

Separation Logic 3/4

Jean-Marie Madiot

Inria Paris

February 12, 2025

part 3/4 : some initial material from Arthur Charguéraud

Chapter 12

Loops in Separation Logic

Verification of a for-loop

```
let facto n =  
  let r = ref 1 in  
  for i = 2 to n do  
    let v = !r in  
    r := v * i;  
  done;  
  !r
```

Verification of a for-loop

```
let facto n =  
  let r = ref 1 in  
  for i = 2 to n do  
    let v = !r in  
    r := v * i;  
  done;  
  !r
```

Before the loop:

$$r \mapsto 1$$

At each iteration:

$$\text{from } r \mapsto (i-1)! \quad \text{to} \quad r \mapsto i!$$

After the loop:

$$r \mapsto n!$$

Verification of a for-loop

```
let facto n =  
  let r = ref 1 in  
  for i = 2 to n do  
    let v = !r in  
    r := v * i;  
  done;  
  !r
```

Before the loop:

$$r \mapsto 1$$

At each iteration:

$$\text{from } r \mapsto (i - 1)! \quad \text{to} \quad r \mapsto i!$$

After the loop:

$$r \mapsto n!$$

Loop invariant ($I : \text{int} \rightarrow \text{Hprop}$) that applies for any $i \in [2, n + 1]$:

$$I i \quad \equiv \quad r \mapsto (i - 1)!$$

Reasoning rule for for-loops

Reasoning rule for the case $a \leq b$:

$$\frac{\forall i \in [a, b]. \quad \{I\ i\} \ t \ \{\lambda_. I\ (i + 1)\}}{\{I\ a\} \ (\text{for } i = a \text{ to } b \text{ do } t) \ \{\lambda_. I\ (b + 1)\}}$$

Reasoning rule for for-loops

Reasoning rule for the case $a \leq b$:

$$\frac{\forall i \in [a, b]. \quad \{I\ i\} \ t \ \{\lambda_{-}. I\ (i + 1)\}}{\{I\ a\} \ (\text{for } i = a \text{ to } b \text{ do } t) \ \{\lambda_{-}. I\ (b + 1)\}}$$

General rule, also covering the case $a > b$:

$$\frac{\forall i \in [a, b]. \quad \{I\ i\} \ t \ \{\lambda_{-}. I\ (i + 1)\}}{\{I\ a\} \ (\text{for } i = a \text{ to } b \text{ do } t) \ \{\lambda_{-}. I\ (\max a\ (b + 1))\}}$$

Reasoning rule for while loops: partial correctness

The loop invariant I describes the state between every iterations.

The post-condition J describes the state after the evaluation of t_1 .

$$\frac{\{I\} t_1 \{J\} \quad \{J \text{ true}\} t_2 \{\lambda_. I\}}{\{I\} (\text{while } t_1 \text{ do } t_2) \{\lambda_. J \text{ false}\}}$$

where $(I : \text{Hprop})$ and $(J : \text{bool} \rightarrow \text{Hprop})$.

Reasoning rule for while loops: partial correctness

The loop invariant I describes the state between every iterations.

The post-condition J describes the state after the evaluation of t_1 .

$$\frac{\{I\} t_1 \{J\} \quad \{J \text{ true}\} t_2 \{\lambda_. I\}}{\{I\} (\text{while } t_1 \text{ do } t_2) \{\lambda_. J \text{ false}\}}$$

where $(I : \text{Hprop})$ and $(J : \text{bool} \rightarrow \text{Hprop})$.

For total correctness: parameterize the invariant with a measure.

Reasoning rule for while loops

We focus on a different approach that:

- supports **total correctness** through the meta-logic;
- allows to apply the **frame rule** during iterations.

Reasoning rule for while loops

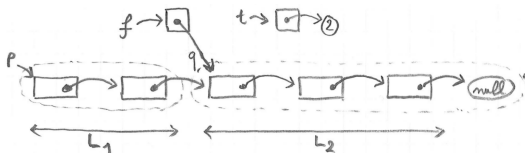
We focus on a different approach that:

- supports **total correctness** through the meta-logic;
- allows to apply the **frame rule** during iterations.

Prove a triple $\{H\} (\text{while } t_1 \text{ do } t_2) \{Q\}$ by induction, using:

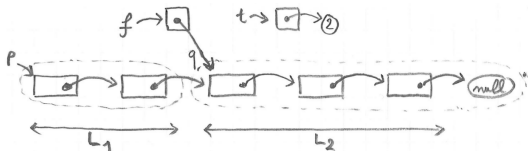
$$\frac{\{H\} (\text{if } t_1 \text{ then } (t_2 ; (\text{while } t_1 \text{ do } t_2)) \text{ else } ()) \{Q\}}{\{H\} (\text{while } t_1 \text{ do } t_2) \{Q\}}$$

Length with a while loop



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

Length with a while loop: induction



We discard L_1 and prove by induction on L_2 that for all n and q :

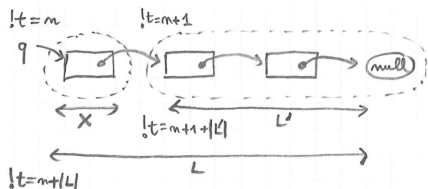
$$\begin{aligned} & \{q \rightsquigarrow \text{MList } L_2 * f \mapsto q * t \mapsto n\} \\ & (\text{while } !f \neq \text{null} \text{ do incr } t; f := (!f).tl; \text{done}) \\ & \{q \rightsquigarrow \text{MList } L_2 * f \mapsto \text{null} * t \mapsto (n + \text{length } L_2)\} \end{aligned}$$

The loop unfolds to:

```
if !f != null
  then (incr t; f := (!f).tl; while .. do .. done)
  else ()
```

Exercise: give a proof sketch for the induction for `mlength`.

Length with a while loop: use of frame



$q \rightsquigarrow \text{MList } L_2$	$* f \mapsto q$	$* t \mapsto n$	begin
$q \mapsto \{x; q'\} * q' \rightsquigarrow \text{MList } L'_2$	$* f \mapsto q$	$* t \mapsto n$	unfold
$q \mapsto \{x; q'\} * q' \rightsquigarrow \text{MList } L'_2$	$* f \mapsto q$	$* t \mapsto n + 1$	increment
<u>$q \mapsto \{x; q'\}$</u> $* q' \rightsquigarrow \text{MList } L'_2$	$* f \mapsto q'$	$* t \mapsto n + 1$	shift head
$q \mapsto \{x; q'\} * q' \rightsquigarrow \text{MList } L'_2$	$* f \mapsto \text{null}$	$* t \mapsto n + 1 + L'_2 $	<u>frame</u> +Ind.hyp.
$q \rightsquigarrow \text{MList } L_2$	$* f \mapsto \text{null}$	$* t \mapsto n + L_2 $	fold

Chapter 13

Aliasing and local state

Functions with aliasing: swap

```
let swap r s =  
  let a = !r in  
  let b = !s in  
  r := b;  
  s := a
```

Exercise: Find three useful specifications for swap:

- 1 a specification for non-aliased (distinct) arguments,
- 2 a specification for aliased (equal) arguments,
- 3 (bonus) a most-general specification.

Functions with aliasing: 3 specifications for swap

Specification 1:

$$\forall rsnm. \{(r \mapsto n) * (s \mapsto m)\} (\text{swap } r \ s) \{\lambda_. (r \mapsto m) * (s \mapsto n)\}$$

Functions with aliasing: 3 specifications for swap

Specification 1:

$$\forall rsnm. \{(r \mapsto n) * (s \mapsto m)\} (\text{swap } r \ s) \{\lambda_. (r \mapsto m) * (s \mapsto n)\}$$

Specification 2:

$$\forall rsn. \{r = s * (r \mapsto n)\} (\text{swap } r \ s) \{\lambda_. r \mapsto n\}$$

or simply:

$$\forall rn. \{r \mapsto n\} (\text{swap } r \ r) \{\lambda_. r \mapsto n\}$$

Functions with aliasing: 3 specifications for swap

Specification 1:

$$\forall rsnm. \{(r \mapsto n) * (s \mapsto m)\} (\text{swap } r \ s) \{\lambda_. (r \mapsto m) * (s \mapsto n)\}$$

Specification 2:

$$\forall rsn. \{r = s' * (r \mapsto n)\} (\text{swap } r \ s) \{\lambda_. r \mapsto n\}$$

or simply:

$$\forall rn. \{r \mapsto n\} (\text{swap } r \ r) \{\lambda_. r \mapsto n\}$$

Specification 3:

$$\begin{aligned} \forall rsM. r, s \in \text{dom } M \Rightarrow & \{(\otimes)_{(p,n) \in M} p \mapsto n\} \\ & (\text{swap } r \ s) \\ & \{\lambda_. (\otimes)_{(p,n) \in (M[r:=M[s]] [s:=M[r]])} p \mapsto n\} \end{aligned}$$

alternatively with a no-offset version of Cells, $\forall rsM. r, s \in \text{dom } M \Rightarrow$

$$\{\text{Cells}'(M)\} (\text{swap } r \ s) \{\lambda_. \text{Cells}'(M[r := M[s]] [s := M[r]])\}$$

Function with local state

Exercise: what is the specification of f in the following program?

```
let r = ref 3
let f () =
  incr r
```

Then, show that the code below returns 5.

```
f();
f();
!r
```

Function with local state

Exercise: what is the specification of f in the following program?

```
let r = ref 3
let f () =
  incr r
```

Then, show that the code below returns 5.

```
f();
f();
!r
```

Specification:

$$\forall n. \{r \mapsto n\} (f \ ()) \{\lambda_. r \mapsto n + 1\}$$

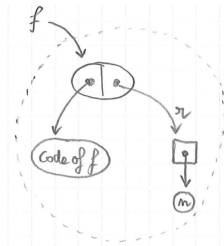
Successive states:

$$r \mapsto 3 \qquad r \mapsto 4 \qquad r \mapsto 5$$

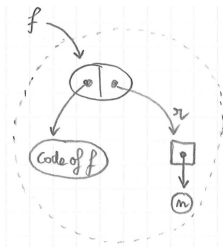
Counter function: code

```
let mkcounter () =  
  let r = ref 0 in  
  (fun () -> incr r; !r)
```

```
let c = mkcounter() in  
let x = c() in  
let y = c() in  
assert (x = 1 && y = 2)
```

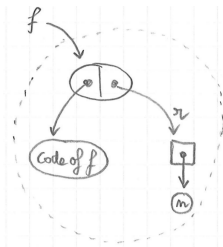


Counter function: specification



$$f \rightsquigarrow \text{Count } n \equiv \exists r. (r \mapsto n) \\ * \text{ '}\forall i. \{r \mapsto i\} (f ()) \{\lambda x. \text{'}x = i + 1\text{' } * (r \mapsto i + 1)\}\text{'}$$

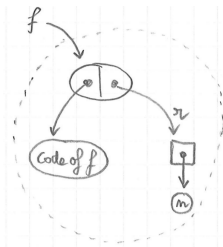
Counter function: specification



$$f \rightsquigarrow \text{Count } n \equiv \exists r. (r \mapsto n) \\ * \text{ '}\forall i. \{r \mapsto i\} (f ()) \{\lambda x. \text{'}x = i + 1\text{'} * (r \mapsto i + 1)\}\text{'}$$

Exercise: specify a counter function, only in terms of $f \rightsquigarrow \text{Count } n$.

Counter function: specification



$$f \rightsquigarrow \text{Count } n \equiv \exists r. (r \mapsto n) \\ * \text{ '}\forall i. \{r \mapsto i\} (f ()) \{\lambda x. \text{'}x = i + 1\text{'} * (r \mapsto i + 1)\}\text{'}$$

Exercise: specify a counter function, only in terms of $f \rightsquigarrow \text{Count } n$.

$$\{\text{'}\}\ (\text{mkcounter}()) \{\lambda f. f \rightsquigarrow \text{Count } 0\}$$

$$\forall f i. \quad \{f \rightsquigarrow \text{Count } i\} (f ()) \{\lambda x. \text{'}x = i + 1\text{'} * f \rightsquigarrow \text{Count } (i + 1)\}$$

Chapter 14

Basic higher-order functions

Apply

```
let apply f x =  
  f x
```

Specification:

$$\begin{aligned} \forall f x H Q. \quad & \{H\} (f x) \{Q\} \\ \Rightarrow & \{H\} (\text{apply } f x) \{Q\} \end{aligned}$$

Apply

```
let apply f x =  
  f x
```

Specification:

$$\begin{aligned} \forall f x H Q. \quad & \{H\} (f x) \{Q\} \\ \Rightarrow & \{H\} (\text{apply } f x) \{Q\} \end{aligned}$$

This is equivalent to the form below, which involves nested triples:

$$\forall f x H Q. \quad \{H * \text{「} \{H\} (f x) \{Q\} \text{」}\} (\text{apply } f x) \{Q\}$$

Apply on a reference

```
let refapply r f =  
  r := f !r
```

Exercise: give two specifications for the function `refapply`.

In the first, assume `f` to be pure, and introduce a predicate $P\ x\ y$.

In the second, assume that `f` also modifies the state from H to H' .

Apply on a reference

```
let refapply r f =  
  r := f !r
```

Exercise: give two specifications for the function `refapply`.

In the first, assume `f` to be pure, and introduce a predicate $P\ x\ y$.

In the second, assume that `f` also modifies the state from H to H' .

$$\begin{aligned} \forall r f x P. \quad & \{\ulcorner \urcorner\} (f\ x) \{\lambda y. \ulcorner P\ x\ y \urcorner\} \\ \Rightarrow \quad & \{r \mapsto x\} (\text{refapply } r\ f) \{\lambda _. \exists y. \ulcorner P\ x\ y \urcorner * r \mapsto y\} \end{aligned}$$

Apply on a reference

```
let refapply r f =  
  r := f !r
```

Exercise: give two specifications for the function `refapply`.

In the first, assume `f` to be pure, and introduce a predicate $P\ x\ y$.

In the second, assume that `f` also modifies the state from H to H' .

$$\begin{aligned} \forall r f x P. \quad & \{ \ulcorner \urcorner \} (f\ x) \{ \lambda y. \ulcorner P\ x\ y \urcorner \} \\ \Rightarrow \quad & \{ r \mapsto x \} (\text{refapply } r\ f) \{ \lambda _. \exists y. \ulcorner P\ x\ y \urcorner * r \mapsto y \} \end{aligned}$$

$$\begin{aligned} \forall r f x H H' P. \quad & \{ H \} (f\ x) \{ \lambda y. \ulcorner P\ x\ y \urcorner * H' \} \\ \Rightarrow \quad & \{ (r \mapsto x) * H \} \\ & (\text{refapply } r\ f) \\ & \{ \lambda _. \exists y. \ulcorner P\ x\ y \urcorner * (r \mapsto y) * H' \} \end{aligned}$$

Function twice

```
let twice f =  
  f(); f()
```

Specification:

$$\begin{aligned} \forall f H' Q. \quad & \{H\} (f ()) \{ \lambda _. H' \} \\ & \wedge \quad \{H'\} (f ()) \{Q\} \\ \Rightarrow & \{H\} (\text{twice } f) \{Q\} \end{aligned}$$

Function repeat

```
let repeat n f =  
  for i = 0 to n-1 do  
    f()  
done
```

Exercise: specify repeat, using an invariant I , of type $\text{int} \rightarrow \text{Hprop}$.

Function repeat

```
let repeat n f =  
  for i = 0 to n-1 do  
    f()  
done
```

Exercise: specify repeat, using an invariant I , of type $\text{int} \rightarrow \text{Hprop}$.

$$\begin{aligned} \forall n f I. \quad & (\forall i \in [0, n). \quad \{I\ i\} (f\ ()) \{\lambda_. I\ (i + 1)\}) \\ \Rightarrow \quad & \{I\ 0\} (\text{repeat } n\ f) \{\lambda_. I\ n\} \end{aligned}$$

The premise consists of a family of hypotheses describing the behavior of applications of f to particular arguments.

Chapter 15

Higher order iteration

Iteration over a pure list



For pedagogical purposes, “pure lists” live outside the heap and need no representation predicate. (In practice, most (but not all) lists are allocated on the heap.)



```
let rec iter f l =  
  match l with  
  | [] -> ()  
  | x::t -> f x; iter f t
```

Exercise: specify `iter`, using an invariant I , of type $\text{list } \alpha \rightarrow \text{Hprop}$.

Iteration over a pure list



For pedagogical purposes, “pure lists” live outside the heap and need no representation predicate. (In practice, most (but not all) lists are allocated on the heap.)



```
let rec iter f l =  
  match l with  
  | [] -> ()  
  | x::t -> f x; iter f t
```

Exercise: specify `iter`, using an invariant I , of type $\text{list } \alpha \rightarrow \text{Hprop}$.

$$\begin{aligned} \forall f l I. \quad & (\forall x k. \{I\ k\} (f\ x) \{\lambda_. I\ (k\&x)\}) \\ \Rightarrow & \{I\ \text{nil}\} (\text{iter } f\ l) \{\lambda_. I\ l\} \end{aligned}$$

where $k\&x \equiv k \mathbin{++} (x :: \text{nil})$.

Length using iter

$$\Rightarrow (\forall xk. \{I\ k\} (f\ x) \{\lambda_. I\ (k\&x)\}) \\ \{I\ \text{nil}\} (\text{iter}\ f\ l) \{\lambda_. I\ l\}$$

```
let length l =  
  let r = ref 0 in  
  iter (fun x -> incr r) l;  
  !r
```

Exercise: give the instantiation of the invariant I for `iter`;
then, write the specialization of the specification of `iter` to I and to
`(fun x -> incr r)`; finally, check that the premise is provable.

Length using iter

$$\begin{aligned} & (\forall x k. \{I\ k\} (f\ x) \{\lambda_. I\ (k\&x)\}) \\ \Rightarrow & \{I\ \text{nil}\} (\text{iter}\ f\ l) \{\lambda_. I\ l\} \end{aligned}$$

```
let length l =  
  let r = ref 0 in  
  iter (fun x -> incr r) l;  
  !r
```

Exercise: give the instantiation of the invariant I for `iter`;
then, write the specialization of the specification of `iter` to I and to
(`fun x -> incr r`); finally, check that the premise is provable.

Invariant: $I \equiv \lambda k. r \mapsto |k|$.

$$\begin{aligned} & (\forall x k. \{r \mapsto |k|\} (\text{incr}\ r) \{\lambda_. r \mapsto |k| + 1\}) \\ \Rightarrow & \{r \mapsto 0\} (\text{iter}\ f\ l) \{\lambda_. r \mapsto |l|\} \end{aligned}$$

Sum using iter

$$\begin{aligned} & (\forall x k. \{I\ k\} (f\ x) \{\lambda_. I\ (k\&x)\}) \\ \Rightarrow & \{I\ \text{nil}\} (\text{iter}\ f\ l) \{\lambda_. I\ l\} \end{aligned}$$

```
let sum l =  
  let r = ref 0 in  
  iter (fun x -> r := !r + x) l;  
  !r
```

Exercise: give the invariant I involved in the above call to `iter`.

Sum using iter

$$\Rightarrow (\forall x k. \{I\ k\} (f\ x) \{\lambda_. I\ (k \& x)\}) \\ \{I\ \text{nil}\} (\text{iter}\ f\ l) \{\lambda_. I\ l\}$$

```
let sum l =  
  let r = ref 0 in  
  iter (fun x -> r := !r + x) l;  
  !r
```

Exercise: give the invariant I involved in the above call to `iter`.

$$I \equiv \lambda k. r \mapsto \text{Sum } k$$

where:

$$\text{Sum } k \equiv \text{Fold } (+) 0\ k$$

Constraints over the items

$$\begin{aligned} & (\forall x k. \{I\ k\} (f\ x) \{\lambda_. I\ (k\&x)\}) \\ \Rightarrow & \{I\ \text{nil}\} (\text{iter}\ f\ l) \{\lambda_. I\ l\} \end{aligned}$$

Given a list $x_1 :: x_2 :: \dots :: x_n :: \text{nil}$, let us compute:

$$\frac{10}{x_1} + \frac{10}{x_2} + \dots + \frac{10}{x_n}$$

```
iter (fun x -> r := !r + 10 / x) [2; -3; 4]
```

Constraints over the items

$$\begin{aligned} & (\forall x k. \{I\ k\} (f\ x) \{\lambda_. I\ (k\&x)\}) \\ \Rightarrow & \{I\ \text{nil}\} (\text{iter}\ f\ l) \{\lambda_. I\ l\} \end{aligned}$$

Given a list $x_1 :: x_2 :: \dots :: x_n :: \text{nil}$, let us compute:

$$\frac{10}{x_1} + \frac{10}{x_2} + \dots + \frac{10}{x_n}$$

```
iter (fun x -> r := !r + 10 / x) [2; -3; 4]
```

The above specification of `iter` is too weak. More general specification:

$$\begin{aligned} \forall f\ l. & \quad (\forall x k. \textcolor{red}{x} \in \textcolor{red}{l} \Rightarrow \{I\ k\} (f\ x) \{\lambda_. I\ (k\&x)\}) \\ \Rightarrow & \{I\ \text{nil}\} (\text{iter}\ f\ l) \{\lambda_. I\ l\} \end{aligned}$$

Constraints over the items, in order

$$\begin{aligned} \forall f l l. \quad & (\forall x k. x \in l \Rightarrow \{I k\} (f x) \{\lambda_. I (k \& x)\}) \\ \Rightarrow & \{I \text{nil}\} (\text{iter } f l) \{\lambda_. I l\} \end{aligned}$$

Given a list $x_1 :: x_2 :: \dots :: x_n :: \text{nil}$, let us compute:

$$\frac{\frac{\frac{10}{0+x_1} + x_2}{\dots} + x_n}{10}$$

```
iter (fun x -> r := 10 / (!r + x)) [2; -3; 4]
```

Constraints over the items, in order

$$\begin{aligned} \forall f l. \quad & (\forall x k. x \in l \Rightarrow \{I k\} (f x) \{\lambda_. I (k \& x)\}) \\ \Rightarrow & \{I \text{nil}\} (\text{iter } f l) \{\lambda_. I l\} \end{aligned}$$

Given a list $x_1 :: x_2 :: \dots :: x_n :: \text{nil}$, let us compute:

$$\frac{10}{\frac{\frac{10}{0+x_1} + x_2}{\dots} + x_n}$$

```
iter (fun x -> r := 10 / (!r + x)) [2; -3; 4]
```

The above specification of `iter` is too weak. Most-general specification:

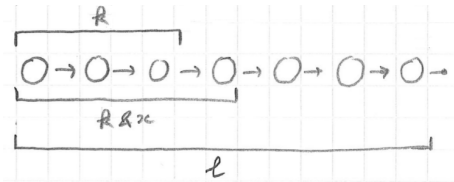
$$\begin{aligned} \forall f l. \quad & (\forall x k s. l = k \& x :: s \Rightarrow \{I k\} (f x) \{\lambda_. I (k \& x)\}) \\ \Rightarrow & \{I \text{nil}\} (\text{iter } f l) \{\lambda_. I l\} \end{aligned}$$

Verification of iter

$$\begin{aligned} & (\forall x k. \{I k\} (f x) \{\lambda_. I (k \& x)\}) \\ \Rightarrow & \{I \text{nil}\} (\text{iter } f l) \{\lambda_. I l\} \end{aligned}$$

```
let rec iter f l =  
  match l with  
  | [] -> ()  
  | x::t -> f x; iter f t
```

How to prove that the code satisfies its specification?



Verification of iter: generalized principle

Assume:

$$\forall xk. \{I\ k\} (f\ x) \{\lambda_. I\ (k\&x)\}$$

Prove:

$$\{I\ \text{nil}\} (\text{iter}\ f\ l) \{\lambda_. I\ l\}$$

Verification of iter: generalized principle

Assume:

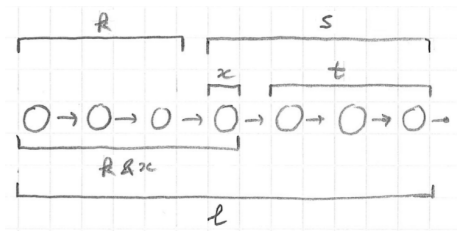
$$\forall xk. \{I k\} (f x) \{\lambda_. I (k \& x)\}$$

Prove:

$$\{I \text{nil}\} (\text{iter } f l) \{\lambda_. I l\}$$

Proof by induction over a generalized statement:

$$\forall lk. \{I k\} (\text{iter } f l) \{\lambda_. I (k ++ l)\}$$



Verification of iter: induction

```
let rec iter f s =  
  match s with  
  | [] -> ()  
  | x::t -> f x; iter f t
```

Assume: $\forall xk. \{I\ k\} (f\ x) \{\lambda_. I\ (k\&x)\}$

Prove: $\forall kl. \{I\ k\} (\text{iter } f\ l) \{\lambda_. I\ (k\&l)\}$

By induction on l :

- Case $l = \text{nil}$. Goal is: $\{I\ k\} (\text{iter } f\ \text{nil}) \{\lambda_. I\ (k\&\text{nil})\}$.
This triple simplifies to: $\{I\ k\} () \{\lambda_. I\ k\}$, which is correct.

Verification of iter: induction

```
let rec iter f s =  
  match s with  
  | [] -> ()  
  | x::t -> f x; iter f t
```

Assume: $\forall xk. \{I\ k\} (f\ x) \{\lambda_. I\ (k\&x)\}$

Prove: $\forall kl. \{I\ k\} (\text{iter } f\ l) \{\lambda_. I\ (k\&l)\}$

By induction on l :

- Case $l = \text{nil}$. Goal is: $\{I\ k\} (\text{iter } f\ \text{nil}) \{\lambda_. I\ (k\&\text{nil})\}$.
This triple simplifies to: $\{I\ k\} () \{\lambda_. I\ k\}$, which is correct.
- Case $l = x :: t$. Goal is: $\{I\ k\} (\text{iter } f\ (x :: t)) \{\lambda_. I\ (k\&(x :: t))\}$.

HYPOTHESIS-ON-F

INDUCTION-HYPOTHESIS

$$\frac{\frac{\{I\ k\} (f\ x) \{\lambda_. I\ (k\&x)\}}{\{I\ k\} (f\ x; \text{iter } f\ t) \{\lambda_. I\ ((k\&x)\&t)\}} \quad \frac{\{I\ (k\&x)\} (\text{iter } f\ t) \{\lambda_. I\ ((k\&x)\&t)\}}{\{I\ k\} (f\ x; \text{iter } f\ t) \{\lambda_. I\ ((k\&x)\&t)\}} \text{SEQ}$$

Invariant on remaining items

$$(\forall x k. \{I\ k\} (f\ x) \{\lambda_. I\ (k\&x)\}) \Rightarrow \{I\ \text{nil}\} (\text{iter}\ f\ l) \{\lambda_. I\ l\}$$

$$(\forall \dots \{\dots\} (f\ x) \{\lambda_. \dots\}) \Rightarrow \{I'\ l\} (\text{iter}\ f\ l) \{\lambda_. I'\ \text{nil}\}$$

Exercise:

- specify `iter` using an invariant that depends on the list of items remaining to process, instead of on the list of items already processed.
- prove the old specification derivable from the new one,
- prove the new specification derivable from the old (most general) one.

Invariant on remaining items

$$(\forall xk. \{I\ k\} (f\ x) \{\lambda_. I\ (k\&x)\}) \Rightarrow \{I\ \text{nil}\} (\text{iter}\ f\ l) \{\lambda_. I\ l\}$$

$$(\forall\ldots \{\ldots\} (f\ x) \{\lambda_. \ldots\}) \Rightarrow \{I'\ l\} (\text{iter}\ f\ l) \{\lambda_. I'\ \text{nil}\}$$

Exercise:

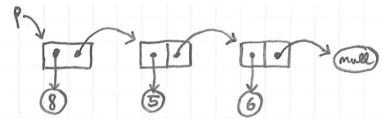
- specify `iter` using an invariant that depends on the list of items remaining to process, instead of on the list of items already processed.
- prove the old specification derivable from the new one,
- prove the new specification derivable from the old (most general) one.

$$(\forall xs. \{I'\ (x :: s)\} (f\ x) \{\lambda_. I'\ s\}) \Rightarrow \{I'\ l\} (\text{iter}\ f\ l) \{\lambda_. I'\ \text{nil}\}$$

$$I'\ s \equiv \exists k. 'l = k ++ s' * I\ k$$

$$I\ k \equiv \exists s. 'l = k ++ s' * I'\ s$$

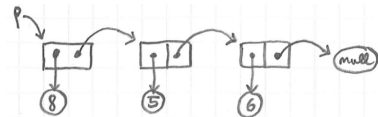
Iterating over a mutable list



```
let rec miter f p =  
  if p == null  
  then ()  
  else (f p.hd; miter f p.tl)
```

Iterating over a mutable list

$$\begin{aligned} \forall flI. \quad & (\forall xk. \{Ik\} (f x) \{\lambda_. I(k\&x)\}) \\ \Rightarrow \quad & \{I \text{ nil}\} (\text{iter } f l) \{\lambda_. Il\} \end{aligned}$$



Specification:

$$\begin{aligned} \forall fpIl. \quad & (\forall xk. \{Ik\} (f x) \{\lambda_. I(k\&x)\}) \\ \Rightarrow \quad & \{p \rightsquigarrow \text{MList } l * I \text{ nil}\} (\text{miter } f p) \{\lambda_. p \rightsquigarrow \text{MList } l * Il\} \end{aligned}$$

Remark: calls to f will not modify the structure of the list while iterating.

Summary

Simplified:

$$\begin{aligned} & (\forall x k. \{I k\} (f x) \{\lambda_. I (k \& x)\}) \\ \Rightarrow & \{I \text{ nil}\} (\text{iter } f l) \{\lambda_. I l\} \end{aligned}$$

Order-irrelevant:

$$\begin{aligned} & (\forall x k. x \in l \Rightarrow \{I k\} (f x) \{\lambda_. I (k \& x)\}) \\ \Rightarrow & \{I \text{ nil}\} (\text{iter } f l) \{\lambda_. I l\} \end{aligned}$$

Most-general:

$$\begin{aligned} & (\forall x k s. l = k \dashv\vdash x :: s \Rightarrow \{I k\} (f x) \{\lambda_. I (k \& x)\}) \\ \Rightarrow & \{I \text{ nil}\} (\text{iter } f l) \{\lambda_. I l\} \end{aligned}$$

Using remaining items (already most general):

$$(\forall x s. \{I' (x :: s)\} (f x) \{\lambda_. I' s\}) \Rightarrow \{I' l\} (\text{iter } f l) \{\lambda_. I' \text{ nil}\}$$

Extension to mutable lists:

$$\begin{aligned} & (\forall x k s. l = k \dashv\vdash x :: s \Rightarrow \{I k\} (f x) \{\lambda_. I (k \& x)\}) \\ \Rightarrow & \{p \rightsquigarrow \text{MList } l * I \text{ nil}\} (\text{miter } f p) \{\lambda_. p \rightsquigarrow \text{MList } l * I l\} \end{aligned}$$

Break?

Chapter 16

Other classic higher-order functions

Fold-left

```
let rec fold_left f a l =  
  match l with  
  | [] -> a  
  | x::k -> fold_left f (f a x) k
```

Example:

$$\text{fold_left } f \ a \ [6 :: 4 :: 7] = f (f (f \ a \ 6) 4) 7$$

Fold-left

```
let rec fold_left f a l =  
  match l with  
  | [] -> a  
  | x::k -> fold_left f (f a x) k
```

Example:

$$\text{fold_left } f \ a \ [6 :: 4 :: 7] = f (f (f \ a \ 6) 4) 7$$

Specification:

$$\begin{aligned} \forall f a l J. \quad & (\forall x i k. \{J \ i \ k\} (f \ i \ x) \{\lambda j. J \ j \ (k \& x)\}) \\ \Rightarrow \quad & \{J \ a \ \text{nil}\} (\text{fold_left } f \ a \ l) \{\lambda b. J \ b \ l\} \end{aligned}$$

Application of fold-left

$$\begin{aligned} \forall f a l J. \quad & (\forall x i k. \{J i k\} (f i x) \{\lambda j. J j (k \& x)\}) \\ \Rightarrow & \{J a \text{ nil}\} (\text{fold_left } f a l) \{\lambda b. J b l\} \end{aligned}$$

```
let r = ref 0
let count_and_sum l =
  fold_left (fun a x -> incr r; a+x) 0 l
```

Exercise: give the instantiation of the invariant J in the code above.

Application of fold-left

$$\begin{aligned} \forall f a l J. \quad & (\forall x i k. \{J i k\} (f i x) \{\lambda j. J j (k \& x)\}) \\ \Rightarrow \quad & \{J a \text{ nil}\} (\text{fold_left } f a l) \{\lambda b. J b l\} \end{aligned}$$

```
let r = ref 0
let count_and_sum l =
  fold_left (fun a x -> incr r; a+x) 0 l
```

Exercise: give the instantiation of the invariant J in the code above.

$$J i k \equiv (r \mapsto |k|) * \text{'}i = \text{Sum } k\text{'}$$

where $\text{Sum } k \equiv \text{Fold } (+) 0 k$.

Fold-right

```
let rec fold_right f l a =  
  match l with  
  | [] -> a  
  | x::k -> f x (fold_right f k a)
```

Example:

$$\text{fold_right } f [6 :: 4 :: 7] a \equiv f\ 6 (f\ 4 (f\ 7 a))$$

Exercise: give a specification for fold_right.

Fold-right

```
let rec fold_right f l a =  
  match l with  
  | [] -> a  
  | x::k -> f x (fold_right f k a)
```

Example:

$$\text{fold_right } f [6 :: 4 :: 7] a \equiv f\ 6 (f\ 4 (f\ 7 a))$$

Exercise: give a specification for `fold_right`.

$$\begin{aligned} \forall f l a J. \quad & (\forall x i k. \{J\ i\ k\} (f\ x\ i) \{\lambda j. J\ j\ (x :: k)\}) \\ \Rightarrow \quad & \{J\ a\ \text{nil}\} (\text{fold_right } f\ l\ a) \{\lambda b. J\ b\ l\} \end{aligned}$$

Map: simple specification for pure functions

```
let rec map f l =  
  match l with  
  | [] -> []  
  | x::k -> (f x)::(map f k)
```

Simple specification, for the case where f is pure:

$$\begin{aligned} \forall f l P. \quad & (\forall x. \{^{\top}\} (f x) \{\lambda x'. {}^{\top}P x x'\}) \\ \Rightarrow & \{^{\top}\} (\text{map } f l) \{\lambda l'. {}^{\top}\text{Forall2 } P l l'\} \end{aligned}$$

where:

$$\frac{}{\text{Forall2 } P \text{ nil nil}} \qquad \frac{P x x' \quad \text{Forall2 } P l l'}{\text{Forall2 } P (x :: l) (x' :: l')}$$

Map: general specification

Specification of map:

$$\begin{aligned} \forall f l P. \quad & (\forall x. \{ \ulcorner \urcorner \} (f x) \{ \lambda x'. \ulcorner P x x' \urcorner \}) \\ \Rightarrow \quad & \{ \ulcorner \urcorner \} (\text{map } f l) \{ \lambda l'. \ulcorner \text{Forall2 } P l l' \urcorner \} \end{aligned}$$

Specification of iter:

$$\begin{aligned} \forall f l I. \quad & (\forall x k. \{ I k \} (f x) \{ \lambda _. I (k \& x) \}) \\ \Rightarrow \quad & \{ I \text{ nil} \} (\text{iter } f l) \{ \lambda _. I l \} \end{aligned}$$

Map: general specification

Specification of map:

$$\begin{aligned}\forall flP. \quad & (\forall x. \{ \ulcorner \urcorner \} (f x) \{ \lambda x'. \ulcorner P x x' \urcorner \}) \\ \Rightarrow & \{ \ulcorner \urcorner \} (\text{map } f l) \{ \lambda l'. \ulcorner \text{Forall2 } P l l' \urcorner \}\end{aligned}$$

Specification of iter:

$$\begin{aligned}\forall flI. \quad & (\forall xk. \{ I k \} (f x) \{ \lambda _. I (k \& x) \}) \\ \Rightarrow & \{ I \text{ nil} \} (\text{iter } f l) \{ \lambda _. I l \}\end{aligned}$$

Combining the two:

$$\begin{aligned}\forall flPI. \quad & (\forall xk. \{ I k \} (f x) \{ \lambda x'. \ulcorner P x x' \urcorner * I (k \& x) \}) \\ \Rightarrow & \{ I \text{ nil} \} (\text{map } f l) \{ \lambda l'. \ulcorner \text{Forall2 } P l l' \urcorner * I l \}\end{aligned}$$

Map: general specification, alternative

$$\begin{aligned} \forall f l P I. \quad & (\forall x k. \{I\ k\} (f\ x) \{\lambda x'. \text{'}P\ x\ x'\text{'}\} * J\ (k \& x)\}) \\ \Rightarrow \quad & \{I\ \text{nil}\} (\text{map } f\ l) \{\lambda l'. \text{'}Forall2\ P\ l\ l'\text{'}\} * I\ l\} \end{aligned}$$

Alternative specification:

$$\begin{aligned} \forall f l J'. \quad & (\forall x k k'. \{J'\ k\ k'\} (f\ x) \{\lambda x'. J'\ (k \& x)\ (k' \& x')\}) \\ \Rightarrow \quad & \{J'\ \text{nil}\ \text{nil}\} (\text{map } f\ l) \{\lambda l'. J'\ l\ l'\} \end{aligned}$$

Previous specification derivable from the above one:

$$J'\ k\ k' \equiv \text{'}Forall2\ P\ k\ k'\text{'}\ * I\ k$$

Sorting with comparison function

Example:

```
List.sort (fun x y -> x - y) [2;4;5;3;2;9]
```

Specification:

$$\begin{aligned} & \forall f l. \forall (\leq). \\ & \quad \text{total-order } (\leq) \\ & \quad \wedge \quad (\forall xy. \{ \ulcorner \} (f \ x \ y) \ \{ \lambda n. \ulcorner n \leq 0 \Leftrightarrow x \leq y \urcorner \}) \\ & \Rightarrow \quad \{ \ulcorner \} (\text{sort } f \ l) \ \{ \lambda l'. \ulcorner \text{permut } l \ l' \wedge \text{sorted } (\leq) \ l' \urcorner \} \end{aligned}$$

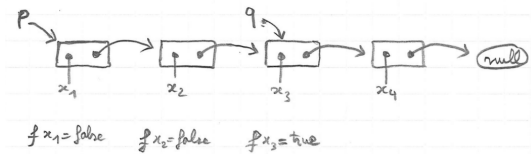
Find with a boolean predicate, on pure lists

```
let rec find f l =  
  match l with  
  | [] -> None  
  | x::k -> if f x  
              then Some x  
              else find f k
```

Specification:

$$\begin{aligned} \forall f l P. \quad & (\forall x. \{ \ulcorner \urcorner \} (f x) \{ \lambda b. \ulcorner b = \text{true} \Leftrightarrow P x \urcorner \}) \\ \Rightarrow \quad & \{ \ulcorner \urcorner \} (\text{find } f l) \{ \lambda o. \ulcorner \text{match } o \text{ with} \\ & \quad | \text{None} \Rightarrow \text{Forall } (\neg P) l \\ & \quad | \text{Some } x \Rightarrow \exists k t. l = k ++ x :: t \\ & \quad \wedge \text{Forall } (\neg P) k \wedge P x \urcorner \} \end{aligned}$$

Find with a boolean predicate, on mutable lists

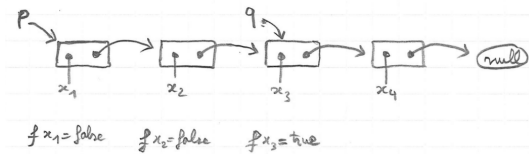


Specification:

$$\forall f p l P. \quad (\forall x. \{ ' \} \} (f x) \{ \lambda b. 'b = \text{true} \Leftrightarrow P x' \})$$

$$\begin{aligned} \Rightarrow \quad & \{ p \rightsquigarrow \text{MList } l \} \\ & (\text{mfind } f p) \\ & \{ \lambda o. \end{aligned}$$

Find with a boolean predicate, on mutable lists



Specification:

$$\forall f p l P. \ (\forall x. \ \{\text{' '}\} \ (f\ x) \ \{\lambda b. \ \text{'b = true'} \Leftrightarrow P\ x\}\})$$

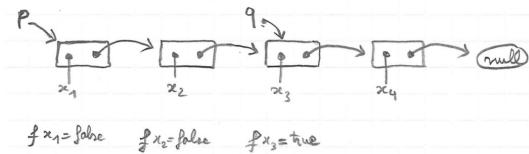
$$\Rightarrow \ \{p \rightsquigarrow \text{MList } l\}$$

$$(\text{mfind } f\ p)$$

$$\{\lambda o. \text{match } o \text{ with}$$

$$| \text{None} \Rightarrow p \rightsquigarrow \text{MList } l * \text{'Forall } (\neg P) \ l'$$

Find with a boolean predicate, on mutable lists



Specification:

$$\begin{aligned}
 & \forall fplP. \quad (\forall x. \{ ' \} \} (f x) \{ \lambda b. 'b = \text{true} \Leftrightarrow P x' \}) \\
 & \Rightarrow \quad \{ p \rightsquigarrow \text{MList } l \} \\
 & \quad (\text{mfind } f p) \\
 & \quad \{ \lambda o. \text{match } o \text{ with} \\
 & \quad \quad | \text{None} \Rightarrow p \rightsquigarrow \text{MList } l * ' \text{Forall } (\neg P) l' \\
 & \quad \quad | \text{Some } q \Rightarrow \exists kt. p \rightsquigarrow \text{MlistSeg } q k * q \rightsquigarrow \text{MList } (x :: t) \\
 & \quad \quad \quad * 'l = k ++ x :: t \wedge \text{Forall } (\neg P) k \wedge P x' \\
 & \quad \quad \}
 \end{aligned}$$

Summary

$$\begin{aligned} & (\forall x k. \{I\ k\} (f\ x) \{\lambda_. I\ (k\&x)\}) \\ \Rightarrow & \{I\ \text{nil}\} (\text{iter}\ f\ l) \{\lambda_. I\ l\} \end{aligned}$$

$$\begin{aligned} & (\forall x i k. \{J\ i\ k\} (f\ i\ x) \{\lambda j. J\ j\ (k\&x)\}) \\ \Rightarrow & \{J\ a\ \text{nil}\} (\text{fold}\ f\ a\ l) \{\lambda b. J\ b\ l\} \end{aligned}$$

$$\begin{aligned} & (\forall x k k'. \{J\ k\ k'\} (f\ x) \{\lambda x'. J\ (k\&x) (k'\&x')\}) \\ \Rightarrow & \{J\ \text{nil}\ \text{nil}\} (\text{map}\ f\ l) \{\lambda l'. J\ l\ l'\} \end{aligned}$$

- Add the hypothesis $l = k ++ x :: s$ if the position of x matters.
- Boolean predicates: $\forall x. \{\ulcorner \urcorner\} (f\ x) \{\lambda b. \ulcorner b = \text{true} \Leftrightarrow P\ x \urcorner\}$.
- Order functions: $\forall x y. \{\ulcorner \urcorner\} (f\ x\ y) \{\lambda n. \ulcorner n \leq 0 \Leftrightarrow x \leq y \urcorner\}$.

Chapter 17

Higher-order representation predicates

Overview

- 1 Higher-order predicate:

$p \rightsquigarrow \text{MList } L$ is generalized into $p \rightsquigarrow \text{Mlistof } R L$

- 2 Identity representation predicate:

$p \rightsquigarrow \text{Mlistof Id } L$ is the same as $p \rightsquigarrow \text{MList } L$

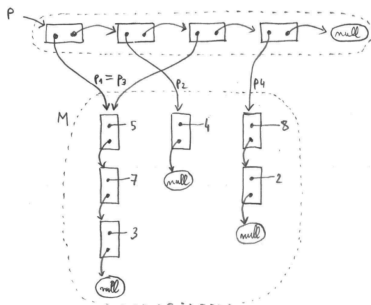
- 3 Control accesses:

$\{p \rightsquigarrow \text{MCellof Id } v_1 R_2 V_2\} (p.\text{hd}) \{\lambda x. \text{' } x = v_1 \text{' } * \dots\}$

- 4 Compose recursively:

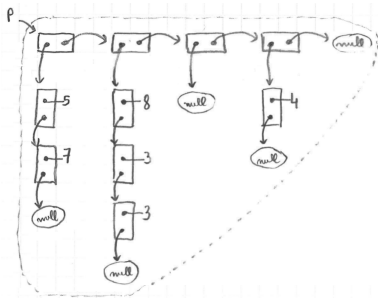
$p \rightsquigarrow \text{Nodeof } R X (\text{Mlistof } (\text{Narytreeof } R)) L$

Mutable list of possibly-aliased lists



$$p \rightsquigarrow \text{MList } K * \left(\bigotimes_{(p_i, L_i) \in M} p_i \rightsquigarrow \text{MList } L_i \right) * \lceil \forall p_i \in K. p_i \in \text{dom } M \rceil$$

Mutable list of disjoint mutable lists

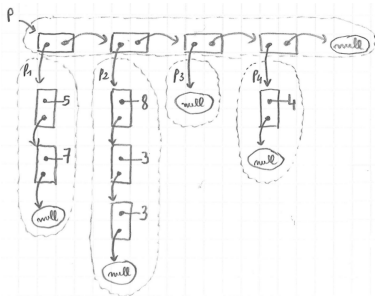


$$L = (5 :: 7 :: \text{nil}) :: (8 :: 3 :: 3 :: \text{nil}) \\ :: (\text{nil}) :: (4 :: \text{nil}) :: \text{nil}$$

$$p \rightsquigarrow \text{MlistofMlist } L$$

(to be later generalized into: $p \rightsquigarrow \text{Mlistof } R L$)

Representation using iterated star



$$L = (5 :: 7 :: \text{nil}) :: (8 :: 3 :: \text{nil}) :: (\text{nil}) :: (4 :: \text{nil}) :: \text{nil}$$

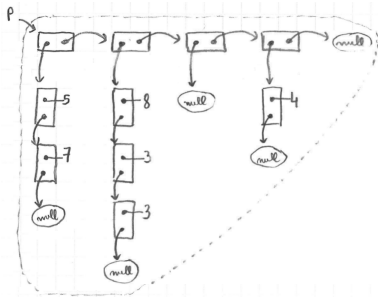
$$K = p_1 :: p_2 :: p_3 :: p_4 :: \text{nil}$$

$$p \rightsquigarrow \text{MlistofMlist } L \quad \equiv \quad \exists K. \quad p \rightsquigarrow \text{Mlist } K$$

$$* \quad \bigotimes_{i \in [0, |L|)} (K[i]) \rightsquigarrow \text{Mlist } (L[i])$$

$$* \quad \ulcorner |K| = |L| \urcorner$$

Representation using a recursive predicate



$$L = (5 :: 7 :: \text{nil}) :: (8 :: 3 :: 3 :: \text{nil}) \\ :: (\text{nil}) :: (4 :: \text{nil}) :: \text{nil}$$

$p \rightsquigarrow \text{MlistofMlist } L \equiv \text{match } L \text{ with}$

| $\text{nil} \Rightarrow \text{'p = null'}$

| $X :: L' \Rightarrow \exists x p'. p \rightsquigarrow \{\text{hd}=x; \text{tl}=p'\}$

* $p' \rightsquigarrow \text{MlistofMlist } L'$

* $x \rightsquigarrow \text{MList } X$

Generalization to a higher-order predicate

$$\begin{aligned} p \rightsquigarrow \text{MlistofMlist } L &\equiv \text{match } L \text{ with} \\ &\quad | \text{nil} \Rightarrow \text{'p = null'} \\ &\quad | X :: L' \Rightarrow \exists x p'. \quad p \rightsquigarrow \{\text{hd}=x; \text{tl}=p'\} \\ &\quad \quad * \quad p' \rightsquigarrow \text{MlistofMlist } L' \\ &\quad \quad * \quad x \rightsquigarrow \text{Mlist } X \end{aligned}$$

Generalization:

$$\begin{aligned} p \rightsquigarrow \text{Mlistof } R L &\equiv \text{match } L \text{ with} \\ &\quad | \text{nil} \Rightarrow \text{'p = null'} \\ &\quad | X :: L' \Rightarrow \exists x p'. \quad p \rightsquigarrow \{\text{hd}=x; \text{tl}=p'\} \\ &\quad \quad * \quad p' \rightsquigarrow \text{Mlistof } R L' \\ &\quad \quad * \quad x \rightsquigarrow R X \end{aligned}$$

In particular:

$$p \rightsquigarrow \text{MlistofMlist } L = p \rightsquigarrow \text{Mlistof Mlist } L$$

Type-checking

$p \rightsquigarrow \text{Mlistof } R L$ is a notation for $\text{Mlistof } R L p$ (of type Hprop)
 $x \rightsquigarrow R X$ is a notation for $R X x$ (of type Hprop)

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

Exercise: since $(p : \text{loc})$ and $(x : \text{Val})$ and $(X : A)$ for some A ,
what is the type of R ? What is the type of Mlistof ?

Type-checking

$p \rightsquigarrow \text{Mlistof } R L$ is a notation for $\text{Mlistof } R L p$ (of type Hprop)
 $x \rightsquigarrow R X$ is a notation for $R X x$ (of type Hprop)

$$\begin{aligned} p \rightsquigarrow \text{Mlistof } R L &\equiv \text{match } L \text{ with} \\ &\quad | \text{nil} \Rightarrow \text{'p = null'} \\ &\quad | X :: L' \Rightarrow \exists x p'. \quad \begin{aligned} &p \rightsquigarrow \{\text{hd}=x; \text{tl}=p'\} \\ &* \quad p' \rightsquigarrow \text{Mlistof } R L' \\ &* \quad x \rightsquigarrow R X \end{aligned} \end{aligned}$$

Exercise: since $(p : \text{loc})$ and $(x : \text{Val})$ and $(X : A)$ for some A , what is the type of R ? What is the type of Mlistof ?

- $R : A \rightarrow \text{Val} \rightarrow \text{Hprop}$
- $\text{Mlistof} : \forall A. (A \rightarrow \text{Val} \rightarrow \text{Hprop}) \rightarrow \text{list } A \rightarrow \text{loc} \rightarrow \text{Hprop}$

The identity representation predicate

$$\begin{aligned} p \rightsquigarrow \text{Mlistof } R L &\equiv \text{match } L \text{ with} \\ &\quad | \text{nil} \Rightarrow \text{'p = null'} \\ &\quad | X :: L' \Rightarrow \exists x p'. \quad p \rightsquigarrow \{\text{hd}=x; \text{tl}=p'\} \\ &\qquad \qquad \qquad * \quad p' \rightsquigarrow \text{Mlistof } R L' \\ &\qquad \qquad \qquad * \quad x \rightsquigarrow R X \end{aligned}$$

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

Exercise: define the identity representation predicate Id such that

$$p \rightsquigarrow \text{Mlistof Id } L = p \rightsquigarrow \text{MList } L$$

The identity representation predicate

$$\begin{aligned} p \rightsquigarrow \text{Mlistof } R L &\equiv \text{match } L \text{ with} \\ &\quad | \text{nil} \Rightarrow \text{'p = null'} \\ &\quad | X :: L' \Rightarrow \exists x p'. \quad p \rightsquigarrow \{\text{hd}=x; \text{tl}=p'\} \\ &\qquad \qquad \qquad * \quad p' \rightsquigarrow \text{Mlistof } R L' \\ &\qquad \qquad \qquad * \quad x \rightsquigarrow R X \end{aligned}$$

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

Exercise: define the identity representation predicate Id such that

$$p \rightsquigarrow \text{Mlistof Id } L = p \rightsquigarrow \text{MList } L$$

Definition:

$$x \rightsquigarrow \text{Id } X \equiv \text{'x = X'}$$

Summary

- ① Higher-order predicate:

$p \rightsquigarrow \text{MList } L$ is generalized into $p \rightsquigarrow \text{Mlistof } R L$

- ② Identity representation predicate:

$p \rightsquigarrow \text{MlistofId } L$ is the same as $p \rightsquigarrow \text{MList } L$

Chapter 18

Separating implication

Separating implication or “magic wand”

Recalling separating conjunction:

$$(P_1 * P_2)h \equiv \exists h_1, h_2. h = h_1 \uplus h_2 \wedge P_1 h_1 \wedge P_2 h_2$$

Introducing *separating implication*:

$$(P \multimap Q)h \equiv \forall h_1. h \perp h_1 \wedge P h_1 \Rightarrow Q(h \uplus h_1)$$

Intuition:

$$(P \multimap Q) * P \triangleright Q$$

Rules:

$$\frac{R * P \vdash Q}{R \vdash (P \multimap Q)} \qquad \frac{R_1 \vdash (P \multimap Q) \quad R_2 \vdash P}{R_1 * R_2 \vdash Q}$$

Separating implication examples

Exercise: Give heaps satisfying the following predicates:

- 1 $\text{''} \multimap (1 \mapsto 2)$
- 2 $\text{'False'} \multimap (1 \mapsto 2)$
- 3 $\text{'}x \geq 1\text{'} \multimap \text{'}x \geq 0\text{'}$
- 4 $(1 \mapsto 4) \multimap (1 \mapsto 4) * (2 \mapsto 3)$
- 5 $(1 \mapsto 2) \multimap (1 \mapsto 2)$
- 6 $(1 \mapsto 2) \multimap \text{'False'}$
- 7 $(1 \mapsto 2) \multimap \text{''}$
- 8 $(1 \mapsto 2) \multimap (1 \mapsto 3)$

Separating implication examples

Exercise: Give heaps satisfying the following predicates:

1 $\text{''} \multimap (1 \mapsto 2)$ $\{(1, 2)\}$

2 $\text{'False'} \multimap (1 \mapsto 2)$

3 $\text{'}x \geq 1\text{' } \multimap \text{'}x \geq 0\text{'}$

4 $(1 \mapsto 4) \multimap (1 \mapsto 4) * (2 \mapsto 3)$

5 $(1 \mapsto 2) \multimap (1 \mapsto 2)$

6 $(1 \mapsto 2) \multimap \text{'False'}$

7 $(1 \mapsto 2) \multimap \text{''}$

8 $(1 \mapsto 2) \multimap (1 \mapsto 3)$

Separating implication examples

Exercise: Give heaps satisfying the following predicates:

- 1 $\text{''} \multimap (1 \mapsto 2)$ $\{(1, 2)\}$
- 2 $\text{'False'} \multimap (1 \mapsto 2)$ all heaps
- 3 $\text{'}x \geq 1\text{' } \multimap \text{'}x \geq 0\text{'}$
- 4 $(1 \mapsto 4) \multimap (1 \mapsto 4) * (2 \mapsto 3)$
- 5 $(1 \mapsto 2) \multimap (1 \mapsto 2)$
- 6 $(1 \mapsto 2) \multimap \text{'False'}$
- 7 $(1 \mapsto 2) \multimap \text{''}$
- 8 $(1 \mapsto 2) \multimap (1 \mapsto 3)$

Separating implication examples

Exercise: Give heaps satisfying the following predicates:

- | | | |
|---|---|------------------|
| 1 | $\text{True} \multimap (1 \mapsto 2)$ | $\{(1, 2)\}$ |
| 2 | $\text{False} \multimap (1 \mapsto 2)$ | all heaps |
| 3 | $\text{True} \multimap (x \geq 1)$ | only $\emptyset$ |
| 4 | $(1 \mapsto 4) \multimap (1 \mapsto 4) * (2 \mapsto 3)$ | |
| 5 | $(1 \mapsto 2) \multimap (1 \mapsto 2)$ | |
| 6 | $(1 \mapsto 2) \multimap \text{False}$ | |
| 7 | $(1 \mapsto 2) \multimap \text{True}$ | |
| 8 | $(1 \mapsto 2) \multimap (1 \mapsto 3)$ | |

Separating implication examples

Exercise: Give heaps satisfying the following predicates:

- | | | |
|---|---|---|
| 1 | $\text{True} \multimap (1 \mapsto 2)$ | $\{(1, 2)\}$ |
| 2 | $\text{False} \multimap (1 \mapsto 2)$ | all heaps |
| 3 | $x \geq 1 \multimap x \geq 0$ | only $\emptyset$ |
| 4 | $(1 \mapsto 4) \multimap (1 \mapsto 4) * (2 \mapsto 3)$ | $\{(2, 3)\}$ and any h with $1 \in \text{dom}(h)$ |
| 5 | $(1 \mapsto 2) \multimap (1 \mapsto 2)$ | |
| 6 | $(1 \mapsto 2) \multimap \text{False}$ | |
| 7 | $(1 \mapsto 2) \multimap \text{True}$ | |
| 8 | $(1 \mapsto 2) \multimap (1 \mapsto 3)$ | |

Separating implication examples

Exercise: Give heaps satisfying the following predicates:

- | | | |
|---|---|---|
| 1 | $\text{True} \multimap (1 \mapsto 2)$ | $\{(1, 2)\}$ |
| 2 | $\text{False} \multimap (1 \mapsto 2)$ | all heaps |
| 3 | $x \geq 1 \multimap x \geq 0$ | only $\emptyset$ |
| 4 | $(1 \mapsto 4) \multimap (1 \mapsto 4) * (2 \mapsto 3)$ | $\{(2, 3)\}$ and any h with $1 \in \text{dom}(h)$ |
| 5 | $(1 \mapsto 2) \multimap (1 \mapsto 2)$ | $\emptyset$ and any h with $1 \in \text{dom}(h)$ |
| 6 | $(1 \mapsto 2) \multimap \text{False}$ | |
| 7 | $(1 \mapsto 2) \multimap \text{True}$ | |
| 8 | $(1 \mapsto 2) \multimap (1 \mapsto 3)$ | |

Separating implication examples

Exercise: Give heaps satisfying the following predicates:

- | | | |
|---|---|---|
| 1 | $\text{True} \multimap (1 \mapsto 2)$ | $\{(1, 2)\}$ |
| 2 | $\text{False} \multimap (1 \mapsto 2)$ | all heaps |
| 3 | $x \geq 1 \multimap x \geq 0$ | only $\emptyset$ |
| 4 | $(1 \mapsto 4) \multimap (1 \mapsto 4) * (2 \mapsto 3)$ | $\{(2, 3)\}$ and any h with $1 \in \text{dom}(h)$ |
| 5 | $(1 \mapsto 2) \multimap (1 \mapsto 2)$ | $\emptyset$ and any h with $1 \in \text{dom}(h)$ |
| 6 | $(1 \mapsto 2) \multimap \text{False}$ | any h with $1 \in \text{dom}(h)$ |
| 7 | $(1 \mapsto 2) \multimap \text{True}$ | |
| 8 | $(1 \mapsto 2) \multimap (1 \mapsto 3)$ | |

Separating implication examples

Exercise: Give heaps satisfying the following predicates:

- | | | |
|---|---|---|
| 1 | $\text{True} \multimap (1 \mapsto 2)$ | $\{(1, 2)\}$ |
| 2 | $\text{False} \multimap (1 \mapsto 2)$ | all heaps |
| 3 | $x \geq 1 \multimap x \geq 0$ | only $\emptyset$ |
| 4 | $(1 \mapsto 4) \multimap (1 \mapsto 4) * (2 \mapsto 3)$ | $\{(2, 3)\}$ and any h with $1 \in \text{dom}(h)$ |
| 5 | $(1 \mapsto 2) \multimap (1 \mapsto 2)$ | $\emptyset$ and any h with $1 \in \text{dom}(h)$ |
| 6 | $(1 \mapsto 2) \multimap \text{False}$ | any h with $1 \in \text{dom}(h)$ |
| 7 | $(1 \mapsto 2) \multimap \text{True}$ | any h with $1 \in \text{dom}(h)$ |
| 8 | $(1 \mapsto 2) \multimap (1 \mapsto 3)$ | |

Separating implication examples

Exercise: Give heaps satisfying the following predicates:

- | | | |
|---|---|---|
| 1 | $\text{True} \multimap (1 \mapsto 2)$ | $\{(1, 2)\}$ |
| 2 | $\text{False} \multimap (1 \mapsto 2)$ | all heaps |
| 3 | $x \geq 1 \multimap x \geq 0$ | only $\emptyset$ |
| 4 | $(1 \mapsto 4) \multimap (1 \mapsto 4) * (2 \mapsto 3)$ | $\{(2, 3)\}$ and any h with $1 \in \text{dom}(h)$ |
| 5 | $(1 \mapsto 2) \multimap (1 \mapsto 2)$ | $\emptyset$ and any h with $1 \in \text{dom}(h)$ |
| 6 | $(1 \mapsto 2) \multimap \text{False}$ | any h with $1 \in \text{dom}(h)$ |
| 7 | $(1 \mapsto 2) \multimap \text{True}$ | any h with $1 \in \text{dom}(h)$ |
| 8 | $(1 \mapsto 2) \multimap (1 \mapsto 3)$ | any h with $1 \in \text{dom}(h)$ |

Separating implication examples

Exercise: Among the following heap entailments, which hold?

- 1 $P \triangleright (Q \multimap P * Q)$
- 2 $(Q \multimap P * Q) \triangleright P$
- 3 $(1 \mapsto 2) \multimap (1 \mapsto 3) \triangleright \text{'False'}$
- 4 $(1 \mapsto 2) \multimap (1 \mapsto 2 * 2 \mapsto 8) \triangleright 2 \mapsto 8$
- 5 $\text{'\texttt{\char"0000'}} \multimap P \triangleright P$
- 6 $P \triangleright \text{'\texttt{\char"0000'}} \multimap P$
- 7 $\text{'\texttt{\char"0000'}} \triangleright (P \multimap Q \multimap P * Q)$
- 8 $\text{'}P \triangleright Q\text{'}$ $\triangleright (P \multimap Q)$
- 9 $(P \multimap Q) \triangleright \text{'}P \triangleright Q\text{'}$

Separating implication examples

Exercise: Among the following heap entailments, which hold?

- ① $P \triangleright (Q \multimap P * Q)$ **yes: unfold and behold the definition of $*$**
- ② $(Q \multimap P * Q) \triangleright P$
- ③ $(1 \mapsto 2) \multimap (1 \mapsto 3) \triangleright \text{'False'}$
- ④ $(1 \mapsto 2) \multimap (1 \mapsto 2 * 2 \mapsto 8) \triangleright 2 \mapsto 8$
- ⑤ $\text{'\top'} \multimap P \triangleright P$
- ⑥ $P \triangleright \text{'\top'} \multimap P$
- ⑦ $\text{'\top'} \triangleright (P \multimap Q \multimap P * Q)$
- ⑧ $\text{'}P \triangleright Q\text{' } \triangleright (P \multimap Q)$
- ⑨ $(P \multimap Q) \triangleright \text{'}P \triangleright Q\text{'}$

Separating implication examples

Exercise: Among the following heap entailments, which hold?

- ① $P \triangleright (Q \multimap P * Q)$ **yes: unfold and behold the definition of $*$**
- ② $(Q \multimap P * Q) \triangleright P$ **no e.g. with $P = Q = \text{'False'}$**
- ③ $(1 \mapsto 2) \multimap (1 \mapsto 3) \triangleright \text{'False'}$
- ④ $(1 \mapsto 2) \multimap (1 \mapsto 2 * 2 \mapsto 8) \triangleright 2 \mapsto 8$
- ⑤ $\text{' $\top$ '} \multimap P \triangleright P$
- ⑥ $P \triangleright \text{' $\top$ '} \multimap P$
- ⑦ $\text{' $\top$ '} \triangleright (P \multimap Q \multimap P * Q)$
- ⑧ $\text{' $P \triangleright Q$ '} \triangleright (P \multimap Q)$
- ⑨ $(P \multimap Q) \triangleright \text{' $P \triangleright Q$ '}$

Separating implication examples

Exercise: Among the following heap entailments, which hold?

- ① $P \triangleright (Q \multimap P * Q)$ **yes: unfold and behold the definition of $*$**
- ② $(Q \multimap P * Q) \triangleright P$ **no e.g. with $P = Q = \text{'False'}$**
- ③ $(1 \mapsto 2) \multimap (1 \mapsto 3) \triangleright \text{'False'}$ **no e.g. $\{(1, 4)\}$ satisfies left**
- ④ $(1 \mapsto 2) \multimap (1 \mapsto 2 * 2 \mapsto 8) \triangleright 2 \mapsto 8$
- ⑤ $\text{' $\top$ '} \multimap P \triangleright P$
- ⑥ $P \triangleright \text{' $\top$ '} \multimap P$
- ⑦ $\text{' $\top$ '} \triangleright (P \multimap Q \multimap P * Q)$
- ⑧ $\text{' $P \triangleright Q$ '} \triangleright (P \multimap Q)$
- ⑨ $(P \multimap Q) \triangleright \text{' $P \triangleright Q$ '}$

Separating implication examples

Exercise: Among the following heap entailments, which hold?

- ① $P \triangleright (Q \multimap P * Q)$ **yes: unfold and behold the definition of $*$**
- ② $(Q \multimap P * Q) \triangleright P$ **no e.g. with $P = Q = \text{'False'}$**
- ③ $(1 \mapsto 2) \multimap (1 \mapsto 3) \triangleright \text{'False'}$ **no e.g. $\{(1, 4)\}$ satisfies left**
- ④ $(1 \mapsto 2) \multimap (1 \mapsto 2 * 2 \mapsto 8) \triangleright 2 \mapsto 8$ **no (same)**
- ⑤ $\text{' $\top$ '} \multimap P \triangleright P$
- ⑥ $P \triangleright \text{' $\top$ '} \multimap P$
- ⑦ $\text{' $\top$ '} \triangleright (P \multimap Q \multimap P * Q)$
- ⑧ $\text{' $P \triangleright Q$ '} \triangleright (P \multimap Q)$
- ⑨ $(P \multimap Q) \triangleright \text{' $P \triangleright Q$ '}$

Separating implication examples

Exercise: Among the following heap entailments, which hold?

- ① $P \triangleright (Q \multimap P * Q)$ **yes: unfold and behold the definition of $*$**
- ② $(Q \multimap P * Q) \triangleright P$ **no e.g. with $P = Q = \text{'False'}$**
- ③ $(1 \mapsto 2) \multimap (1 \mapsto 3) \triangleright \text{'False'}$ **no e.g. $\{(1, 4)\}$ satisfies left**
- ④ $(1 \mapsto 2) \multimap (1 \mapsto 2 * 2 \mapsto 8) \triangleright 2 \mapsto 8$ **no (same)**
- ⑤ $\text{'}\top\text{'} \multimap P \triangleright P$ **yes, and...**
- ⑥ $P \triangleright \text{'}\top\text{'} \multimap P$
- ⑦ $\text{'}\top\text{'} \triangleright (P \multimap Q \multimap P * Q)$
- ⑧ $\text{'}P \triangleright Q\text{'}$ $\triangleright (P \multimap Q)$
- ⑨ $(P \multimap Q) \triangleright \text{'}P \triangleright Q\text{'}$

Separating implication examples

Exercise: Among the following heap entailments, which hold?

- ① $P \triangleright (Q \multimap P * Q)$ **yes: unfold and behold the definition of $*$**
- ② $(Q \multimap P * Q) \triangleright P$ **no e.g. with $P = Q = \text{'False'}$**
- ③ $(1 \mapsto 2) \multimap (1 \mapsto 3) \triangleright \text{'False'}$ **no e.g. $\{(1, 4)\}$ satisfies left**
- ④ $(1 \mapsto 2) \multimap (1 \mapsto 2 * 2 \mapsto 8) \triangleright 2 \mapsto 8$ **no (same)**
- ⑤ $\text{'}\top\text{'} \multimap P \triangleright P$ **yes, and...**
- ⑥ $P \triangleright \text{'}\top\text{'} \multimap P$ **...yes: P and $\text{'}\top\text{'} \multimap P$ are equivalent**
- ⑦ $\text{'}\top\text{'} \triangleright (P \multimap Q \multimap P * Q)$
- ⑧ $\text{'}P \triangleright Q\text{'}$ $\triangleright (P \multimap Q)$
- ⑨ $(P \multimap Q) \triangleright \text{'}P \triangleright Q\text{'}$

Separating implication examples

Exercise: Among the following heap entailments, which hold?

- ① $P \triangleright (Q \multimap P * Q)$ **yes:** unfold and behold the definition of $*$
- ② $(Q \multimap P * Q) \triangleright P$ **no** e.g. with $P = Q = \text{'False'}$
- ③ $(1 \mapsto 2) \multimap (1 \mapsto 3) \triangleright \text{'False'}$ **no** e.g. $\{(1, 4)\}$ satisfies left
- ④ $(1 \mapsto 2) \multimap (1 \mapsto 2 * 2 \mapsto 8) \triangleright 2 \mapsto 8$ **no** (same)
- ⑤ $\text{'}\top\text{'} \multimap P \triangleright P$ **yes, and...**
- ⑥ $P \triangleright \text{'}\top\text{'} \multimap P$ **...yes:** P and $\text{'}\top\text{'} \multimap P$ are equivalent
- ⑦ $\text{'}\top\text{'} \triangleright (P \multimap Q \multimap P * Q)$ **yes:** unfold $\triangleright$ and obtain approx. (1)
- ⑧ $\text{'}P \triangleright Q\text{'}$ $\triangleright (P \multimap Q)$
- ⑨ $(P \multimap Q) \triangleright \text{'}P \triangleright Q\text{'}$

Separating implication examples

Exercise: Among the following heap entailments, which hold?

- ① $P \triangleright (Q \multimap P * Q)$ yes: unfold and behold the definition of $*$
- ② $(Q \multimap P * Q) \triangleright P$ no e.g. with $P = Q = \text{'False'}$
- ③ $(1 \mapsto 2) \multimap (1 \mapsto 3) \triangleright \text{'False'}$ no e.g. $\{(1, 4)\}$ satisfies left
- ④ $(1 \mapsto 2) \multimap (1 \mapsto 2 * 2 \mapsto 8) \triangleright 2 \mapsto 8$ no (same)
- ⑤ $\text{'}\top\text{'} \multimap P \triangleright P$ yes, and...
- ⑥ $P \triangleright \text{'}\top\text{'} \multimap P$...yes: P and $\text{'}\top\text{'} \multimap P$ are equivalent
- ⑦ $\text{'}\top\text{'} \triangleright (P \multimap Q \multimap P * Q)$ yes: unfold $\triangleright$ and obtain approx. (1)
- ⑧ $\text{'}P \triangleright Q\text{'}$ $\triangleright (P \multimap Q)$ yes
- ⑨ $(P \multimap Q) \triangleright \text{'}P \triangleright Q\text{'}$

Separating implication examples

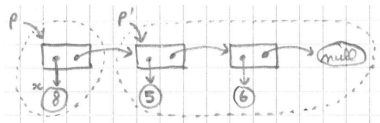
Exercise: Among the following heap entailments, which hold?

- ① $P \triangleright (Q \ast P \ast Q)$ yes: unfold and behold the definition of $\ast$
- ② $(Q \ast P \ast Q) \triangleright P$ no e.g. with $P = Q = \text{'False'}$
- ③ $(1 \mapsto 2) \ast (1 \mapsto 3) \triangleright \text{'False'}$ no e.g. $\{(1, 4)\}$ satisfies left
- ④ $(1 \mapsto 2) \ast (1 \mapsto 2 \ast 2 \mapsto 8) \triangleright 2 \mapsto 8$ no (same)
- ⑤ $\text{'\text{}} \ast P \triangleright P$ yes, and...
- ⑥ $P \triangleright \text{'\text{}} \ast P$...yes: P and $\text{'\text{}} \ast P$ are equivalent
- ⑦ $\text{'\text{}} \triangleright (P \ast Q \ast P \ast Q)$ yes: unfold $\triangleright$ and obtain approx. (1)
- ⑧ $\text{'}P \triangleright Q\text{'}$ $\triangleright (P \ast Q)$ yes
- ⑨ $(P \ast Q) \triangleright \text{'}P \triangleright Q\text{'}$ no, e.g. $P = \text{'\text{}}$ and $Q = 1 \mapsto 2$

Chapter 19

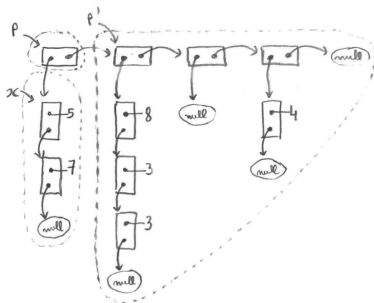
Higher-order representation predicates and the access problem

Specification of construction, for basic values



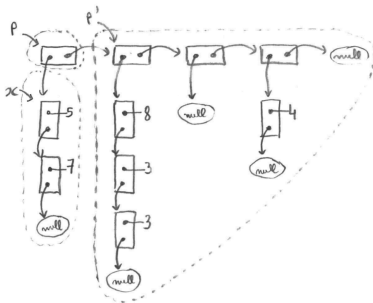
$$\{p' \rightsquigarrow \text{MList } L\} (\text{cons } x \ p') \ \{\lambda p. \ p \rightsquigarrow \text{MList } (x :: L)\}$$

Specification of construction



$$\{x \rightsquigarrow R X * p' \rightsquigarrow \text{Mlistof } R L\} (\text{cons } x p') \{\lambda p. p \rightsquigarrow \text{Mlistof } R (X :: L)\}$$

Specification of deconstruction

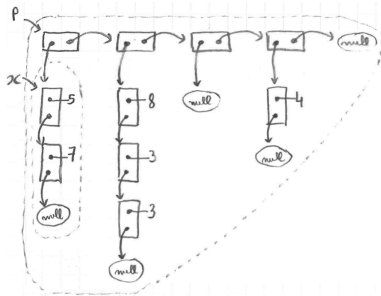


$$\{p \rightsquigarrow \text{Mlistof } R(X :: L)\}$$

$\text{uncons } p$

$$\{\lambda(x, p'). x \rightsquigarrow R X * p' \rightsquigarrow \text{Mlistof } R L\}$$

Specification of accesses: the problem



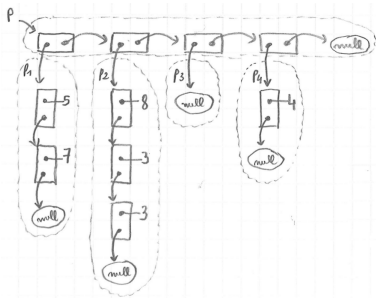
Incorrect specification for head:

$$\{p \rightsquigarrow \text{Mlistof } R(X :: L)\}$$

head p

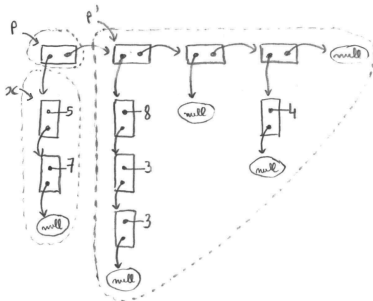
$$\{\lambda x. x \rightsquigarrow R X * p \rightsquigarrow \text{Mlistof } R(X :: L)\}$$

Specification of accesses: a brute force solution



$$\begin{aligned}
 p \rightsquigarrow \text{Mlistof } R L &= \exists K. \quad p \rightsquigarrow \text{MList } K \\
 &* \quad \bigotimes_{i \in \{0, \dots, |L| - 1\}} (K[i]) \rightsquigarrow R(L[i]) \\
 &* \quad \ulcorner |K| = |L| \urcorner
 \end{aligned}$$

Specification of accesses: focus before read

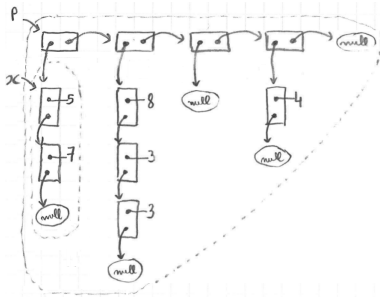


$$\begin{aligned}
 p \rightsquigarrow \text{Mlistof } R (X :: L) &= \exists x p'. \quad p \rightsquigarrow \{\text{hd}=x; \text{tl}=p'\} \\
 &\quad * \quad x \rightsquigarrow R X \\
 &\quad * \quad p' \rightsquigarrow \text{Mlistof } R L
 \end{aligned}$$

Then read using:

$$\{p \mapsto \{\text{hd}=x; \text{tl}=p'\}\} (p.\text{hd}) \{\lambda y. \text{'y = x'} * p \mapsto \{\text{hd}=x; \text{tl}=p'\}\}$$

Specification of accesses with separating implication



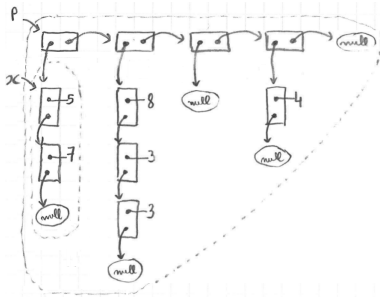
Correct specification with $\multimap$:

$$\{p \rightsquigarrow \text{Mlistof } R(X :: L)\}$$

$\text{head } p$

$$\{\lambda x. x \rightsquigarrow R X * (x \rightsquigarrow R X \multimap p \rightsquigarrow \text{Mlistof } R(X :: L))\}$$

Specification of accesses with separating implication



Correct specification with $\multimap$:

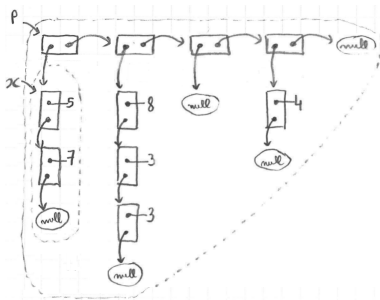
$$\{p \rightsquigarrow \text{Mlistof } R(X :: L)\}$$

$\text{head } p$

$$\{\lambda x. x \rightsquigarrow R X * (x \rightsquigarrow R X \multimap p \rightsquigarrow \text{Mlistof } R(X :: L))\}$$

Exercise: What problem is there, e.g. if $x \rightsquigarrow R X$ is $x \mapsto X$ (i.e. $R = \text{Ref}$)? How to generalize the specification to solve this problem?

Specification of accesses with separating implication



Correct specification with $\multimap$, generalized:

$$\{p \rightsquigarrow \text{Mlistof } R(X :: L)\}$$

$\text{head } p$

$$\{\lambda x. x \rightsquigarrow R X * (\forall X', x \rightsquigarrow R X' \multimap p \rightsquigarrow \text{Mlistof } R(X' :: L))\}$$

Reminder keywords

for

while

swap

$f \rightsquigarrow \text{Count } n$

iter

miter

fold_left

map

$p \rightsquigarrow \text{Mlistof } R\ L$

$P \rightarrow^* Q$

head