Separation Logic 1/4

Jean-Marie Madiot

Inria Paris

January 24, 2023

slides (mostly) from Arthur Charguéraud

Separation Logic

Hoare Logic / Floyd-Hoare logic / Program logic / Axiomatic semantics

- mathematical proofs for imperative programs with variables
- tedious for pointer aliasing, concurrent programs

Separation Logic: Hoare logic with a more robust notion of memory

- allocation on the heap
- operations on pointers
- many extensions, including concurrent programs

Origins

- Burstall (1972): reasoning on with no sharing
  Distinct Nonrepeating List Systems
- Reynolds (1999): separating conjunction
  Intuitionistic Reasoning about Shared Mutable
- O'Hearn and Pym (1999): linear resources
  The Logic of Bunched Implications
- O'Hearn, Reynolds, Yang (2001)
  Local Reasoning about Programs that Alter Data Structures.

Examples

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
Rust	Jung et al	MPI	Coq-Iris
Time complexity	Guéneau et al	Inria	Coq
Multicore OCaml	Mével et al	Inria	Coq-Iris
Space complexity	Madiot et al	Inria	Coq-Iris
...	...	...	Coq-Iris

Interactive vs automated

Fully automated (e.g. Infer, SpaceInvader, Predator, MemCAD, SLAyer)

- find many bugs
- don't find proofs

Semi-automated (Smallfoot, Heap Hop, VeriFast, Viper)

- work well on some classes of programs
- rely on user-provided invariants
- blackbox problem (hard to debug, extend, prove...)

Interactive (Iris, VST, Ynot, CFML):

- verified
- easier to debug, understand, extend
- expressive
- often slower

Choice of the logic

Most research projects, including mine, define separation logic inside a logic framework.

We will use **Coq** and more precisely **Iris**, which is fact a whole proof mode inside Coq.

Installation:

- Install `opam`
- Create new switch
  `opam switch create 4.12.1-SL 4.12.1`
- Install Iris and its toy langage:
  `opam install coq-iris-heap-lang`

Differences with previous years

```
opam switch create 4.12.1-SL 4.12.1
opam install coq-iris-heap-lang
```

Previous years, CFML:

- typed ML-langage
- termination
- tools to accomodate ocaml

This year: **Iris**, with its toy langage

- prolific research production
- concurrent programs
- very expressive
- no types (in this course)
- no termination (in this course)

Chapter 1

Separation Logic Operators

The heap in programming

“The heap”

- = the dynamically-allocated memory
- `malloc` in C, `new` in some object-oriented languages,
- sometimes implicit, especially in langages with garbage collection such as Python, Javascript, OCaml
- contains most things (not local variables, which are on the stack)

Mathematical (sub)heaps

Definition

A *map*, or *partial function*, from a set X to a set Y is a subset F of $X \times Y$ such that $(x, y_1) \in F \wedge (x, y_2) \in F \Rightarrow y_1 = y_2$.

Definition

A *subheap*, or more simply *heap*, is a finite map from *locations* (= memory adresses) to *values*.

Examples, with locations = values = $\mathbb{N}$:

- the empty heap $\varnothing$
- $\{(1,2)\}$ and $\{(1,2),(2,3)\}$ are heaps,
- $\{(2,1)\} \cup \{(2,3)\}$ is not a heap.

Joining

When $\mathrm{dom}(h_1) \cap \mathrm{dom}(h_2) = \varnothing$ we write $h_1 \uplus h_2$ for $h_1 \cup h_2$.

Heap predicates

A *heap predicate* H is a predicate on heaps.
i.e. if h is a heap then $H\,h$ is a proposition.

In Coq: H : `heap` $\rightarrow$ `Prop` where `Prop` is the type of propositions.

Primitive heap predicates:

$\ulcorner\,\urcorner$	empty heap
$\ulcorner P \urcorner$	pure fact
$l \mapsto v$	singleton heap
$H * H'$	separating conjunction
$\exists x, H$	existential quantification

Empty heap and pure facts

Definition:

$$\ulcorner\urcorner \equiv \lambda m.\ m = \emptyset$$

$$\ulcorner P \urcorner \equiv \lambda m.\ m = \emptyset \wedge P$$

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

Before: $\ulcorner\urcorner$

After: $\ulcorner a = 3 \wedge b = 4 \urcorner$

Empty heap and pure facts

Definition:

$$\begin{array}{rcl} \ulcorner\urcorner & \equiv & \lambda m.\ m = \varnothing \\ \ulcorner P \urcorner & \equiv & \lambda m.\ m = \varnothing \ \wedge\ P \end{array}$$

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

Before: $\ulcorner\urcorner$

After: $\ulcorner a = 3\ \wedge\ b = 4 \urcorner$

Observe that $\ulcorner\urcorner$ is equivalent to $\ulcorner$True$\urcorner$.

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`".

$$\text{Before:} \quad \ulcorner\urcorner$$
$$\text{After:} \quad r \mapsto 3$$

Singleton heap

Definition:

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

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

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

Example: specification of "`incr s`".

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

Separating conjunction

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

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

Separating conjunction

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

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

Definition:

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

where:

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

Heaps and heap predicates

Exercise: give heaps satisfying the following heap predicates

$$\ulcorner\,\urcorner \qquad \ulcorner 0 = 1 \urcorner \qquad \ulcorner 1 = 1 \urcorner \qquad \ulcorner 1 = 1 \urcorner * \ulcorner 0 = 1 \urcorner \qquad 1 \mapsto 2$$

$$(1 \mapsto 2) * \ulcorner 1 = 1 \urcorner \qquad (1 \mapsto 2) * (1 \mapsto 3) \qquad (1 \mapsto 2) * (2 \mapsto 1)$$

Heaps and heap predicates

Exercise: give heaps satisfying the following heap predicates

$$\ulcorner\,\urcorner \qquad \ulcorner 0 = 1 \urcorner \qquad \ulcorner 1 = 1 \urcorner \qquad \ulcorner 1 = 1 \urcorner * \ulcorner 0 = 1 \urcorner \qquad 1 \mapsto 2$$

$$(1 \mapsto 2) * \ulcorner 1 = 1 \urcorner \qquad (1 \mapsto 2) * (1 \mapsto 3) \qquad (1 \mapsto 2) * (2 \mapsto 1)$$

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`.

Heaps and heap predicates

Exercise: give heaps satisfying the following heap predicates

$$\ulcorner\ \urcorner \qquad \ulcorner 0 = 1 \urcorner \qquad \ulcorner 1 = 1 \urcorner \qquad \ulcorner 1 = 1 \urcorner * \ulcorner 0 = 1 \urcorner \qquad 1 \mapsto 2$$

$$(1 \mapsto 2) * \ulcorner 1 = 1 \urcorner \qquad (1 \mapsto 2) * (1 \mapsto 3) \qquad (1 \mapsto 2) * (2 \mapsto 1)$$

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: $(r \mapsto 5) * (s \mapsto 3) * (t \mapsto 5)$.

Heaps and heap predicates

Exercise: give heaps satisfying the following heap predicates

$$\ulcorner\,\urcorner \qquad \ulcorner 0 = 1 \urcorner \qquad \ulcorner 1 = 1 \urcorner \qquad \ulcorner 1 = 1 \urcorner * \ulcorner 0 = 1 \urcorner \qquad 1 \mapsto 2$$

$$(1 \mapsto 2) * \ulcorner 1 = 1 \urcorner \qquad (1 \mapsto 2) * (1 \mapsto 3) \qquad (1 \mapsto 2) * (2 \mapsto 1)$$

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: $(r \mapsto 5) * (s \mapsto 3) * (t \mapsto 5)$.

Correct answer:

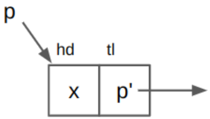
1. $(r \mapsto 5) * (s \mapsto 3) * \ulcorner t = r \urcorner$
2. $(r \mapsto 6) * (s \mapsto 3) * \ulcorner t = r \urcorner$
3. $(r \mapsto 7) * (s \mapsto 3) * \ulcorner t = r \urcorner$

Record fields

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

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

Example:



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

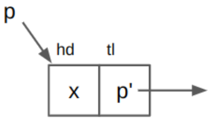
$$p.\mathsf{tl} \mapsto p'$$

Record fields

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

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

Example:



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

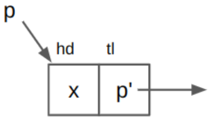
$$p.\mathsf{tl} \mapsto p'$$

In the C memory model:

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

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

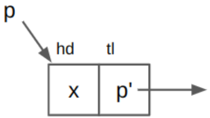
Representation of list cells



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

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

Representation of list cells



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

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

Remark: the new arrow symbol will be overloaded later.

Existential quantification

Definition:

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

Compare:

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

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

Existential quantification

Exercise: give heaps satisfying the following heap predicates

$$\exists x.\, \ulcorner(1 \mapsto x)\urcorner \qquad \exists x.\, (1 \mapsto x) * (2 \mapsto x) \qquad \exists x.\, \ulcorner x = x + 1\urcorner$$

$$\exists x.\, (x \mapsto x + 1) * (x + 1 \mapsto x) \qquad \exists x.\, 1 \mapsto x \qquad \exists x.\, (x \mapsto 1) * (x \mapsto 2)$$

$$\exists P.\, \ulcorner P\urcorner \qquad \exists H.\, H$$

Summary

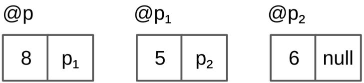
$$
\begin{array}{lcl}
\ulcorner\,\urcorner & \equiv & \ulcorner\textsf{True}\urcorner \\
\ulcorner P \urcorner & \equiv & \lambda m.\ m = \varnothing \ \wedge\ P \\
l \mapsto v & \equiv & \lambda m.\ m = \{(l, v)\} \wedge l \neq \textsf{null} \\
H_1 * H_2 & \equiv & \lambda m.\ \exists m_1 m_2. \begin{cases}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

Chapter 2

Representation Predicate for Lists

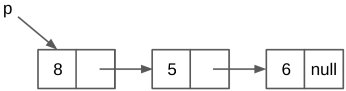
Implementation of mutable lists

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



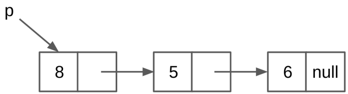

```
type 'a cell = { mutable hd : 'a;
                 mutable tl : 'a cell }
```

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



Representation of mutable lists

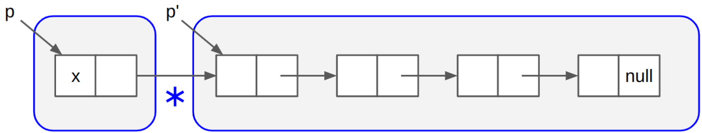
$$L \;=\; 8 :: 5 :: 6 :: \mathsf{nil}$$



$$\begin{array}{rcl} p \rightsquigarrow \mathsf{MList}\,L & \equiv & \quad \exists p_1.\;\; p \rightsquigarrow \{\!|\mathsf{hd}{=}8;\ \mathsf{tl}{=}p_1|\!\} \\ & & *\;\; \exists p_2.\;\; p_1 \rightsquigarrow \{\!|\mathsf{hd}{=}5;\ \mathsf{tl}{=}p_2|\!\} \\ & & *\;\; \exists p_3.\;\; p_2 \rightsquigarrow \{\!|\mathsf{hd}{=}6;\ \mathsf{tl}{=}p_3|\!\} \\ & & *\;\; \ulcorner p_3 = \mathsf{null} \urcorner \end{array}$$

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

Representation predicate



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

Separation properties

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

Separation enforces: no cycles, and no sharing.

Union heap predicate

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

Equivalent to:

$$
\begin{aligned}
p \leadsto \mathsf{MList}\, L \quad \equiv \quad & \qquad \ulcorner L = \mathsf{nil} \,\wedge\, p = \mathsf{null} \urcorner \\
& \Cup \quad \big(\exists x L' p'.\ \ulcorner L = x :: L' \urcorner \\
& \qquad\qquad\qquad \ast\;\; p \leadsto \{\!|\mathsf{hd}{=}x;\, \mathsf{tl}{=}p'|\!\} \\
& \qquad\qquad\qquad \ast\;\; p' \leadsto \mathsf{MList}\, L' \quad \big)
\end{aligned}
$$

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:

$$\ulcorner n \geqslant 0 \urcorner$$

Post-condition, where p denotes the result:

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

List construction: proof (1/2)

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

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)

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

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:

$$(\mathsf{null} \rightsquigarrow \mathsf{MList}\, \mathsf{nil}) \ = \ \ulcorner\,\urcorner$$

List construction: proof (1/2)

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

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:

$$(\mathsf{null} \rightsquigarrow \mathsf{MList}\,\mathsf{nil}) \ = \ \ulcorner\,\urcorner$$

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

List construction: proof (2/2)

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

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

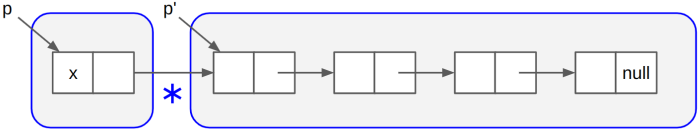
To produce $p \rightsquigarrow \mathsf{MList}\, L$, we have $p' \rightsquigarrow \mathsf{MList}\, L'$ and $p \rightsquigarrow \{\!|\mathsf{hd}{=}v;\ \mathsf{tl}{=}p'|\!\}$.

List construction: proof (2/2)

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

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 $p \rightsquigarrow \{\!|\mathsf{hd}{=}v;\ \mathsf{tl}{=}p'|\!\}$.



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

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:

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

In-place list reversal: invariants

Before the loop:

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

After the loop:

In-place list reversal: invariants

Before the loop:

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

After the loop:

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

In-place list reversal: invariants

Before the loop:

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

After the loop:

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

Loop invariant:

In-place list reversal: invariants

Before the loop:

$$r \mapsto p_0 \ * \ s \mapsto \mathsf{null} \ * \ p_0 \leadsto \mathsf{MList}\, L$$

After the loop:

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

Loop invariant:

$$\begin{aligned} \exists p q L_1 L_2. \quad & r \mapsto p \ * \ p \leadsto \mathsf{MList}\, L_2 \\ * \ & s \mapsto q \ * \ q \leadsto \mathsf{MList}\, L_1 \\ * \ & \ulcorner L = \mathsf{rev}\, L_1 +\!\!+\, L_2 \urcorner \end{aligned}$$

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

Invariant:

$$\exists p q L_1 L_2.\ r \mapsto p \ * \ s \mapsto q \ * \ p \rightsquigarrow \mathsf{MList}\, L_2 \ * \ q \rightsquigarrow \mathsf{MList}\, L_1 \ * \ \ulcorner L = \mathsf{rev}\, L_1 +\!\!+ L_2 \urcorner$$

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

$r \mapsto p_0 \ * \ p_0 \rightsquigarrow \mathsf{MList}\, L \ * \ s \mapsto \mathsf{null} \ * \ \mathsf{null} \rightsquigarrow \mathsf{MList}\, \mathsf{nil} \ * \ \ulcorner L = \mathsf{rev}\, \mathsf{nil} +\!\!+ L \urcorner$

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

Invariant:

$$\begin{aligned}\exists p q L_1 L_2.\ & r \mapsto p \ * \ s \mapsto q \\ & * \ p \rightsquigarrow \mathsf{MList}\, L_2 \ * \ q \rightsquigarrow \mathsf{MList}\, L_1 \\ & * \ \ulcorner L = \mathsf{rev}\, L_1 +\!\!+ L_2 \urcorner\end{aligned}$$

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

$$r \mapsto p_0 \ * \ p_0 \rightsquigarrow \mathsf{MList}\, L \ * \ s \mapsto \mathsf{null} \ * \ \mathsf{null} \rightsquigarrow \mathsf{MList}\,\mathsf{nil} \ * \ \ulcorner L = \mathsf{rev}\,\mathsf{nil} +\!\!+ L \urcorner$$

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

$$r \mapsto \mathsf{null} \ * \ \mathsf{null} \rightsquigarrow \mathsf{MList}\, L_2 \ * \ s \mapsto q \ * \ q \rightsquigarrow \mathsf{MList}\, L_1 \ * \ \ulcorner L = \mathsf{rev}\, L_1 +\!\!+ L_2 \urcorner$$

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

Invariant:

$$\exists pqL_1L_2.\ r \mapsto p \ * \ s \mapsto q \\ * \ p \rightsquigarrow \mathsf{MList}\, L_2 \ * \ q \rightsquigarrow \mathsf{MList}\, L_1 \\ * \ \ulcorner L = \mathsf{rev}\, L_1 +\!\!+ L_2 \urcorner$$

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

$$r \mapsto p_0 \ * \ p_0 \rightsquigarrow \mathsf{MList}\, L \ * \ s \mapsto \mathsf{null} \ * \ \mathsf{null} \rightsquigarrow \mathsf{MList}\, \mathsf{nil} \ * \ \ulcorner L = \mathsf{rev}\, \mathsf{nil} +\!\!+ L \urcorner$$

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

$$r \mapsto \mathsf{null} \ * \ \mathsf{null} \rightsquigarrow \mathsf{MList}\, L_2 \ * \ s \mapsto q \ * \ q \rightsquigarrow \mathsf{MList}\, L_1 \ * \ \ulcorner L = \mathsf{rev}\, L_1 +\!\!+ L_2 \urcorner$$

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

$$(\mathsf{null} \rightsquigarrow \mathsf{MList}\, L) \ = \ \ulcorner L = \mathsf{nil} \urcorner$$

Conversion rule for empty lists

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

Let us prove: $\quad (\mathsf{null} \rightsquigarrow \mathsf{MList}\,L) \;=\; \ulcorner L = \mathsf{nil} \urcorner$

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

$$\ulcorner \mathsf{nil} = \mathsf{nil} \urcorner \;=\; \ulcorner\,\urcorner \;=\; (\mathsf{null} \rightsquigarrow \mathsf{MList}\,\mathsf{nil})$$

Conversion rule for empty lists

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

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

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

$$\ulcorner \mathsf{nil} = \mathsf{nil} \urcorner \ = \ \ulcorner\,\urcorner \ = \ (\mathsf{null} \rightsquigarrow \mathsf{MList}\, \mathsf{nil})$$

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

$$\mathsf{null} \rightsquigarrow \mathsf{MList}\, (x :: L') \ = \ \big(\exists p'.\ \mathsf{null} \rightsquigarrow \{\!|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\!\} \ \ast \ p' \rightsquigarrow \mathsf{MList}\, L'\big)$$

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

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

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

$$p \rightsquigarrow \mathsf{MList}\, L_2 \;*\; q \rightsquigarrow \mathsf{MList}\, L_1 \;*\; \ulcorner L = \mathsf{rev}\, L_1 + L_2 \urcorner$$

to

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

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

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

$$p \rightsquigarrow \mathsf{MList}\, L_2 \; * \; q \rightsquigarrow \mathsf{MList}\, L_1 * \; \ulcorner L = \mathsf{rev}\, L_1 +\!\!+ L_2 \urcorner$$

to

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

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

$$p \rightsquigarrow \{\!| \mathsf{hd}{=}x;\ \mathsf{tl}{=}q |\!\} \; * \; q \rightsquigarrow \mathsf{MList}\, L_1 \\ * \; p' \rightsquigarrow \mathsf{MList}\, L_2' \; * \; \ulcorner L = \mathsf{rev}\, L_1 +\!\!+ (x :: L_2') \urcorner$$

to

$$q \rightsquigarrow \mathsf{MList}\,(x :: \mathsf{rev}\, L_1) * \; p' \rightsquigarrow \mathsf{MList}\, L_2' * \; \ulcorner L = \mathsf{rev}\,(x :: L_1) +\!\!+ L_2 \urcorner$$

Conversion rules for nonempty lists

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

$$
\begin{aligned}
p \rightsquigarrow \mathsf{MList}\, L * \ulcorner p \neq \mathsf{null} \urcorner \quad = \quad \exists x L' p'. \quad & \ulcorner L = x :: L' \urcorner \\
* \;\; & p \rightsquigarrow \{\!|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\!\} \\
* \;\; & p' \rightsquigarrow \mathsf{MList}\, L'
\end{aligned}
$$

Summary

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

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

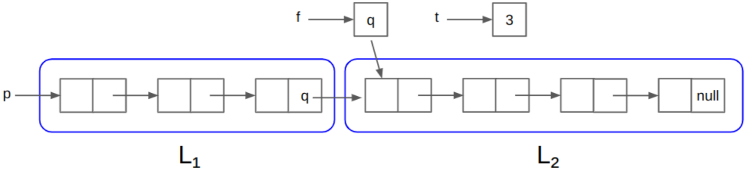
Before the loop:

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

After the loop:

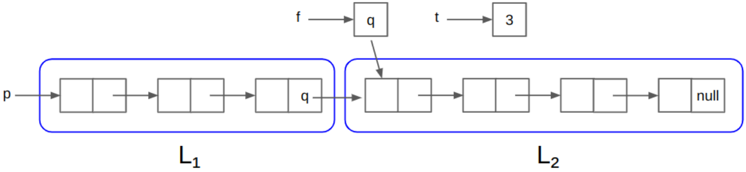
$$(p \rightsquigarrow \mathsf{MList}\, L) \; * \; (f \mapsto \mathsf{null}) \; * \; (t \mapsto \mathsf{length}\, L)$$

Mlength: loop invariant



Loop invariant:

Mlength: loop invariant

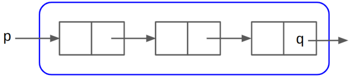


Loop invariant:

$$\exists L_1 L_2 q. \quad \ulcorner L = L_1 +\!\!+ L_2 \urcorner \ast (t \mapsto \mathsf{length}\, L_1) \ast (f \mapsto q)$$
$$\ast\ (p \leadsto \mathsf{MlistSeg}\, q\, L_1) \ast (q \leadsto \mathsf{MList}\, L_2)$$

Representation predicate for list segments

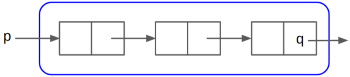
$$
\begin{array}{lcl}
p \rightsquigarrow \mathsf{MList}\,L & \equiv & \mathsf{match}\,L\,\mathsf{with} \\
& & \mid \mathsf{nil} \;\Rightarrow\; \ulcorner p = \mathsf{null} \urcorner \\
& & \mid x :: L' \;\Rightarrow\; \exists p'.\; p \rightsquigarrow \{\!|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\!\} \\
&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



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).

Representation predicate for list segments

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



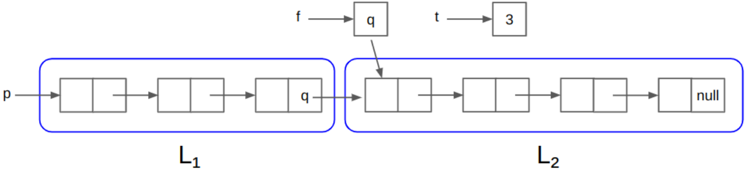
Exercise: generalize MList to define $p \rightsquigarrow \mathsf{MlistSeg}\, q\, L$, where L denotes the list of items in the list segment from p (inclusive) to q (exclusive).

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

Remark:

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

Mlength: proof

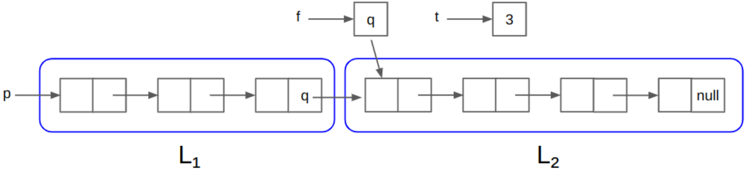


Enter:

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

$$\ulcorner \urcorner \ = \ (p \rightsquigarrow \mathsf{MlistSeg}\, p \,\mathsf{nil})$$

Mlength: proof



Enter:

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

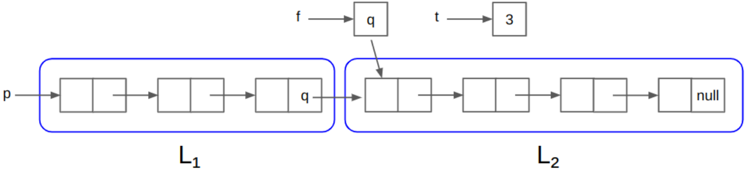
$$\ulcorner\,\urcorner \;=\; (p \rightsquigarrow \mathsf{MlistSeg}\, p\, \mathsf{nil})$$

Exit:

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

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

Mlength: proof



Enter:

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

$$\ulcorner\,\urcorner \;=\; (p \rightsquigarrow \text{MlistSeg}\, p\, \text{nil})$$

Exit:

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

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

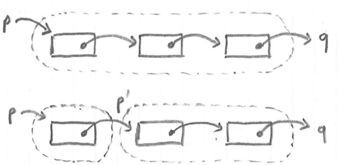
Step:

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

$$\exists q.\;\; p \rightsquigarrow \text{MlistSeg}\, q\, L_1 \;*\; q \rightsquigarrow \{\!|\text{hd}{=}x;\ \text{tl}{=}q'|\!\}$$

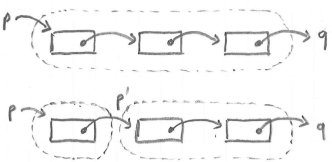
$$=\;\; p \rightsquigarrow \text{MlistSeg}\, q'\, (L_1 +\!\!+\, x :: \text{nil})$$

Splitting rules for list segments



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

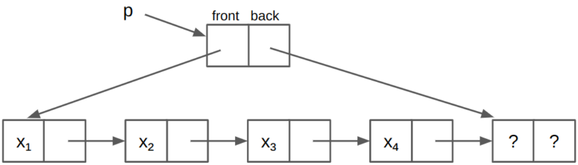
Splitting rules for list segments



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

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

An implementation of mutable queues

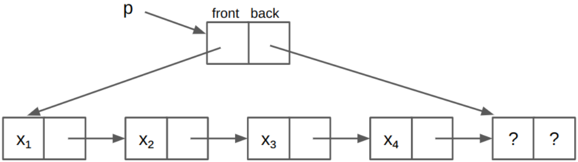


Represent a queue as a list segment, with the last cell storing no item (in fact, storing unknown values, marked "?" above)

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

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

An implementation of mutable queues



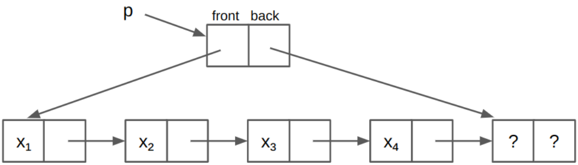
Represent a queue as a list segment, with the last cell storing no item (in fact, storing unknown values, marked "?" above)

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

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

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

An implementation of mutable queues



Represent a queue as a list segment, with the last cell storing no item (in fact, storing unknown values, marked "?" above)

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

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

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

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

Summary

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

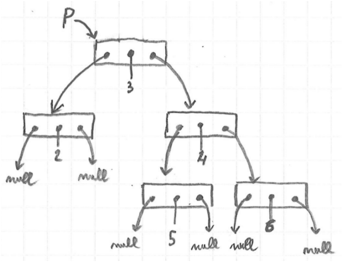
Split and merge of segments:

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

Chapter 4

Representation Predicate for Trees

Implementation of a mutable binary trees



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

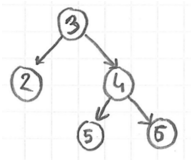
```
type node = {
  mutable item : int;
  mutable left : node;
  mutable right : node; }
```

Logical binary trees

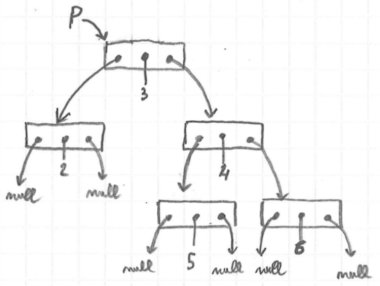
```
Inductive tree : Type :=
  | Leaf : tree
  | Node : int → tree → tree → tree.
```

Example:

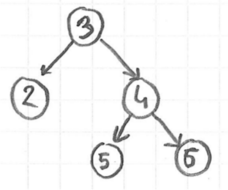
```
Node 3
    (Node 2 Leaf Leaf)
    (Node 4 (Node 5 Leaf Leaf)
            (Node 6 Leaf Leaf))
```



Representation predicate for binary trees



$T =$



Representation predicate:

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

Representation predicate for binary trees

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

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

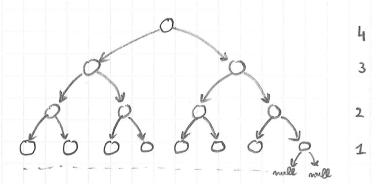
Representation predicate for binary trees

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

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

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

Complete binary tree



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

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

Complete binary tree (1/2)

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

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

Complete binary tree (1/2), solution

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

Complete binary tree (1/2), solution

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

Or:

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

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.

Complete binary tree (2/2)

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

$$p \rightsquigarrow \mathsf{MtreeDepth}\, n\, T \;\; \equiv \;\; p \rightsquigarrow \mathsf{Mtree}\, T \;\ast\; \ulcorner \mathsf{depth}\, n\, T \urcorner$$

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

Complete binary tree of unspecified depth

$$p \rightsquigarrow \mathsf{MtreeDepth}\, n\, T \;\equiv\; (p \rightsquigarrow \mathsf{Mtree}\, T) \,*\, \ulcorner \mathsf{depth}\, n\, T \urcorner$$

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

Complete binary tree of unspecified depth

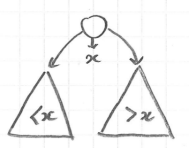
$$p \leadsto \mathsf{MtreeDepth}\, n\, T \;\equiv\; (p \leadsto \mathsf{Mtree}\, T) \,*\, \ulcorner \mathsf{depth}\, n\, T \urcorner$$

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

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

1. $\exists n.\; p \leadsto \mathsf{MtreeDepth}\, n\, T$
2. $\exists n.\; (p \leadsto \mathsf{Mtree}\, T) \,*\, \ulcorner \mathsf{depth}\, n\, T \urcorner$
3. $(p \leadsto \mathsf{Mtree}\, T) \,*\, \ulcorner \exists n.\, \mathsf{depth}\, n\, T \urcorner$

Binary search tree property



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

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

Binary search tree predicate

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

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.

$$p \rightsquigarrow \mathsf{MsearchTree}\, E \;\equiv\; \exists T.\; p \rightsquigarrow \mathsf{Mtree}\, T \;*\; \ulcorner \mathsf{search}\, T\, E \urcorner$$

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.

$$p \rightsquigarrow \mathsf{MsearchTree}\, E \ \equiv \ \exists T.\ p \rightsquigarrow \mathsf{Mtree}\, T \ * \ \ulcorner \mathsf{search}\, T\, E \urcorner$$

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:

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

Invariants are expressed on the pure trees:

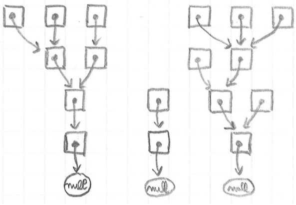
$$p \rightsquigarrow \mathsf{MsearchTree}\, E \quad \equiv \quad \exists T.\ p \rightsquigarrow \mathsf{Mtree}\, T \ * \ \ulcorner \mathsf{search}\, T\, E \urcorner$$

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



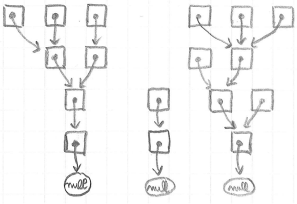
```
type node = node ref
```

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

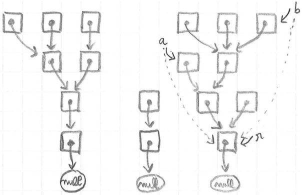


$$(p_1 \mapsto q_1) \ * \ (p_2 \mapsto q_2) \ * \ \dots \ * \ (p_n \mapsto q_n)$$

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

where G is a finite map from locations to locations.

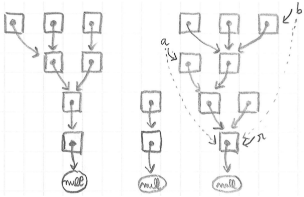
Invariants of union-find



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

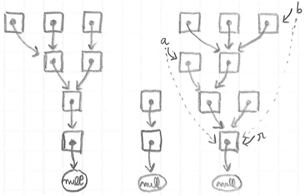
```
Inductive root : fmap loc loc → loc → loc → Prop :=
  | root_init : ∀G x,
      binds G x null →
      root G x x
  | root_step : ∀G x y r,
      binds G x y →
      y ≠null →
      root G y r →
      root G x r.
```

Specification of the union-find structure



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

Specification of the union-find structure



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

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

- pre-condition: $\ulcorner S\,a\,a\ \wedge\ S\,b\,b \urcorner\ \ast\ \mathsf{UnionFind}\,S$
- post-condition: $\ulcorner x = \mathsf{true} \Leftrightarrow S\,a\,b \urcorner\ \ast\ \mathsf{UnionFind}\,S$

Summary

Iterated separating conjunction, written $\circledast$.

For Union-Find:

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

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

Examples of triples

Example 1:

$$\{\ulcorner\urcorner\}\ (\texttt{ref 3})\ \{\lambda r.\ r \mapsto 3\}$$

Examples of triples

Example 1:

$$\{\ulcorner\urcorner\}\ (\texttt{ref 3})\ \{\lambda r.\ r \mapsto 3\}$$

Example 2:

$$\{\ulcorner\urcorner\}\ (\texttt{3})\ \{\lambda x.\ \ulcorner x = 3\urcorner\}$$

Examples of triples

Example 1:

$$\{\ulcorner\urcorner\}\ (\texttt{ref 3})\ \{\lambda r.\ r \mapsto 3\}$$

Example 2:

$$\{\ulcorner\urcorner\}\ (\texttt{3})\ \{\lambda x. \ulcorner x = 3 \urcorner\}$$

Example 3:

$$\{r \mapsto 3\}\ (\texttt{!r})\ \{\lambda x.\ \ulcorner x = 3 \urcorner * (r \mapsto 3)\}$$

Examples of triples

Example 1:

$$\{\ulcorner\urcorner\}\ (\texttt{ref 3})\ \{\lambda r.\ r \mapsto 3\}$$

Example 2:

$$\{\ulcorner\urcorner\}\ (\texttt{3})\ \{\lambda x. \ulcorner x = 3\urcorner\}$$

Example 3:

$$\{r \mapsto 3\}\ (\texttt{!r})\ \{\lambda x.\ \ulcorner x = 3\urcorner * (r \mapsto 3)\}$$

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:

$$\forall v. \quad \{\ulcorner\urcorner\}\ (\texttt{ref v})\ \{\lambda r.\ (r \mapsto v)\}$$

$$\forall r v. \quad \{r \mapsto v\}\ (\texttt{!r})\ \{\lambda x.\ \ulcorner x = v \urcorner * (r \mapsto v)\}$$

Specification of operations on memory cells

Exercise: specify the primitive operations on references.

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

Solution:

$$\forall v. \quad \{\ulcorner\urcorner\}\ (\texttt{ref v})\ \{\lambda r.\ (r \mapsto v)\}$$

$$\forall rv. \quad \{r \mapsto v\}\ (\texttt{!r})\ \{\lambda x.\ \ulcorner x = v \urcorner * (r \mapsto v)\}$$

$$\forall rvw. \quad \{r \mapsto w\}\ (\texttt{r := v})\ \{\lambda\_.\, (r \mapsto v)\}$$
$$\forall rv. \quad \{\exists w.\ r \mapsto w\}\ (\texttt{r := v})\ \{\lambda\_.\, (r \mapsto v)\}$$
$$\forall rv. \quad \{r \mapsto -\}\ (\texttt{r := v})\ \{\lambda\_.\, (r \mapsto v)\}$$

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

Specification of partial functions

Presentation 1:

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

Presentation 2:

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

Specification of operations on arrays

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

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

Notation:

$$
\begin{array}{lcl}
L[i] & \equiv & i\text{-th element of the list } L \\
L[i := v] & \equiv & \text{copy of } L \text{ with } v \text{ at index } i \\
|L| & \equiv & \text{length of } L \\
i \in \text{dom}\, L & \equiv & 0 \leqslant i < |L|
\end{array}
$$

Specification of operations on arrays

$$\forall i L p \quad i \in \mathsf{dom}\, L \Rightarrow \{p \rightsquigarrow \mathsf{Array}\, L\}$$
$$(\texttt{Array.get p i})$$
$$\{\lambda x.\ \ulcorner x = L[i] \urcorner * p \rightsquigarrow \mathsf{Array}\, L\}$$

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

Specification of operations on arrays

$$\forall i L p \quad i \in \mathsf{dom}\, L \Rightarrow \{p \rightsquigarrow \mathsf{Array}\, L\}$$
$$(\texttt{Array.get p i})$$
$$\{\lambda x.\ \ulcorner x = L[i] \urcorner \ast\ p \rightsquigarrow \mathsf{Array}\, L\}$$

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

$$\forall p L \quad \{p \rightsquigarrow \mathsf{Array}\, L\}$$
$$(\texttt{Array.length p})$$
$$\{\lambda n.\ \ulcorner n = |L| \urcorner \ast\ p \rightsquigarrow \mathsf{Array}\, L\}$$

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

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:

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

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:

$$\{\ulcorner\urcorner\}\ (\texttt{myref x})\ \{\lambda r.\ r \mapsto x \ast \exists s.\, s \mapsto r\}$$

To:

$$\{\ulcorner\urcorner\}\ (\texttt{myref x})\ \{\lambda r.\ r \mapsto x\}$$

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:

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

Summary

Separation Logic triple:

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

Specification of a function:

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

Specification of primitive functions:

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

Interpretation of triples: see definition.

Summary of Course 1

Summary of Chapter 1

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

Summary of Chapter 2

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

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

Summary of Chapter 3

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

Split and merge of segments:

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

Summary of Chapter 4

Common representation predicate for all binary trees:

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

Invariants are expressed on the pure trees:

$$p \rightsquigarrow \mathsf{MsearchTree}\, E \ \equiv \ \exists T.\ p \rightsquigarrow \mathsf{Mtree}\, T \ * \ \ulcorner \mathsf{search}\, T\, E \urcorner$$

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:

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

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:

$$\begin{array}{ll} \forall v. & \{\ulcorner\urcorner\}\ (\texttt{ref v})\ \{\lambda r.\ (r \mapsto v)\} \\ \forall rv. & \{r \mapsto v\}\ (\texttt{!r})\ \{\lambda x.\ \ulcorner x = v \urcorner * (r \mapsto v)\} \\ \forall rvv'. & \{r \mapsto v'\}\ (\texttt{r := v})\ \{\lambda\_.\ (r \mapsto v)\} \end{array}$$

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:

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

Smart constructors for complete binary trees

A node constructor:

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

Specification:

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

Specification of the leaf constructor:

$$\{\ulcorner\urcorner\}\ (\texttt{null})\ \{\lambda p.\ p \rightsquigarrow \mathsf{MtreeDepth}\, 0\, \mathsf{Leaf}\}$$

The end!