notation for $\mathsf{MList}\, L\, p$.

Representation predicate



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

Separation properties



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

Separation enforces: no cycles, and no sharing.

Union heap predicate

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

Equivalent to:

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

where:

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

List construction

```
let rec build n v =
  if n = 0 then null else
    let p' = build (n-1) v in
    { hd = v; tl = p' }
```

List construction

```
let rec build n v =
  if n = 0 then null else
    let p' = build (n-1) v in
    { hd = v; tl = p' }
```

Pre-condition:

$$[n \geqslant 0]$$

Post-condition, where p denotes the result:

$$\exists L.\ p \rightsquigarrow \mathsf{MList}\, L \star [\mathsf{length}\, L = n \ \wedge\ (\forall i.\, 0 \leqslant i < n \Rightarrow L[i] = v)]$$

List construction: proof (1/2)

$$\exists L.\ p \rightsquigarrow \mathsf{MList}\, L \star [\mathsf{length}\, L = n \;\;\wedge\;\; (\forall i.\, 0 \leqslant i < n \Rightarrow L[i] = v)]$$

Case $n = 0$. We have $p = \mathsf{null}$. We take $L = \mathsf{nil}$.

To produce $p \rightsquigarrow \mathsf{MList}\, L$, we need to produce $\mathsf{null} \rightsquigarrow \mathsf{MList}\, \mathsf{nil}$.

List construction: proof (1/2)

$$\exists L.\ p \rightsquigarrow \mathsf{MList}\, L \star [\mathsf{length}\, L = n \;\wedge\; (\forall i.\, 0 \leqslant i < n \Rightarrow L[i] = v)]$$

Case $n = 0$. We have $p = \mathsf{null}$. We take $L = \mathsf{nil}$.

To produce $p \rightsquigarrow \mathsf{MList}\, L$, we need to produce $\mathsf{null} \rightsquigarrow \mathsf{MList}\,\mathsf{nil}$. We use:

$$(\mathsf{null} \rightsquigarrow \mathsf{MList}\,\mathsf{nil}) = [\,]$$

List construction: proof (1/2)

$$\exists L.\ p \rightsquigarrow \mathsf{MList}\,L \star [\mathsf{length}\,L = n \ \wedge\ (\forall i.\ 0 \leqslant i < n \Rightarrow L[i] = v)]$$

Case $n = 0$. We have $p = \mathsf{null}$. We take $L = \mathsf{nil}$.

To produce $p \rightsquigarrow \mathsf{MList}\,L$, we need to produce $\mathsf{null} \rightsquigarrow \mathsf{MList}\,\mathsf{nil}$. We use:

$$(\mathsf{null} \rightsquigarrow \mathsf{MList}\,\mathsf{nil}) \;=\; [\,]$$

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

List construction: proof (2/2)

$$\exists L.\ p \rightsquigarrow \mathsf{MList}\, L \star [\mathsf{length}\, L = n \ \wedge \ (\forall i.\, 0 \leqslant i < n \Rightarrow L[i] = v)]$$

Case $n > 0$. By IH, we have: $p' \rightsquigarrow \mathsf{MList}\, L'$, with L' of length $n - 1$.

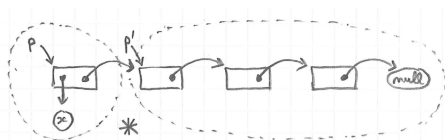
To produce $p \rightsquigarrow \mathsf{MList}\, L$, we have $p' \rightsquigarrow \mathsf{MList}\, L'$ and $p \rightsquigarrow \{\!|\mathsf{hd}{=}v;\ \mathsf{tl}{=}p'|\!\}$.

List construction: proof (2/2)

$$\exists L.\ p \rightsquigarrow \mathsf{MList}\, L \star [\mathsf{length}\, L = n \ \wedge\ (\forall i.\, 0 \leqslant i < n \Rightarrow L[i] = v)]$$

Case $n > 0$. By IH, we have: $p' \rightsquigarrow \mathsf{MList}\, L'$, with L' of length $n - 1$.

To produce $p \rightsquigarrow \mathsf{MList}\, L$, we have $p' \rightsquigarrow \mathsf{MList}\, L'$ and $p \rightsquigarrow \{\!|\mathsf{hd}{=}v;\ \mathsf{tl}{=}p'|\!\}$.



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

In-place list reversal: code

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

In-place list reversal: code

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

Exercise:

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

In-place list reversal: invariants

Before the loop:

In-place list reversal: invariants

Before the loop:

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

In-place list reversal: invariants

Before the loop:

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

After the loop:

In-place list reversal: invariants

Before the loop:

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

After the loop:

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

In-place list reversal: invariants

Before the loop:

$$r \mapsto p_0 \star s \mapsto \textsf{null} \star p_0 \rightsquigarrow \textsf{MList}\, L$$

After the loop:

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

Loop invariant:

In-place list reversal: invariants

Before the loop:

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

After the loop:

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

Loop invariant:

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

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

Invariant:

$$\exists p q L_1 L_2.\ r \mapsto p \star s \mapsto q$$
$$\star\, p \rightsquigarrow \mathsf{MList}\, L_2 \star q \rightsquigarrow \mathsf{MList}\, L_1$$
$$\star\, [L = \mathsf{rev}\, L_1 +\!\!+ L_2]$$

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

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

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

Invariant:

$$\exists p q L_1 L_2.\ r \mapsto p \star s \mapsto q \\ \star\ p \rightsquigarrow \mathsf{MList}\, L_2 \star q \rightsquigarrow \mathsf{MList}\, L_1 \\ \star\ [L = \mathsf{rev}\, L_1 \mathbin{+\!\!+} L_2]$$

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

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

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

$$r \mapsto \mathsf{null} \star \mathsf{null} \rightsquigarrow \mathsf{MList}\, L_2 \star s \mapsto q \star q \rightsquigarrow \mathsf{MList}\, L_1 \star [L = \mathsf{rev}\, L_1 \mathbin{+\!\!+} L_2]$$

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

Invariant:

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

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

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

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

$$r \mapsto \mathsf{null} \star \mathsf{null} \rightsquigarrow \mathsf{MList}\, L_2 \star s \mapsto q \star q \rightsquigarrow \mathsf{MList}\, L_1 \star [L = \mathsf{rev}\, L_1 +\!\!+ L_2]$$

Derive $L_2 = \mathsf{nil}$ using:

$$(\mathsf{null} \rightsquigarrow \mathsf{MList}\, L) = [L = \mathsf{nil}]$$

Conversion rule for empty lists

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

Let us prove: $(\mathsf{null} \leadsto \mathsf{MList}\, L) \;=\; [L = \mathsf{nil}]$

– From right to left: we may assume $L = \mathsf{nil}$, thus:

$$[\mathsf{nil} = \mathsf{nil}] \;=\; [\,] \;=\; (\mathsf{null} \leadsto \mathsf{MList}\, \mathsf{nil})$$

Conversion rule for empty lists

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

Let us prove: $(\mathsf{null} \rightsquigarrow \mathsf{MList}\, L) = [L = \mathsf{nil}]$

– From right to left: we may assume $L = \mathsf{nil}$, thus:

$$[\mathsf{nil} = \mathsf{nil}] = [\,] = (\mathsf{null} \rightsquigarrow \mathsf{MList}\,\mathsf{nil})$$

– From left to right: if $L = \mathsf{nil}$, then easy; otherwise $L = x :: L'$ and:

$$\mathsf{null} \rightsquigarrow \mathsf{MList}\,(x :: L') = \left(\exists p'.\ \mathsf{null} \rightsquigarrow \{|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\} \star p' \rightsquigarrow \mathsf{MList}\, L'\right)$$

contradicts the fact that no data can be allocated at the null address.

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

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

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

to

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

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

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

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

to

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

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

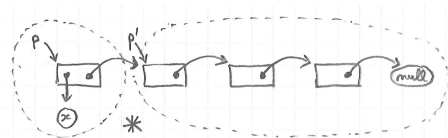
$$p \rightsquigarrow \{\!|\text{hd}{=}x;\ \text{tl}{=}q|\!\} \star q \rightsquigarrow \text{MList}\, L_1 \star p' \rightsquigarrow \text{MList}\, L_2' \star [L = \text{rev}\, L_1 +\!\!+ (x :: L_2')]$$

to

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

Conversion rules for nonempty lists

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



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

Summary

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

Break!

Chapter 3

Representation Predicate for List Segments

Length of a mutable list using a while loop

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

Exercise:

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

Mlength: initial and final states

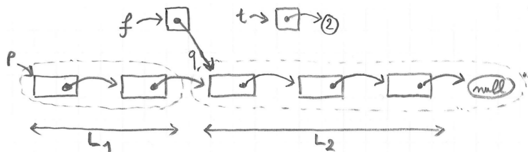
Before the loop:

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

After the loop:

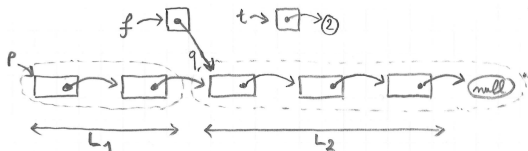
$$(p \rightsquigarrow \mathsf{MList}\, L) \star (f \mapsto \mathsf{null}) \star (t \mapsto \mathsf{length}\, L)$$

Mlength: loop invariant



Loop invariant:

Mlength: loop invariant



Loop invariant:

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

Representation predicate for list segments

$$p \leadsto \mathsf{MList}\, L \;\; \equiv \;\; \mathsf{match}\, L\, \mathsf{with}$$
$$|\, \mathsf{nil} \;\Rightarrow\; [p = \mathsf{null}]$$
$$|\, x :: L' \;\Rightarrow\; \exists p'.\; p \leadsto \{|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\}$$
$$\star\, p' \leadsto \mathsf{MList}\, L'$$

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

$$p \leadsto \mathsf{MlistSeg}\, q\, L \;\; \equiv \;\; \mathsf{match}\, L\, \mathsf{with}$$
$$|\, \mathsf{nil} \;\Rightarrow\; [p = q]$$
$$|\, x :: L' \;\Rightarrow\; \exists p'.\; p \leadsto \{|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\}$$
$$\star\, p' \leadsto \mathsf{MlistSeg}\, q\, L'$$

Representation predicate for list segments

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

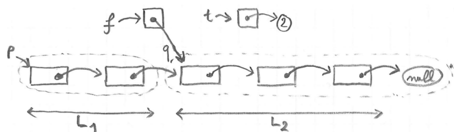
Exercise: generalize MList to define $p \rightsquigarrow \mathsf{MlistSeg}\, q\, L$, where L denotes the list of items in the list segment from p (inclusive) to q (exclusive).

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

Remark:

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

Mlength: proof

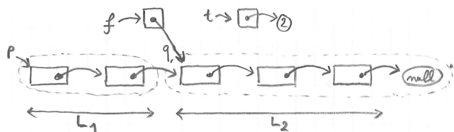


Enter:

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

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

Mlength: proof



Enter:

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

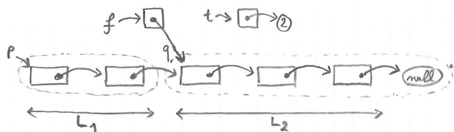
$$[\,] \ = \ (p \leadsto \text{MlistSeg}\, p\, \text{nil})$$

Exit:

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

$$(p \leadsto \text{MlistSeg}\, \text{null}\, L) \ = \ (p \leadsto \text{MList}\, L)$$

Mlength: proof



Enter:

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

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

Exit:

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

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

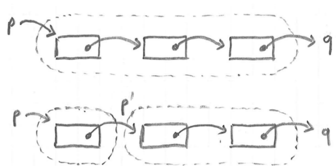
Step:

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

$$\exists q.\ p \rightsquigarrow \mathsf{MlistSeg}\, q\, L_1 \ \star \ q \rightsquigarrow \{|\mathsf{hd}{=}x;\ \mathsf{tl}{=}q'|\}$$

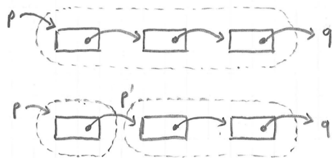
$$= \ \ p \rightsquigarrow \mathsf{MlistSeg}\, q'\, (L_1 +\!\!+\, x :: \mathsf{nil})$$

Splitting rules for list segments



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

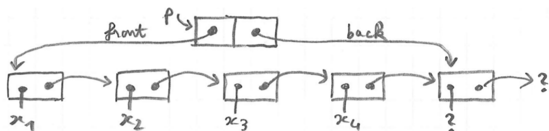
Splitting rules for list segments



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

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

An implementation of mutable queues

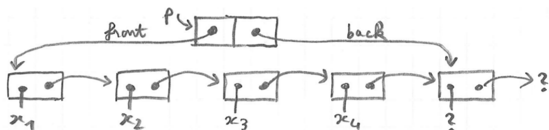


Represent a queue as a list segment, with the last cell storing no item.

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

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

An implementation of mutable queues



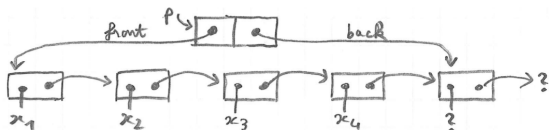
Represent a queue as a list segment, with the last cell storing no item.

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

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

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

An implementation of mutable queues



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

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

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

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

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

Summary

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

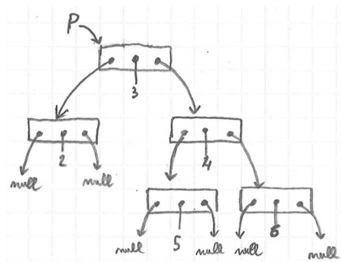
Split and merge of segments:

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

Chapter 4

Representation Predicate for Trees

Implementation of a mutable binary trees



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

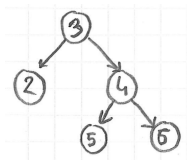
```
type node = {
  mutable item : int;
  mutable left : node;
  mutable right : node; }
```

Logical binary trees

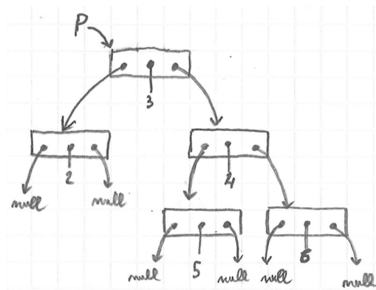
```
Inductive tree : Type :=
  | Leaf : tree
  | Node : int → tree → tree → tree.
```

Example:

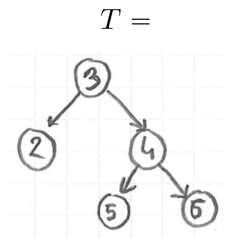
```
Node 3
     (Node 2 Leaf Leaf)
     (Node 4 (Node 5 Leaf Leaf)
             (Node 6 Leaf Leaf))
```



Representation predicate for binary trees



$T =$



Representation predicate:

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

Representation predicate for binary trees

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

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

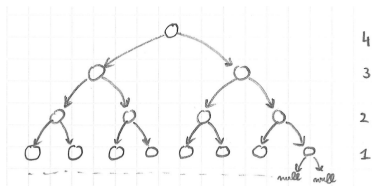
Representation predicate for binary trees

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

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

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

Complete binary tree



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

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

Complete binary tree (1/2)

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

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

Complete binary tree (1/2), solution

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

Complete binary tree (1/2), solution

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

Or:

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

Complete binary tree (2/2)

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

Complete binary tree (2/2)

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

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

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

Complete binary tree of unspecified depth

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

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

Complete binary tree of unspecified depth

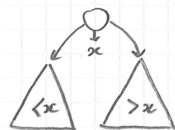
$$p \rightsquigarrow \mathsf{MtreeDepth}\, n\, T \;\equiv\; (p \rightsquigarrow \mathsf{Mtree}\, T) \star [\mathsf{depth}\, n\, T]$$

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

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

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

Binary search tree property



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

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

Binary search tree predicate

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

Binary search tree predicate

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

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

Binary search tree predicate

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

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

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

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

Summary

Common representation predicate for all binary trees:

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

Invariants are expressed on the pure trees:

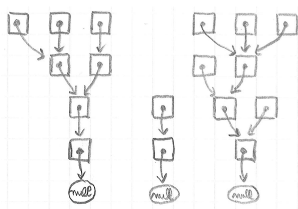
$$
p \rightsquigarrow \mathsf{MsearchTree}\, E \;\equiv\; \exists T.\; p \rightsquigarrow \mathsf{Mtree}\, T \star [\mathsf{search}\, T\, E]
$$

Operations are specified in terms of the model. For example, `add x p` changes $p \rightsquigarrow \mathsf{MsearchTree}\, E$ into $p \rightsquigarrow \mathsf{MsearchTree}\, (E \cup \{x\})$.

Chapter 5

Structures with sharing

The union-find data structure



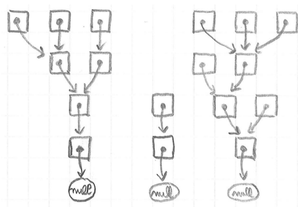

```
type node = node ref
```

Implements an equivalence relation S of type: loc $\rightarrow$ loc $\rightarrow$ `Prop`.

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

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

Representation of union-find cells

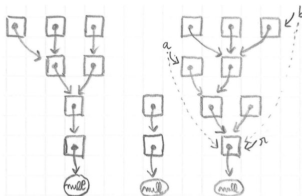


$$(p_1 \mapsto q_1) \star (p_2 \mapsto q_2) \star \ldots \star (p_n \mapsto q_n)$$

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

where G is a finite map from locations to locations.

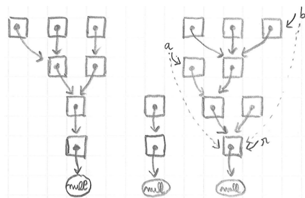
Invariants of union-find



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

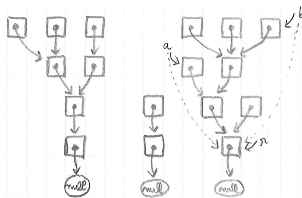
```
Inductive root : fmap loc loc → loc → loc → Prop :=
  | root_init : ∀G x,
      binds G x null →
      root G x x
  | root_step : ∀G x y r,
      binds G x y →
      y ≠null →
      root G y r →
      root G x r.
```

Specification of the union-find structure



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

Specification of the union-find structure



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

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

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

Summary

Iterated separating conjunction, written $\circledast$.

For Union-Find:

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

Chapter 6

Separation Logic Triples

Separation Logic triples

A term t is specified using a Separation Logic triple of the form:

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

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

Separation Logic triples

A term t is specified using a Separation Logic triple of the form:

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

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

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

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

Examples of triples

Example 1:

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

Examples of triples

Example 1:

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

Example 2:

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

Examples of triples

Example 1:

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

Example 2:

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

Example 3:

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

Examples of triples

Example 1:

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

Example 2:

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

Example 3:

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

Example 4:

$$\{r \mapsto 3\}\ (\texttt{incr r})\ \{\lambda_.\,(r \mapsto 4)\}$$

Remark: in "$\lambda_.\,(r \mapsto 4)$" we do not care about the return value.

Specification of functions

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

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

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

Example:

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

Specification of operations on memory cells

Exercise: specify the primitive operations on references.

$$\big(\texttt{ref v}\big)$$

$$\big(\texttt{!r}\big)$$

$$\big(\texttt{r := v}\big)$$

Specification of operations on memory cells

Exercise: specify the primitive operations on references.

$$(\texttt{ref v})$$

$$(\texttt{!r})$$

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

Solution:

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

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

Specification of operations on memory cells

Exercise: specify the primitive operations on references.

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

Solution:

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

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

$$\forall r v w. \quad \{r \mapsto w\}\ (\texttt{r := v})\ \{\lambda_.\,(r \mapsto v)\}$$
$$\forall r v. \quad \{\exists w.\ r \mapsto w\}\ (\texttt{r := v})\ \{\lambda_.\,(r \mapsto v)\}$$
$$\forall r v. \quad \{r \mapsto -\}\ (\texttt{r := v})\ \{\lambda_.\,(r \mapsto v)\}$$

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

Specification of partial functions

Presentation 1:

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

Presentation 2:

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

Specification of operations on arrays

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

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

Notation:

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

Specification of operations on arrays

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

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

Specification of operations on arrays

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

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

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

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

Interpretation of triples (1/3)

Assume for now that triples describe the entire state.

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

$$\forall m. \quad H\, m \quad \Rightarrow \quad \exists v.\, \exists m'. \quad \langle t, m\rangle \Downarrow \langle v, m'\rangle \ \wedge\ ([x \to v]\, H')\, m'$$

Interpretation of triples (1/3)

Assume for now that triples describe the entire state.

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

$$\forall m. \quad H\, m \quad \Rightarrow \quad \exists v.\, \exists m'. \quad \langle t, m\rangle \Downarrow \langle v, m'\rangle \ \wedge \ ([x \to v]\, H')\, m'$$

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

Interpretation of triples (1/3)

Assume for now that triples describe the entire state.

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

$$\forall m. \quad H\, m \quad \Rightarrow \quad \exists v.\, \exists m'. \quad \langle t, m \rangle \Downarrow \langle v, m' \rangle \ \wedge\ ([x \to v]\, H')\, m'$$

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

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

$$\forall m. \quad H\, m \quad \Rightarrow \quad \exists v.\, \exists m'. \quad \langle t, m \rangle \Downarrow \langle v, m' \rangle \ \wedge\ Q\, v\, m'$$

Interpretation of triples (2/3)

In Separation Logic, a triple describes only a part m_1 of the heap.
The rest of the heap, call it m_2, is assumed to remain unchanged.

Recall that:

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

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

Interpretation of triples (2/3)

In Separation Logic, a triple describes only a part m_1 of the heap. The rest of the heap, call it m_2, is assumed to remain unchanged.

Recall that:

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

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

$$\forall m_1\, m_2.\; \begin{cases} H\, m_1 \\ m_1 \perp m_2 \end{cases} \;\Rightarrow\; \exists v.\, \exists m_1'.\; \begin{cases} \langle t, m_1 \uplus m_2 \rangle \Downarrow \langle v, m_1' \uplus m_2 \rangle \\ Q\, v\, m_1' \\ m_1' \perp m_2 \end{cases}$$

Function with garbage collection

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

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

What is missing from our current interpretation of triple?

Function with garbage collection

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

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

What is missing from our current interpretation of triple?

From:

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

To:

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

We need the post-condition to describe only a subset of the output heap.

Interpretation of triples (3/3)

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

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

$$\forall m_1\, m_2.\ \begin{cases} H\, m_1 \\ m_1 \perp m_2 \end{cases} \Rightarrow\ \exists v m_1' m_3.\ \begin{cases} \langle t, m_1 \uplus m_2 \rangle \Downarrow \langle v, m_1' \uplus m_2 \uplus m_3 \rangle \\ Q\, v\, m_1' \\ m_1' \perp m_2 \perp m_3 \end{cases}$$

Interpretation of triples (3/3), revisited

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

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

Interpretation of triples (3/3), revisited

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

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

Definition (Separation Logic Triple)

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

$$\forall H' m.\; (H \star H')\, m \;\Rightarrow\; \exists v m'.\; \langle t, m\rangle \Downarrow \langle v, m'\rangle \;\wedge\; (Q\, v \star H' \star \mathsf{GC})\, m'$$

Summary

Separation Logic triple:

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

Specification of a function:

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

Specification of primitive functions:

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

Interpretation of triples: see definition.

Summary of Course 1

Summary of Chapter 1

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

Summary of Chapter 2

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

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

Summary of Chapter 3

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

Split and merge of segments:

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

Summary of Chapter 4

Common representation predicate for all binary trees:

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

Invariants are expressed on the pure trees:

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

Operations are specified in terms of the model. For example, `add x p` changes $p \rightsquigarrow \mathsf{MsearchTree}\, E$ into $p \rightsquigarrow \mathsf{MsearchTree}\, (E \cup \{x\})$.

Summary of Chapter 5

Iterated separating conjunction, written $\circledast$.

For Union-Find:

$$\mathop{\circledast}_{(p,q)\in G} p \mapsto q$$

Summary of Chapter 6

Separation Logic triple:

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

Specification of a function:

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

Specification of primitive functions:

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

Interpretation of triples: see definition.

Exercises

- Exam from 2014, Exercise 2: Circular lists.

Available from the webpage of the course.

`https://madiot.fr/sepcourse/`

Smart constructors for complete binary trees

A node constructor:

```
let mk_node x p1 p2 =
  { item = x; left = p1; right = p2 }
```

Specification:

$$\{p_1 \rightsquigarrow \mathsf{MtreeDepth}\, n\, T_1 \star p_2 \rightsquigarrow \mathsf{MtreeDepth}\, n\, T_2\}$$
$$(\texttt{mk_node x p1 p2})$$
$$\{\lambda p.\ p \rightsquigarrow \mathsf{MtreeDepth}\,(n+1)\,(\mathsf{Node}\, x\, T_1\, T_2)\}$$

Smart constructors for complete binary trees

A node constructor:

```
let mk_node x p1 p2 =
  { item = x; left = p1; right = p2 }
```

Specification:

$$\{p_1 \leadsto \mathsf{MtreeDepth}\, n\, T_1 \,\star\, p_2 \leadsto \mathsf{MtreeDepth}\, n\, T_2\,\}$$
$$(\texttt{mk_node x p1 p2})$$
$$\{\lambda p.\ p \leadsto \mathsf{MtreeDepth}\,(n+1)\,(\mathsf{Node}\, x\, T_1\, T_2)\}$$

Specification of the leaf constructor:

$$\{[\,]\}\ (\texttt{null})\ \{\lambda p.\ p \leadsto \mathsf{MtreeDepth}\, 0\, \mathsf{Leaf}\}$$

The end!