

# Separation Logic 1/4

Jean-Marie Madiot

Inria Paris

February 3, 2026

part 1/4 : some initial material from Arthur Charguéraud

# Condensed history of separation logic

**Floyd-Hoare Logic**, or *program logic*, or *axiomatic semantics*

- Alan Turing (1949), *Checking a large routine*: first program proof
- Floyd, Hoare (1967-1969): current form of rules
- fine for simple programs, tedious for pointer aliasing and concurrency

**Separation Logic**: assertions about fragments of memory/heap

- Reynolds (2002): *A Logic for Shared Mutable Data Structures*
- local reasoning, making it easy to reason about pointers

**Concurrent Separation Logic (CSL)**

- Peter O'Hearn (2007): *Resources, Concurrency and Local Reasoning*
- separation combined with critical sections for shared resources

**The Iris framework (2015-2017)**

- Expressive and complex logic designed for CSL inside Rocq
- Modalities, resources more general than memory, ghost state

# 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               | Rocq      |
| Operating system     | Shao et al      | Yale              | Rocq      |
| C (drivers)          | Yang et al      | Oxford            | Other     |
| C-light (concurrent) | Appel et al     | Princeton         | Rocq      |
| 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           | Rocq      |
| OCaml                | Charguéraud     | Inria             | Rocq      |
| SML                  | Myreen et al    | U. of Cambridge   | HOL       |
| Rust                 | Jung et al      | MPI               | Rocq-Iris |
| Time complexity      | Guéneau et al   | Inria             | Rocq      |
| Multicore OCaml      | Mével et al     | Inria             | Rocq-Iris |
| Space complexity     | Madiot et al    | Inria             | Rocq-Iris |
| ...                  | ...             | ...               | Rocq-Iris |

# Interactive vs automated

Fully automated (e.g. Infer, SpacInvader, 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
- hard to debug / understand failure

Interactive (Iris, VST, Ynot, CFML):

- verified
- easier to debug
- expressive

# Choice of the logic

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

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

Installation instructions can be found at

<https://gitlab.mpi-sws.org/iris/tutorial-pop120>

# Differences with previous years

<https://gitlab.mpi-sws.org/iris/tutorial-popl20>

2-3 years ago and before, CFML:

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

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

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

In the beginning we use a much simpler model

# 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 languages with garbage collection such as Python, Javascript, OCaml
- contains most things (not local variables, which are on the stack)

# The heap, formally

## Definition

A *partial map*, or *partial function*, from a set  $X$  to a set  $Y$  is a subset  $F$  of  $X \times Y$  that is *functional*:  $\forall xy_1y_2 (x, y_1) \in F \wedge (x, y_2) \in F \Rightarrow y_1 = y_2$ .

## Definition

A *subheap*, or *heaplet*, or just *heap*, is a finite map from *locations* (= memory addresses) to *values*.

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

- $\{(1, 2), (2, 3)\}$  and  $\emptyset$  are heaps,
- $\{(2, 1), (2, 3)\}$  is not a heap.

## Joining

When  $\text{dom}(h_1) \cap \text{dom}(h_2) = \emptyset$  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 Rocq:  $H : \text{heap} \rightarrow \text{Prop}$  where  $\text{Prop}$  is the type of propositions.

Primitive heap predicates:

|                |                            |
|----------------|----------------------------|
| $\text{!}$     | empty heap                 |
| $\text{!}P$    | pure fact                  |
| $l \mapsto v$  | singleton heap             |
| $H * H'$       | separating conjunction     |
| $\exists x, H$ | existential quantification |

## Empty heap and pure facts

Definition:

$$\begin{aligned} \ulcorner P \urcorner &\equiv \lambda m. m = \emptyset \wedge P \\ \ulcorner \urcorner &\equiv \ulcorner \text{True} \urcorner \end{aligned}$$

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

Before:  $\ulcorner \urcorner$

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

Note:  $a$  and  $b$  are program variables and  $a$  and  $b$  are logical variables. In reality  $a$  will be replaced by 3 in the rest of the program, or kept in an environment, but it is convenient to name it  $a$  in proofs.

# Singleton heap

Definition:

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

Example: specification of “`let r = ref 3`”.

Before:  $\top$

After:  $r \mapsto 3$

# Singleton heap

Definition:

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

Example: specification of “`let r = ref 3`”.

Before:  $\top$

After:  $r \mapsto 3$

Example: specification of “`incr s`”.

Before:  $s \mapsto n$  for some  $n$

After:  $s \mapsto (n + 1)$

## 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:

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

where:

$$m_1 \perp m_2 \equiv \text{dom } m_1 \cap \text{dom } m_2 = \emptyset$$

$$m_1 \uplus m_2 \equiv m_1 \cup m_2 \quad \text{when } m_1 \perp m_2$$

# Heaps and heap predicates

**Exercise:** give heaps satisfying the following heap predicates

|                                 |  |   |  |
|---------------------------------|--|---|--|
| $\lceil \rceil$                 |  | $\lceil 0 = 1 \rceil$                       |  |
| $\lceil 1 = 1 \rceil$           |  | $\lceil 1 = 1 \rceil * \lceil 0 = 1 \rceil$ |  |
| $1 \mapsto 2$                   |  | $(1 \mapsto 2) * \lceil 1 = 1 \rceil$       |  |
| $(1 \mapsto 2) * (1 \mapsto 3)$ |  | $(1 \mapsto 2) * (2 \mapsto 1)$             |  |

# Heaps and heap predicates

**Exercise:** give heaps satisfying the following heap predicates

|                                 |             |                                 |            |
|---------------------------------|-------------|---------------------------------|------------|
| $\top$                          | $\emptyset$ | $\top$                          | impossible |
| $\top$                          | $\emptyset$ | $\top * \top$                   | impossible |
| $1 \mapsto 2$                   |             | $(1 \mapsto 2) * \top$          |            |
| $(1 \mapsto 2) * (1 \mapsto 3)$ |             | $(1 \mapsto 2) * (2 \mapsto 1)$ |            |

# Heaps and heap predicates

**Exercise:** give heaps satisfying the following heap predicates

|                                 |              |   |                      |
|---------------------------------|--------------|---|----------------------|
| $\ulcorner \urcorner$           | $\emptyset$  | $\ulcorner 0 = 1 \urcorner$                             | impossible           |
| $\ulcorner 1 = 1 \urcorner$     | $\emptyset$  | $\ulcorner 1 = 1 \urcorner * \ulcorner 0 = 1 \urcorner$ | impossible           |
| $1 \mapsto 2$                   | $\{(1, 2)\}$ | $(1 \mapsto 2) * \ulcorner 1 = 1 \urcorner$             | $\{(1, 2)\}$         |
| $(1 \mapsto 2) * (1 \mapsto 3)$ | impossible   | $(1 \mapsto 2) * (2 \mapsto 1)$                         | $\{(1, 2), (2, 1)\}$ |

# Heaps and heap predicates

## 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:

- 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:

- 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:

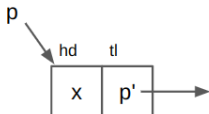
- 1  $(r \mapsto 5) * (s \mapsto 3) * \ulcorner t = r \urcorner$
- 2  $(r \mapsto 6) * (s \mapsto 3) * \ulcorner t = r \urcorner$
- 3  $(r \mapsto 7) * (s \mapsto 3) * \ulcorner t = r \urcorner$

# Record fields

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

$$p.f \mapsto v \equiv (p + \text{offset of } f) \mapsto v$$

Example:

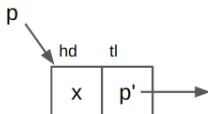


$$p \mapsto x \quad * \quad p + 1 \mapsto p'$$

sometimes written

$$p.hd \mapsto x \quad * \quad p.tl \mapsto p'$$

# Representation of list cells



$$p \mapsto (x, p') \equiv p \mapsto x * (p + 1) \mapsto p'$$

In general

$$p \mapsto (v_0, \dots, v_n) \equiv p \mapsto v_0 * \dots * (p + n) \mapsto v_n$$

Sometimes, to specify the names of the field, we write:

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

# Existential quantification

Definition:

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

Compare:

$(\exists x. P) : \text{Prop}$                       when  $(P : \text{Prop})$

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

# Existential quantification

**Exercise:** give heaps satisfying the following heap predicates

$$\exists x. \lceil 1 \mapsto x \rceil$$

$$\exists x. (1 \mapsto x) * (2 \mapsto x)$$

$$\exists x. \lceil x = x + 1 \rceil$$

$$\exists x. (x \mapsto x + 1) * (x + 1 \mapsto x)$$

$$\exists x. 1 \mapsto x$$

$$\exists x. (x \mapsto 1) * (x \mapsto 2)$$

$$\exists P. \lceil P \rceil$$

$$\exists H. H$$

# Existential quantification

**Exercise:** give heaps satisfying the following heap predicates

$\exists x. \lceil 1 \mapsto x \rceil$   
nonsense

$\exists x. (1 \mapsto x) * (2 \mapsto x)$   
 $\{(1, 7), (2, 7)\}$

$\exists x. \lceil x = x + 1 \rceil$   
impossible

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

$\exists x. 1 \mapsto x$   
 $\{(1, 2)\}$

$\exists x. (x \mapsto 1) * (x \mapsto 2)$   
impossible

$\exists P. \lceil P \rceil$   
 $\emptyset$

$\exists H. H$   
any heap

# Summary

$$\text{true} \equiv \text{True}$$

$$\text{!}P \equiv \lambda m. m = \emptyset \wedge P$$

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

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

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

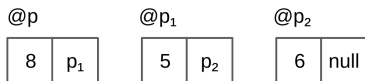
# Chapter 2

## Representation Predicate for Lists

# Linked lists

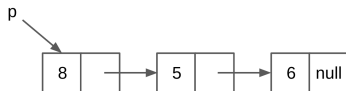
Example of an assertion about a linked list containing [8; 5; 6]

$$p \mapsto (8, p_1) * p_1 \mapsto (5, p_2) * p_2 \mapsto (6, \text{null})$$



We want it to work for any intermediate pointers, so we write instead:

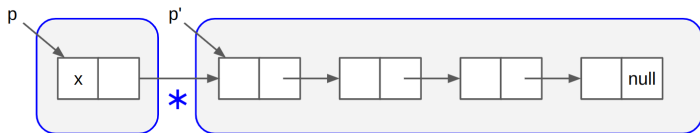
$$\exists p_1. p \mapsto (8, p_1) * \exists p_2. p_1 \mapsto (5, p_2) * p_2 \mapsto (6, \text{null})$$



# Representation predicate for lists

Where  $L$  is a “logical” list (e.g. a Rocq list), we write:

$$\begin{aligned} p \rightsquigarrow \text{MList } L &\equiv \text{match } L \text{ with} \\ &| \text{nil} \Rightarrow \ulcorner p = \text{null} \urcorner \\ &| x :: L' \Rightarrow \exists p'. \quad p \mapsto (x, p') * p' \rightsquigarrow \text{MList } L' \end{aligned}$$



Note:  $p \rightsquigarrow \text{MList } L$  is notation for  $\text{MList } L p$ .

# Implementation of mutable lists in OCaml

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

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



The we write just:

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

but behind, there are 3 allocations and 6 assignments

# Separation properties

$$p_1 \rightsquigarrow \text{MList } L_1 * p_2 \rightsquigarrow \text{MList } L_2 * p_3 \rightsquigarrow \text{MList } L_3$$

Separation enforces: no cycles, and no sharing.

# Union heap predicate

$$\begin{aligned} p \rightsquigarrow \text{MList } L &\equiv \text{match } L \text{ with} \\ &| \text{nil} \Rightarrow \lceil p = \text{null} \rceil \\ &| x :: L' \Rightarrow \exists p'. p \mapsto (x, p') * p' \rightsquigarrow \text{MList } L' \end{aligned}$$

Equivalent to:

$$\begin{aligned} p \rightsquigarrow \text{MList } L &\equiv \lceil L = \text{nil} \wedge p = \text{null} \rceil \\ &\vee (\exists x L' p'. \lceil L = x :: L' \rceil \\ &\quad * p \mapsto (x, p') * p' \rightsquigarrow \text{MList } L') \end{aligned}$$

where:

$$H_1 \vee H_2 \equiv \lambda m. H_1 m \vee H_2 m$$

# 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:

$$\lceil n \geq 0 \rceil$$

Post-condition, where  $p$  denotes the result:

$$\exists L. p \rightsquigarrow \text{MList } L * \lceil \text{length } L = n \wedge (\forall i. 0 \leq i < n \Rightarrow L[i] = v) \rceil$$

## List construction: proof (1/2)

$$\exists L. p \rightsquigarrow \text{MList } L * \text{'length } L = n \wedge (\forall i. 0 \leq i < n \Rightarrow L[i] = v)'$$

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

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

## List construction: proof (1/2)

$$\exists L. p \rightsquigarrow \text{MList } L * \text{'length } L = n \wedge (\forall i. 0 \leq i < n \Rightarrow L[i] = v)\text{'}$$

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

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

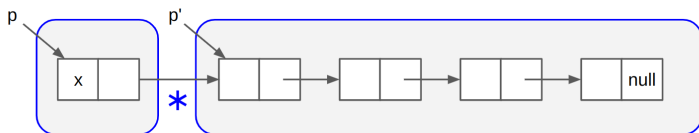
$$(\text{null} \rightsquigarrow \text{MList nil}) = \text{'}$$

## List construction: proof (2/2)

$\exists L. p \rightsquigarrow \text{MList } L * \text{'length } L = n \wedge (\forall i. 0 \leq i < n \Rightarrow L[i] = v)\text{'}$

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

To produce  $p \rightsquigarrow \text{MList } L$ , we have  $p' \rightsquigarrow \text{MList } L'$  and  $p \mapsto (v, p')$ .



$$(\exists p'. p \mapsto (v, p') * p' \rightsquigarrow \text{MList } L') = p \rightsquigarrow \text{MList } (v :: L')$$

## 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:

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

# In-place list reversal: invariants

Before the loop:

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

After the loop:

## In-place list reversal: invariants

Before the loop:

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

After the loop:

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

# In-place list reversal: invariants

Before the loop:

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

After the loop:

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

Loop invariant:

# In-place list reversal: invariants

Before the loop:

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

After the loop:

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

Loop invariant:

$$\begin{aligned} \exists pqL_1L_2. \quad & r \mapsto p * p \rightsquigarrow \text{MList } L_2 \\ & * s \mapsto q * q \rightsquigarrow \text{MList } L_1 \\ & * \text{' } L = \text{rev } L_1 \uparrow\uparrow L_2 \text{' } \end{aligned}$$

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

Invariant:

$$\begin{aligned} & \exists pqL_1L_2. r \mapsto p * s \mapsto q \\ & * p \rightsquigarrow \text{MList } L_2 * q \rightsquigarrow \text{MList } L_1 \\ & * \text{'}L = \text{rev } L_1 \text{++} L_2 \text{'} \end{aligned}$$

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

$$r \mapsto p_0 * p_0 \rightsquigarrow \text{MList } L * s \mapsto \text{null} * \text{null} \rightsquigarrow \text{MList nil} * \text{'}L = \text{rev nil++} L \text{'}$$

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

Invariant:

$$\begin{aligned} & \exists pqL_1L_2. r \mapsto p * s \mapsto q \\ & * p \rightsquigarrow \text{MList } L_2 * q \rightsquigarrow \text{MList } L_1 \\ & * \text{'}L = \text{rev } L_1 \uparrow\uparrow L_2\text{'} \end{aligned}$$

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

$$r \mapsto p_0 * p_0 \rightsquigarrow \text{MList } L * s \mapsto \text{null} * \text{null} \rightsquigarrow \text{MList nil} * \text{'}L = \text{rev nil} \uparrow\uparrow L\text{'}$$

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

$$r \mapsto \text{null} * \text{null} \rightsquigarrow \text{MList } L_2 * s \mapsto q * q \rightsquigarrow \text{MList } L_1 * \text{'}L = \text{rev } L_1 \uparrow\uparrow L_2\text{'}$$

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

Invariant:

$$\begin{aligned} & \exists pqL_1L_2. r \mapsto p * s \mapsto q \\ & * p \rightsquigarrow \text{MList } L_2 * q \rightsquigarrow \text{MList } L_1 \\ & * \text{'}L = \text{rev } L_1 \uparrow\uparrow L_2\text{'} \end{aligned}$$

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

$$r \mapsto p_0 * p_0 \rightsquigarrow \text{MList } L * s \mapsto \text{null} * \text{null} \rightsquigarrow \text{MList nil} * \text{'}L = \text{rev nil} \uparrow\uparrow L\text{'}$$

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

$$r \mapsto \text{null} * \text{null} \rightsquigarrow \text{MList } L_2 * s \mapsto q * q \rightsquigarrow \text{MList } L_1 * \text{'}L = \text{rev } L_1 \uparrow\uparrow L_2\text{'}$$

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

$$(\text{null} \rightsquigarrow \text{MList } L) = \text{'}L = \text{nil}\text{'}$$

## Conversion rule for empty lists

$$\begin{aligned} p \rightsquigarrow \text{MList } L &\equiv \text{match } L \text{ with} \\ &| \text{nil} \Rightarrow \lceil p = \text{null} \rceil \\ &| x :: L' \Rightarrow \exists p'. p \mapsto (x, p') * p' \rightsquigarrow \text{MList } L' \end{aligned}$$

Let us prove:  $(\text{null} \rightsquigarrow \text{MList } L) = \lceil L = \text{nil} \rceil$

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

$$\lceil \text{nil} = \text{nil} \rceil = \lceil \top \rceil = (\text{null} \rightsquigarrow \text{MList nil})$$

## Conversion rule for empty lists

$$\begin{aligned} p \rightsquigarrow \text{MList } L &\equiv \text{match } L \text{ with} \\ &| \text{nil} \Rightarrow \lceil p = \text{null} \rceil \\ &| x :: L' \Rightarrow \exists p'. p \mapsto (x, p') * p' \rightsquigarrow \text{MList } L' \end{aligned}$$

Let us prove:  $(\text{null} \rightsquigarrow \text{MList } L) = \lceil L = \text{nil} \rceil$

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

$$\lceil \text{nil} = \text{nil} \rceil = \lceil \top \rceil = (\text{null} \rightsquigarrow \text{MList } \text{nil})$$

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

$$\text{null} \rightsquigarrow \text{MList } (x :: L') = (\exists p'. \text{null} \mapsto (x, p') * p' \rightsquigarrow \text{MList } L')$$

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

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

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

$$p \rightsquigarrow \text{MList } L_2 * q \rightsquigarrow \text{MList } L_1 * \lceil L = \text{rev } L_1 ++ L_2 \rceil$$

to

$$\begin{aligned} \exists x L'_2 p'. \lceil L_2 = x :: L'_2 \rceil * p \mapsto (x, p') * p' \rightsquigarrow \text{MList } L'_2 \\ * q \rightsquigarrow \text{MList } L_1 * \lceil L = \text{rev } L_1 ++ L_2 \rceil \end{aligned}$$

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

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

$$p \rightsquigarrow \text{MList } L_2 * q \rightsquigarrow \text{MList } L_1 * \lceil L = \text{rev } L_1 \uplus L_2 \rceil$$

to

$$\begin{aligned} \exists x L'_2 p'. \lceil L_2 = x :: L'_2 \rceil * p \mapsto (x, p') * p' \rightsquigarrow \text{MList } L'_2 \\ * q \rightsquigarrow \text{MList } L_1 * \lceil L = \text{rev } L_1 \uplus L_2 \rceil \end{aligned}$$

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

$$p \mapsto (x, q) * q \rightsquigarrow \text{MList } L_1$$

$$* p' \rightsquigarrow \text{MList } L'_2 * \lceil L = \text{rev } L_1 \uplus (x :: L'_2) \rceil$$

to

$$q \rightsquigarrow \text{MList } (x :: L_1) * p' \rightsquigarrow \text{MList } L'_2 * \lceil L = \text{rev } (x :: L_1) \uplus L_2 \rceil$$

# Conversion rules for nonempty lists

$$\begin{aligned} p \rightsquigarrow \text{MList } L &\equiv \text{match } L \text{ with} \\ &| \text{nil} \Rightarrow \lceil p = \text{null} \rceil \\ &| x :: L' \Rightarrow \exists p'. \quad p \mapsto (x, p') \\ &\quad * p' \rightsquigarrow \text{MList } L' \end{aligned}$$

$$\begin{aligned} p \rightsquigarrow \text{MList } L * \lceil p \neq \text{null} \rceil &= \exists x L' p'. \quad \lceil L = x :: L' \rceil \\ &* p \mapsto (x, p') \\ &* p' \rightsquigarrow \text{MList } L' \end{aligned}$$

# Summary

$$\begin{aligned} p \rightsquigarrow \text{MList } L &\equiv \text{match } L \text{ with} \\ &| \text{nil} \Rightarrow \ulcorner p = \text{null} \urcorner \\ &| x :: L' \Rightarrow \exists p'. \quad p \mapsto (x, p') \\ &\quad * p' \rightsquigarrow \text{MList } L' \end{aligned}$$

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

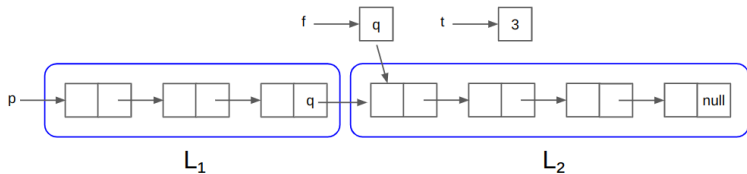
Before the loop:

$$(p \rightsquigarrow \text{MList } L) * (f \mapsto p) * (t \mapsto 0)$$

After the loop:

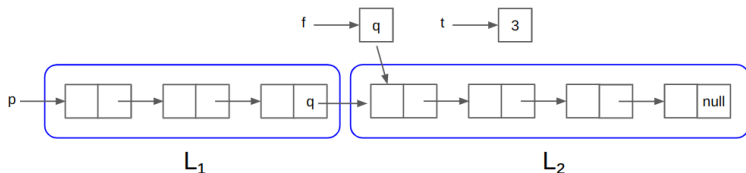
$$(p \rightsquigarrow \text{MList } L) * (f \mapsto \text{null}) * (t \mapsto \text{length } L)$$

# Mlength: loop invariant



Loop invariant:

# Mlength: loop invariant

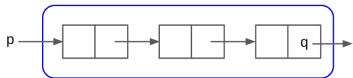


Loop invariant:

$$\begin{aligned} \exists L_1 L_2 q. \quad & \lceil L = L_1 ++ L_2 \rceil * (t \mapsto \text{length } L_1) * (f \mapsto q) \\ & * (p \rightsquigarrow \text{MlistSeg } q \ L_1) * (q \rightsquigarrow \text{Mlist } L_2) \end{aligned}$$

# Representation predicate for list segments

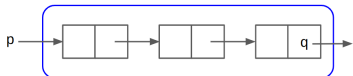
$$\begin{aligned} p \rightsquigarrow \text{MList } L &\equiv \text{match } L \text{ with} \\ &| \text{nil} \Rightarrow \ulcorner p = \text{null} \urcorner \\ &| x :: L' \Rightarrow \exists p'. p \mapsto (x, p') \\ &\quad * p' \rightsquigarrow \text{MList } L' \end{aligned}$$



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

# Representation predicate for list segments

$$\begin{aligned} p \rightsquigarrow \text{MList } L &\equiv \text{match } L \text{ with} \\ &| \text{nil} \Rightarrow \lceil p = \text{null} \rceil \\ &| x :: L' \Rightarrow \exists p'. p \mapsto (x, p') \\ &\quad * p' \rightsquigarrow \text{MList } L' \end{aligned}$$



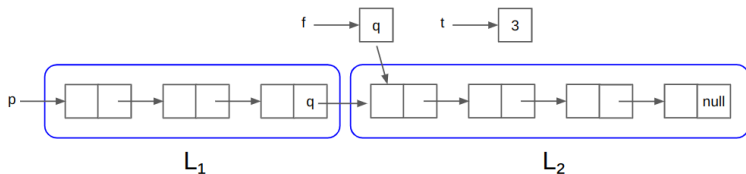
**Exercise:** generalize MList to define  $p \rightsquigarrow \text{MlistSeg } q L$ , where  $L$  denotes the list of items in the list segment from  $p$  (inclusive) to  $q$  (exclusive).

$$\begin{aligned} p \rightsquigarrow \text{MlistSeg } q L &\equiv \text{match } L \text{ with} \\ &| \text{nil} \Rightarrow \lceil p = q \rceil \\ &| x :: L' \Rightarrow \exists p'. p \mapsto (x, p') \\ &\quad * p' \rightsquigarrow \text{MlistSeg } q L' \end{aligned}$$

Remark:

$$p \rightsquigarrow \text{MList } L = p \rightsquigarrow \text{MlistSeg } \text{null } L$$

# Mlength: proof

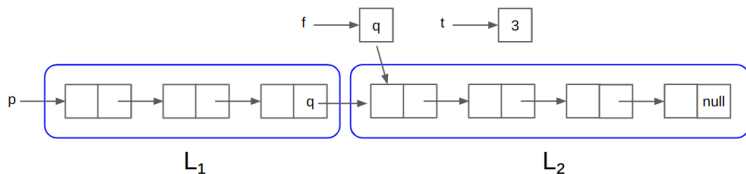


Enter:

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

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

# Mlength: proof



Enter:

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

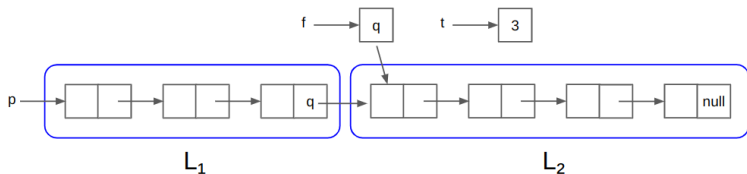
$$\ulcorner = (p \rightsquigarrow \text{MlistSeg } p \text{ nil})$$

Exit:

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

$$(p \rightsquigarrow \text{MlistSeg } \text{null } L) = (p \rightsquigarrow \text{Mlist } L)$$

# Mlength: proof



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

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

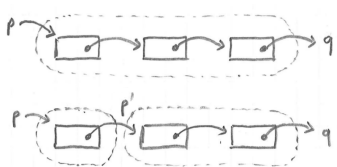
Exit:  $L_1 = L \wedge L_2 = \text{nil} \wedge q = \text{null}$

$$(p \rightsquigarrow \text{MlistSeg } \text{null } L) = (p \rightsquigarrow \text{Mlist } L)$$

Step:  $L_2 = x :: L'_2 \wedge q \neq \text{null} \wedge q.\text{tl} = q'$

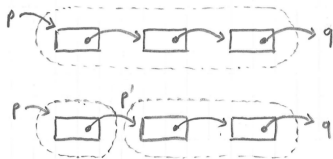
$$\begin{aligned} & \exists q. p \rightsquigarrow \text{MlistSeg } q L_1 * q \mapsto (x, q') \\ & = p \rightsquigarrow \text{MlistSeg } q' (L_1 ++ x :: \text{nil}) \end{aligned}$$

# Splitting rules for list segments



$$p \rightsquigarrow \text{MlistSeg } q (x :: L') = \exists p'. p \mapsto (x, p') * p' \rightsquigarrow \text{MlistSeg } q L'$$

# Splitting rules for list segments

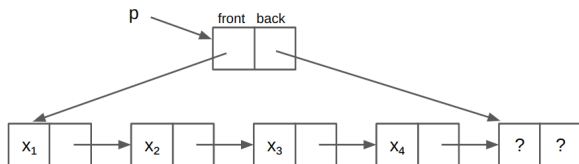


$$p \rightsquigarrow \text{MlistSeg } q (x :: L') = \exists p'. p \mapsto (x, p') * p' \rightsquigarrow \text{MlistSeg } q L'$$



$$p \rightsquigarrow \text{MlistSeg } q (L_1 ++ L_2) = \exists p'. p \rightsquigarrow \text{MlistSeg } p' L_1 * p' \rightsquigarrow \text{MlistSeg } q L_2$$

# An implementation of mutable queues

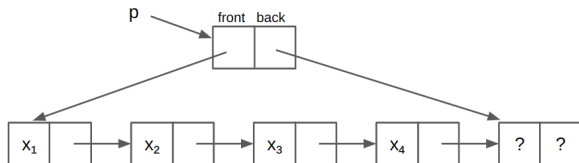


We implement a queue with two pointers to the front and back of the same list segment, with the last cell storing dummy values allowing for less branching (simpler and slightly faster)

```
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



We implement a queue with two pointers to the front and back of the same list segment, with the last cell storing dummy values allowing for less branching (simpler and slightly faster)

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

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

$$p \rightsquigarrow \text{Queue } L \equiv \exists fb. \quad p \mapsto (f, b) * f \rightsquigarrow \text{MlistSeg } b L * b \mapsto (-, -)$$

Alternative for the last cell:  $\exists yq. \quad b \mapsto (y, q)$

# Summary

$$\begin{aligned} p \rightsquigarrow \text{MlistSeg } q L &\equiv \text{match } L \text{ with} \\ &| \text{nil} \Rightarrow \lceil p = q \rceil \\ &| x :: L' \Rightarrow \exists p'. p \mapsto (x, p') \\ &\quad * p' \rightsquigarrow \text{MlistSeg } q L' \end{aligned}$$

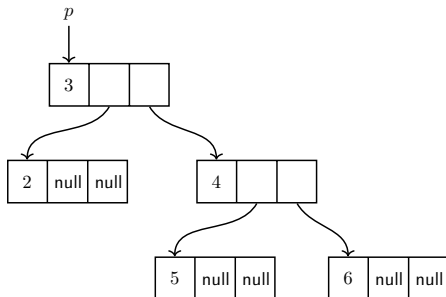
Split and merge of segments:

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

# 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.

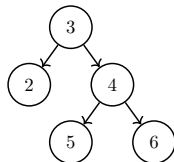
Example:

Node 3

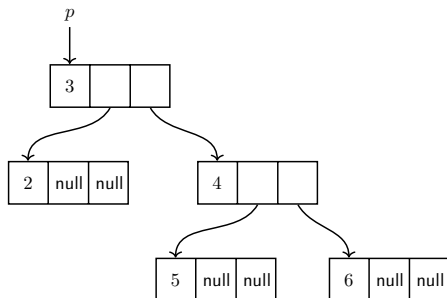
(Node 2 Leaf Leaf)

(Node 4 (Node 5 Leaf Leaf)

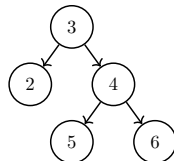
(Node 6 Leaf Leaf))



# Representation predicate for binary trees



$T =$



Representation predicate:

$$p \rightsquigarrow \text{Mtree } T$$

# Representation predicate for binary trees

$$\begin{aligned} p \rightsquigarrow \text{MList } L &\equiv \text{match } L \text{ with} \\ &| \text{nil} \Rightarrow \lceil p = \text{null} \rceil \\ &| x :: L' \Rightarrow \exists p'. \quad p \mapsto (x, p') \\ &\quad * p' \rightsquigarrow \text{MList } L' \end{aligned}$$

**Exercise:** define  $p \rightsquigarrow \text{Mtree } T$ .

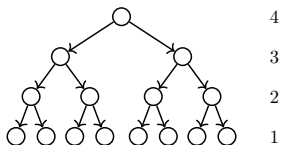
# Representation predicate for binary trees

$$\begin{aligned} p \rightsquigarrow \text{MList } L &\equiv \text{match } L \text{ with} \\ &| \text{nil} \Rightarrow \text{'}p = \text{null'} \\ &| x :: L' \Rightarrow \exists p'. \quad p \mapsto (x, p') \\ &\quad \quad \quad * p' \rightsquigarrow \text{MList } L' \end{aligned}$$

**Exercise:** define  $p \rightsquigarrow \text{Mtree } T$ .

$$\begin{aligned} p \rightsquigarrow \text{Mtree } T &\equiv \text{match } T \text{ with} \\ &| \text{Leaf} \Rightarrow \text{'}p = \text{null'} \\ &| \text{Node } x T_1 T_2 \Rightarrow \exists p_1 p_2. \\ &\quad \quad \quad p \mapsto (x, p_1, p_2) \\ &\quad \quad \quad * p_1 \rightsquigarrow \text{Mtree } T_1 \\ &\quad \quad \quad * p_2 \rightsquigarrow \text{Mtree } T_2 \end{aligned}$$

# Complete binary tree



$$p \rightsquigarrow \text{MtreeDepth } n T$$

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

## Complete binary tree (1/2)

$p \rightsquigarrow \text{Mtree } T \equiv \text{match } T \text{ with}$

- | Leaf  $\Rightarrow$   $\ulcorner p = \text{null} \urcorner$
- | Node  $x T_1 T_2 \Rightarrow \exists p_1 p_2.$ 
  - $p \mapsto (x, p_1, p_2)$
  - \*  $p_1 \rightsquigarrow \text{Mtree } T_1$
  - \*  $p_2 \rightsquigarrow \text{Mtree } T_2$

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

## Complete binary tree (1/2), solution

$$\begin{aligned} p \rightsquigarrow \text{MtreeDepth } n T &\equiv \text{match } T \text{ with} \\ | \text{Leaf} &\Rightarrow \lceil p = \text{null} \wedge n = 0 \rceil \\ | \text{Node } x T_1 T_2 &\Rightarrow \exists p_1 p_2. \lceil n > 0 \rceil * \\ &\quad p \mapsto (x, p_1, p_2) \\ &\quad * p_1 \rightsquigarrow \text{MtreeDepth } (n - 1) T_1 \\ &\quad * p_2 \rightsquigarrow \text{MtreeDepth } (n - 1) T_2 \end{aligned}$$

## Complete binary tree (1/2), solution

$$\begin{aligned} p \rightsquigarrow \text{MtreeDepth } n T &\equiv \text{match } T \text{ with} \\ &| \text{Leaf} \Rightarrow \text{'} p = \text{null} \wedge n = 0 \text{' } \\ &| \text{Node } x T_1 T_2 \Rightarrow \exists p_1 p_2. \text{' } n > 0 \text{' } * \\ &\quad p \mapsto (x, p_1, p_2) \\ &\quad * p_1 \rightsquigarrow \text{MtreeDepth } (n - 1) T_1 \\ &\quad * p_2 \rightsquigarrow \text{MtreeDepth } (n - 1) T_2 \end{aligned}$$

Or:

$$\begin{aligned} p \rightsquigarrow \text{MtreeDepth } n T &\equiv \text{match } n, T \text{ with} \\ &| O, \text{Leaf} \Rightarrow \text{' } p = \text{null' } \\ &| S m, \text{Node } x T_1 T_2 \Rightarrow \exists p_1 p_2. \\ &\quad p \mapsto (x, p_1, p_2) \\ &\quad * p_1 \rightsquigarrow \text{MtreeDepth } m T_1 \\ &\quad * p_2 \rightsquigarrow \text{MtreeDepth } m T_2 \\ &| -, - \Rightarrow \text{'False' } \end{aligned}$$

## Complete binary tree (2/2)

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

## Complete binary tree (2/2)

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

$$p \rightsquigarrow \text{MtreeDepth } n T \equiv p \rightsquigarrow \text{Mtree } T * \text{'depth } n T'$$

**Inductive** `depth` : `int`  $\rightarrow$  `tree`  $\rightarrow$  `Prop` :=

```
| depth_leaf :  
  depth 0 Leaf  
| depth_node :  $\forall n$  x T1 T2,  
  depth n T1  $\rightarrow$   
  depth n T2  $\rightarrow$   
  depth (n+1) (Node x T1 T2).
```

# Complete binary tree of unspecified depth

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

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

# Complete binary tree of unspecified depth

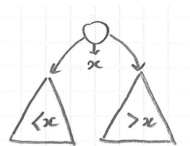
$$p \rightsquigarrow \text{MtreeDepth } n T \equiv (p \rightsquigarrow \text{Mtree } T) * \ulcorner \text{depth } n T \urcorner$$

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

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

- 1  $\exists n. p \rightsquigarrow \text{MtreeDepth } n T$
- 2  $\exists n. (p \rightsquigarrow \text{Mtree } T) * \ulcorner \text{depth } n T \urcorner$
- 3  $(p \rightsquigarrow \text{Mtree } T) * \ulcorner \exists n. \text{depth } n T \urcorner$

## Binary search tree property



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

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

|  $\text{search\_leaf} :$

$\text{search Leaf } \emptyset$

|  $\text{search\_node} : \forall x T1 T2,$

$\text{search } T1 E1 \rightarrow$

$\text{search } T2 E2 \rightarrow$

$\text{foreach (is\_lt } x) E1 \rightarrow$

$\text{foreach (is\_gt } x) E2 \rightarrow$

$\text{search (Node } x T1 T2) (\{x\} \cup E1 \cup E2).$

# Binary search tree predicate

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

# Binary search tree predicate

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

$$p \rightsquigarrow \text{MsearchTree } E \equiv \exists T. p \rightsquigarrow \text{Mtree } T * \text{'search } T \ E'$$

# Binary search tree predicate

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

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

For example, a call “add  $x$   $p$ ” can be specified as follows:

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

# Summary

Common representation predicate for all binary trees:

$$\begin{aligned} p \rightsquigarrow \text{Mtree } T &\equiv \text{match } T \text{ with} \\ &| \text{Leaf} \Rightarrow \text{'}p = \text{null'} \\ &| \text{Node } x T_1 T_2 \Rightarrow \exists p_1 p_2. \\ &\quad p \mapsto (x, p_1, p_2) \\ &\quad * p_1 \rightsquigarrow \text{Mtree } T_1 * p_2 \rightsquigarrow \text{Mtree } T_2 \end{aligned}$$

Invariants are expressed on the pure trees:

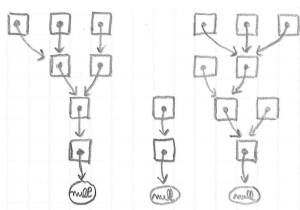
$$p \rightsquigarrow \text{MsearchTree } E \equiv \exists T. p \rightsquigarrow \text{Mtree } T * \text{'search } T E'$$

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

# Chapter 5

## Structures with sharing

# The union-find data structure



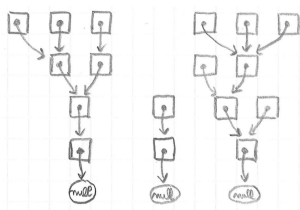
`type node = node ref`

Implements an equivalence relation  $S$  of type:  $\text{loc} \rightarrow \text{loc} \rightarrow \text{Prop}$ .

$S a b \Leftrightarrow a$  and  $b$  are two valid nodes with the same root

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

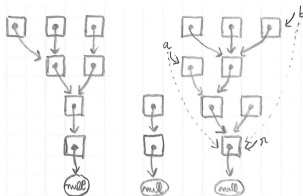
# Representation of union-find cells



$$(p_1 \mapsto q_1) * (p_2 \mapsto q_2) * \dots * (p_n \mapsto q_n)$$
$$= *_{(p_i, q_i) \in G} (p_i \mapsto q_i)$$

where  $G$  is a finite map from locations to locations.

# Invariants of union-find



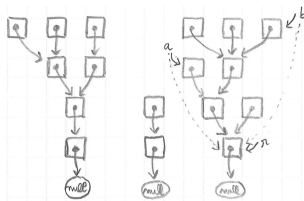
Predicate “ $\text{root } G a r$ ” asserts that in the graph  $G$ , node  $a$  has root  $r$ .

**Inductive**  $\text{root} : \text{fmap } \text{loc } \text{loc} \rightarrow \text{loc} \rightarrow \text{loc} \rightarrow \text{Prop} :=$

|  $\text{root\_init} : \forall G x,$   
|  $\text{binds } G x \text{ null} \rightarrow$   
|  $\text{root } G x x$

|  $\text{root\_step} : \forall G x y r,$   
|  $\text{binds } G x y \rightarrow$   
|  $y \neq \text{null} \rightarrow$   
|  $\text{root } G y r \rightarrow$   
|  $\text{root } G x r.$

# Specification of the union-find structure

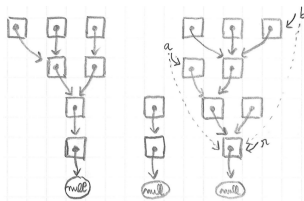


$$\text{UnionFind } S \quad \equiv \quad \exists G. \quad \left( *_{(p,q) \in G} p \mapsto q \right)$$

$$* \ulcorner \forall a \in \text{dom } G. \exists r. \text{root } G a r \urcorner$$

$$* \ulcorner \forall ab. S a b \Leftrightarrow \exists r. \text{root } G a r \wedge \text{root } G b r \urcorner$$

# Specification of the union-find structure



$$\begin{aligned} \text{UnionFind } S &\equiv \exists G. \quad \left( *_{(p,q) \in G} p \mapsto q \right) \\ &* \ulcorner \forall a \in \text{dom } G. \exists r. \text{root } G a r \urcorner \\ &* \ulcorner \forall ab. S a b \Leftrightarrow \exists r. \text{root } G a r \wedge \text{root } G b r \urcorner \end{aligned}$$

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

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

# Summary

Iterated separating conjunction, written  $\ast$ .

For Union-Find:

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

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

# Examples of triples

Example 1:

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

# Examples of triples

Example 1:

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

Example 2:

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

# Examples of triples

Example 1:

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

Example 2:

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

Example 3:

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

# Examples of triples

Example 1:

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

Example 2:

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

Example 3:

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

Example 4:

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

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:

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

- $H$  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:

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

# Specification of operations on memory cells

**Exercise:** specify the primitive operations on references.

`(ref v)`

`(!r)`

`(r := v)`

# Specification of operations on memory cells

**Exercise:** specify the primitive operations on references.

(ref v)

(!r)

(r := v)

Solution:

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

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

# Specification of operations on memory cells

**Exercise:** specify the primitive operations on references.

(ref v)

(!r)

(r := v)

Solution:

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

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

$\forall r v w. \{ r \mapsto w \} (r := v) \{ \lambda \_. (r \mapsto v) \}$

$\forall r v. \{ \exists w. r \mapsto w \} (r := v) \{ \lambda \_. (r \mapsto v) \}$

$\forall r v. \{ r \mapsto - \} (r := v) \{ \lambda \_. (r \mapsto v) \}$

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

# Specification of partial functions

Presentation 1:

$$\forall n. \{ \ulcorner n \geq 0 \urcorner \} (\text{fact } n) \{ \lambda x. \ulcorner x = n! \urcorner \}$$

Presentation 2:

$$\forall n. n \geq 0 \Rightarrow \{ \ulcorner \urcorner \} (\text{fact } n) \{ \lambda x. \ulcorner x = n! \urcorner \}$$

distinguish:

*fact*  $n$ : the program called with argument  $n$

$n!$ : the mathematical quantity

# Specification of operations on arrays

**Exercise:** specify operations on arrays in terms of  $p \rightsquigarrow \text{Array } L$  (assuming `Array` is already defined).

`(Array.get p i)`

`(Array.set p i 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  $i$

$|L|$   $\equiv$  length of  $L$

$i \in \text{dom } L$   $\equiv$   $0 \leq i < |L|$

# Specification of operations on arrays

$$\begin{aligned} \forall i L p \quad i \in \text{dom } L \Rightarrow \{ & p \rightsquigarrow \text{Array } L \} \\ & (\text{Array.get } p \ i) \\ & \{ \lambda x. \text{「} x = L[i] \text{」} * p \rightsquigarrow \text{Array } L \} \end{aligned}$$

$$\begin{aligned} \forall i L p \quad i \in \text{dom } L \Rightarrow \{ & p \rightsquigarrow \text{Array } L \} \\ & (\text{Array.set } p \ i \ v) \\ & \{ \lambda_. p \rightsquigarrow \text{Array } (L[i := v]) \} \end{aligned}$$

# Specification of operations on arrays

$$\begin{aligned} \forall i L p \quad i \in \text{dom } L &\Rightarrow \{p \rightsquigarrow \text{Array } L\} \\ &(\text{Array.get } p \ i) \\ &\{\lambda x. \ulcorner x = L[i] \urcorner * p \rightsquigarrow \text{Array } L\} \end{aligned}$$

$$\begin{aligned} \forall i L p \quad i \in \text{dom } L &\Rightarrow \{p \rightsquigarrow \text{Array } L\} \\ &(\text{Array.set } p \ i \ v) \\ &\{\lambda \_. p \rightsquigarrow \text{Array } (L[i := v])\} \end{aligned}$$

$$\begin{aligned} \forall p L & \{p \rightsquigarrow \text{Array } L\} \\ &(\text{Array.length } p) \\ &\{\lambda n. \ulcorner n = |L| \urcorner * p \rightsquigarrow \text{Array } L\} \end{aligned}$$

$$\begin{aligned} \forall n v \quad n \geq 0 &\Rightarrow \{\ulcorner \urcorner\} \\ &(\text{Array.create } n \ v) \\ &\{\lambda p. \exists L. (p \rightsquigarrow \text{Array } L) * \ulcorner |L| = n \urcorner \\ &\quad * \ulcorner \forall i \in \text{dom } L. L[i] = v \urcorner\} \end{aligned}$$

# 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:

$$\forall m. H m \Rightarrow \exists v. \exists m'. \langle t, m \rangle \Downarrow \langle v, m' \rangle \wedge H'[v/x] m'$$

(assuming a deterministic semantics)

# 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:

$$\forall m. H m \Rightarrow \exists v. \exists m'. \langle t, m \rangle \Downarrow \langle v, m' \rangle \wedge H'[v/x] m'$$

(assuming a deterministic semantics)

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

# 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:

$$\forall m. H m \Rightarrow \exists v. \exists m'. \langle t, m \rangle \Downarrow \langle v, m' \rangle \wedge H'[v/x] m'$$

(assuming a deterministic semantics)

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

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

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

## 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:

$$m_1 \perp m_2 \equiv (\text{dom } m_1 \cap \text{dom } m_2 = \emptyset)$$

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

## 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:

$$m_1 \perp m_2 \equiv (\text{dom } m_1 \cap \text{dom } m_2 = \emptyset)$$

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

$$\forall m_1 m_2. \begin{cases} H m_1 \\ m_1 \perp m_2 \end{cases} \Rightarrow \exists v. \exists m'_1. \begin{cases} \langle t, m_1 \uplus m_2 \rangle \Downarrow \langle v, m'_1 \uplus m_2 \rangle \\ Q v m'_1 \\ m'_1 \perp m_2 \end{cases}$$

# 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:

$$\{\ulcorner \urcorner\} (\text{myref } x) \{\lambda r. r \mapsto x * \exists s. s \mapsto r\}$$

To:

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

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.

$$GC \equiv \exists H. H$$

(sometimes just called *True*, because it is equivalent to  $\lambda m. \text{True}$ , but it looks too much like 'True'.)

## Interpretation of triples (3/3), revisited

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

$$GC \equiv \exists H. H$$

(sometimes just called *True*, because it is equivalent to  $\lambda m. \text{True}$ , but it looks too much like 'True'.)

### Definition (Separation Logic Triple)

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

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

# Summary

Separation Logic triple:

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

Specification of a function:

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

Specification of primitive functions:

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

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

$$\forall r v. \{r \mapsto -\} (\mathbf{r} := v) \{\lambda \_. (r \mapsto v)\}$$

Interpretation of triples: see definition.

# Chapter 7

## The Frame Rule

# Preservation of independent state

We have:

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

We also have:

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

# Preservation of independent state

We have:

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

We also have:

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

More generally:

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

# The frame rule

Principle: a triple remains valid when both the pre-condition and the post-condition are extended with a same heap predicate.

General form:

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

# Frame rule and allocation

We have:

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

By the frame rule, we have:

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

# Frame rule and allocation

We have:

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

By the frame rule, we have:

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

This post-condition ensures  $r \neq s$ .

The reference cell  $r$  is thus guaranteed to be distinct from any cell that might exist prior to the allocation of  $r$ .

The end!