

# Separation Logic

## 3/4

Jean-Marie Madiot

Inria Paris

February 7, 2018

# 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:

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

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:

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

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  $a \leq b$ :

$$\frac{\begin{array}{c} H \triangleright I a \\ \forall i \in [a, b]. \{I i\} t \{\lambda.. I (i + 1)\} \\ I (b + 1) \triangleright Q () \end{array}}{\{H\} (\text{for } i = a \text{ to } b \text{ do } t) \{Q\}}$$

## Reasoning rule for for-loops

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

$$\frac{\begin{array}{c} H \triangleright I a \\ \forall i \in [a, b]. \{I i\} t \{\lambda.. I (i + 1)\} \\ I (b + 1) \triangleright Q () \end{array}}{\{H\} (\text{for } i = a \text{ to } b \text{ do } t) \{Q\}}$$

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

$$\frac{\begin{array}{c} H \triangleright I a \\ \forall i \in [a, b]. \{I i\} t \{\lambda.. I (i + 1)\} \\ I (\max a (b + 1)) \triangleright Q () \end{array}}{\{H\} (\text{for } i = a \text{ to } b \text{ do } t) \{Q\}}$$

## 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  $t_1$ .

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

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  $t_1$ .

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

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:

- inherently supports total correctness;
- allows to apply frame during iterations.

## Reasoning rule for while loops

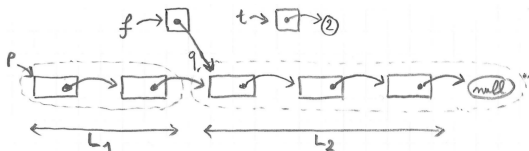
We focus on a different approach that:

- inherently supports total correctness;
- allows to apply frame during iterations.

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

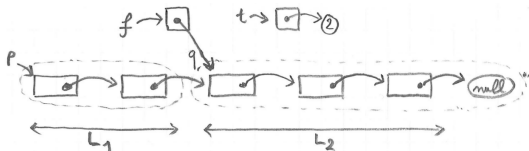
$$\frac{\{H\} (\text{if } t_1 \text{ then } (t_2; (\text{while } t_1 \text{ do } t_2)) \text{ else } ()) \{Q\}}{\{H\} (\text{while } t_1 \text{ do } t_2) \{Q\}}$$

# Length with a while loop



```
let rec 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 prove by induction on  $L_2$  that for any  $n$  and  $q$ :

$$\{q \rightsquigarrow \text{Mlist } L_2 \star f \mapsto q \star t \mapsto n\}$$

(while !f != null do incr t; f := (!f).tl; done)

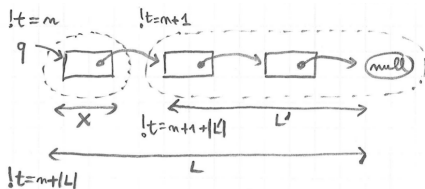
$$\{q \rightsquigarrow \text{Mlist } L_2 \star f \mapsto \text{null} \star t \mapsto (n + \text{length } L_2)\}$$

The loop unfolds to:

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

Exercise: describe the frame process in the induction for length.

# Length with a while loop: frame process




---

|  |  |                                  |                                  |             |
|--|--|----------------------------------|----------------------------------|-------------|
| $q \rightsquigarrow \text{Mlist } L_2$                             | $\star f \mapsto q$                            | $\star t \mapsto n$              | begin                            |             |
| $q \mapsto \{x; q'\} \star q' \rightsquigarrow \text{Mlist } L'_2$ | $\star f \mapsto q$                            | $\star t \mapsto n$              | unfold                           |             |
| $q \mapsto \{x; q'\} \star q' \rightsquigarrow \text{Mlist } L'_2$ | $\star f \mapsto q$                            | $\star t \mapsto n + 1$          | increment                        |             |
| $q \mapsto \{x; q'\} \star q' \rightsquigarrow \text{Mlist } L'_2$ | $\star f \mapsto q'$                           | $\star t \mapsto n + 1$          | shift head                       |             |
|  | $\star q' \rightsquigarrow \text{Mlist } L'_2$ | $\star f \mapsto q'$             | $\star t \mapsto n + 1$          | begin frame |
|  | $\star q' \rightsquigarrow \text{Mlist } L'_2$ | $\star f \mapsto \text{null}$    | $\star t \mapsto n + 1 +  L'_2 $ | induction   |
| $q \mapsto \{x; q'\} \star q' \rightsquigarrow \text{Mlist } L'_2$ | $\star f \mapsto \text{null}$                  | $\star t \mapsto n + 1 +  L'_2 $ | end frame                        |             |
| $q \rightsquigarrow \text{Mlist } L_2$                             | $\star f \mapsto \text{null}$                  | $\star 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
```

Find three useful specifications for `swap`:

- 1 a specification for non-aliased (distinct) arguments,
- 2 a specification for aliased (equal) arguments,
- 3 a most-general specification, stated using iterated conjunction (or another construct from Course 2).



# Functions with aliasing: 3 specifications for swap

Specification 1:

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

# Functions with aliasing: 3 specifications for swap

Specification 1:

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

Specification 2:

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

or simply:

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

# Functions with aliasing: 3 specifications for swap

Specification 1:

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

Specification 2:

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

or simply:

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

Specification 3:

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

## 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:

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

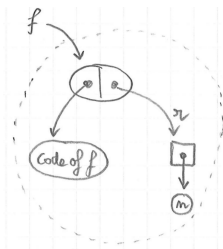
Successive states:

$$r \mapsto 3 \quad r \mapsto 4 \quad r \mapsto 5$$

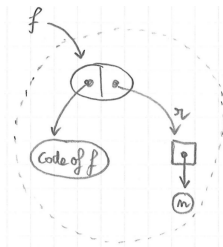
# Counter function: code

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

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

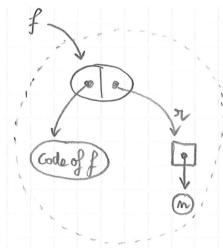


# Counter function: specification



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

# Counter function: specification

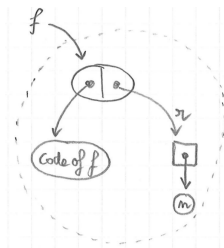


$$f \rightsquigarrow \text{Count } n \equiv \exists r. (r \mapsto n) \\ \star [\forall i. \{r \mapsto i\} (f ()) \{\lambda x. [x = i + 1] \star (r \mapsto i + 1)\}]$$

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



# Counter function: specification



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

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

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

$$\forall f i. \{f \rightsquigarrow \text{Count } i\} (f ()) \{\lambda x. [x = i + 1] \star f \rightsquigarrow \text{Count } (i + 1)\}$$

# 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:

$$\forall f x H Q. \quad \{H \star [ \{H\} (f x) \{Q\} ]\} (\text{apply } f x) \{Q\}$$

## 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  $H'$ .

$$\begin{aligned} \forall r f x P. \quad & \{[]\} (f x) \{\lambda y. [P x y]\} \\ \Rightarrow & \{r \mapsto x\} (\text{refapply } r f) \{\lambda_. \exists y. [P x y] \star r \mapsto y\} \end{aligned}$$

## 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  $Pxy$ .

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

$$\begin{aligned} \forall rfxP. \quad & \{[]\} (f\ x) \{\lambda y. [P\ x\ y]\} \\ \Rightarrow & \{r \mapsto x\} (\text{refapply } r\ f) \{\lambda_. \exists y. [P\ x\ y] \star r \mapsto y\} \end{aligned}$$

$$\begin{aligned} \forall rfxHH'P. \quad & \{H\} (f\ x) \{\lambda y. [P\ x\ y] \star H'\} \\ \Rightarrow & \{(r \mapsto x) \star H\} \\ & (\text{refapply } r\ f) \\ & \{\lambda_. \exists y. [P\ x\ y] \star (r \mapsto y) \star H'\} \end{aligned}$$

# Function twice

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

Specification:

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



# 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  $\text{int} \rightarrow \text{Hprop}$ .

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

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

```
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

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

## Length using iter

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

```
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.

## Length using iter

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

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

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

## Sum using iter

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

```
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

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

```
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`.

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

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:  $\sqrt{x_1} + \dots + \sqrt{x_n}$ .

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

## Constraints over the items

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

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

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

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

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

## 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:  $\sqrt{\sqrt{\sqrt{x_1 + x_2} + x_3}}$ .

```
iter (fun x -> r := sqrt (!r +. x)) [2.; -1.; 3.]
```

## 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:  $\sqrt{\sqrt{\sqrt{x_1 + x_2} + x_3}}$ .

```
iter (fun x -> r := sqrt (!r +. x)) [2.; -1.; 3.]
```

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

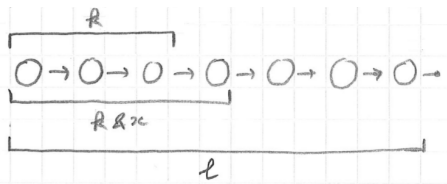
$$\begin{aligned} \forall f l. \quad & (\forall x k s. l = k ++ x :: s \Rightarrow \{I k\} (f x) \{\lambda_. I (k \& x)\}) \\ \Rightarrow & \{I \text{nil}\} (\text{iter } f l) \{\lambda_. I l\} \end{aligned}$$

# 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:

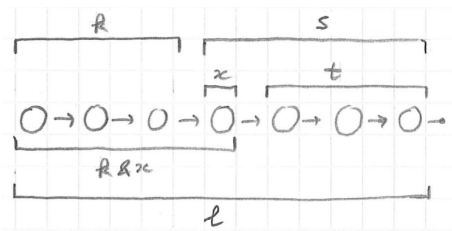
$$\forall xk. \{I k\} (f x) \{\lambda_. I (k&x)\}$$

Prove:

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

Proof by induction over a generalized statement:

$$\forall ks. \{I k\} (\text{iter } f s) \{\lambda_. I (k+s)\}$$





# 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:  $\forall ks. \{I k\} (\text{iter } f s) \{\lambda_. I (k ++ s)\}$

By induction on  $s$ :

- Case  $s = \text{nil}$ . 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:  $\forall ks. \{I k\} (\text{iter } f s) \{\lambda_. I (k\&s)\}$

By induction on  $s$ :

- Case  $s = \text{nil}$ . 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  $s = x :: t$ . Goal is:  $\{I k\} (\text{iter } f (x :: t)) \{\lambda_. I (k\&(x :: t))\}$ .

HYPOTHESIS-ON-F

INDUCTION-HYPOTHESIS

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

## Invariant on remaining items

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

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

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. Then, prove the new specification derivable from the old one.

## Invariant on remaining items

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

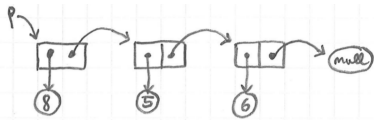
$$\begin{aligned} & (\forall \dots \{\dots\} (f x) \{\lambda_. \dots\}) \\ \Rightarrow & \{I' l\} (\text{iter } f l) \{\lambda_. I' \text{nil}\} \end{aligned}$$

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. Then, prove the new specification derivable from the old one.

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

Derivable using:  $I \equiv \lambda k. \exists s. [l = k ++ s] \star I' s.$

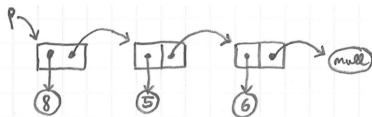
# Iterating over a mutable list



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

# Iterating over a mutable list

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



Specification:

$$\begin{aligned} \forall f p I l. & \quad (\forall x k. \{I k\} (f x) \{\lambda_. I (k \& x)\}) \\ \Rightarrow & \quad \{p \rightsquigarrow \mathbf{Mlist } l \star I \text{ nil}\} (\text{miter } f p) \{\lambda_. p \rightsquigarrow \mathbf{Mlist } l \star I l\} \end{aligned}$$

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

# Summary

Simplified:

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

Order-irrelevant:

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

Most-general:

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

Extension to mutable lists:

$$\begin{aligned} & (\forall xks. l = k ++ x :: s \Rightarrow \{I k\} (f x) \{\lambda_. I (k\&x)\}) \\ \Rightarrow & \{p \rightsquigarrow \mathbf{Mlist } l \star I \text{nil}\} (\text{miter } f p) \{\lambda_. p \rightsquigarrow \mathbf{Mlist } l \star I l\} \end{aligned}$$

# 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

$$\begin{aligned} \forall falJ. \quad & (\forall xik. \{Jik\} (f ix) \{\lambda j. Jj(k&x)\}) \\ \Rightarrow & \{J a nil\} (\text{fold\_left } fal) \{\lambda b. Jbl\} \end{aligned}$$

```
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

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

```
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.

$$J i k \equiv (r \mapsto |k|) \star [i = \text{Sum } k]$$

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:

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

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:

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

Exercise: give a specification for `fold_right`.

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

# 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:

$$\forall f l P. \quad (\forall x. \{[]\} (f x) \{\lambda x'. [P x x']\}) \\ \Rightarrow \{[]\} (\text{map } f l) \{\lambda l'. [\text{Forall2 } P l l']\}$$

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:

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

Specification of iter:

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



## Map: general specification

Specification of map:

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

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:

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

## Map: general specification, alternative

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

Alternative specification:

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

Above specification derivable from the previous one:

$$J' k k' \equiv [\text{Forall2 } P k k'] \star I k$$

# Sorting with comparison function

Example:

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

Specification:

$\forall fl. \forall (\leq).$

total-order ( $\leq$ )

$\wedge (\forall xy. \{[]\} (f\ x\ y) \{\lambda n. [n \leq 0 \Leftrightarrow x \leq y]\})$

$\Rightarrow \{[]\} (\text{sort } f\ l) \{\lambda l'. [ \text{permut } l\ l' \wedge \text{sorted } (\leq)\ l' ]\}$

# Sorting with comparison function

Example:

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

Specification:

$\forall fl. \forall (\leq).$

total-order ( $\leq$ )

$\wedge (\forall xy. \{[]\} (f\ x\ y) \{\lambda n. [n \leq 0 \Leftrightarrow x \leq y]\})$

$\Rightarrow \{[]\} (\text{sort } f\ l) \{\lambda l'. [\text{permut } l\ l' \wedge \text{sorted } (\leq)\ l']\}$

More general specification of comparison functions:

$\{[]\} (f\ x\ y) \{\lambda n. [\text{if } n = 0 \text{ then } x \approx y \text{ else if } n < 0 \text{ then } x < y \text{ else } x > y]\}$

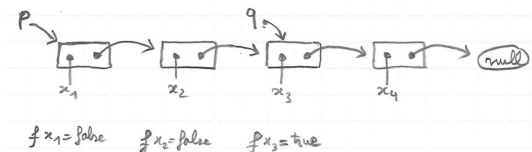
## Find with a boolean predicate, on pure lists

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

Specification:

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

# Find with a boolean predicate, on mutable lists



Specification:

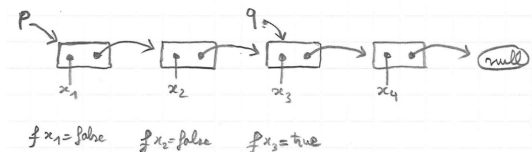
$$\forall f p l P. (\forall x. \{[]\} (f x) \{\lambda b. [b = \text{true} \Leftrightarrow P x]\})$$

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

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

$$\{\lambda o.$$

# Find with a boolean predicate, on mutable lists



Specification:

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

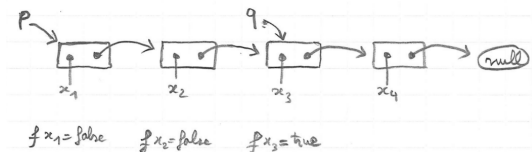
$$\Rightarrow \{p \rightsquigarrow \text{Mlist } l\}$$

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

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

$$| \text{None} \Rightarrow p \rightsquigarrow \text{Mlist } l \star [\text{Forall } (\neg P) l]$$

# Find with a boolean predicate, on mutable lists



Specification:

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

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

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

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

$$| \text{None} \Rightarrow p \rightsquigarrow \text{Mlist } l \star [\text{Forall } (\neg P) l]$$

$$| \text{Some } q \Rightarrow \exists kt. p \rightsquigarrow \text{MlistSeg } q k \star q \rightsquigarrow \text{Mlist } (x :: t) \star [l = k ++ x :: t \wedge \text{Forall } (\neg P) k \wedge P x]$$



# Summary

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

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

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

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

# Chapter 17

## Principle of Characteristic Formulas

# Idea of characteristic formulas

Goal: perform all the Separation Logic reasoning inside Coq.

Idea: build a logical formula  $\llbracket t \rrbracket$  satisfying the equivalence below.

$$\forall H Q. \llbracket t \rrbracket H Q \Leftrightarrow \{H\} t \{Q\}$$

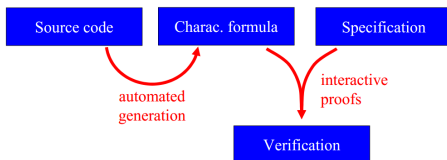
# Idea of characteristic formulas

Goal: perform all the Separation Logic reasoning inside Coq.

Idea: build a logical formula  $\llbracket t \rrbracket$  satisfying the equivalence below.

$$\forall H Q. \llbracket t \rrbracket H Q \Leftrightarrow \{H\} t \{Q\}$$

Schema:



# Properties of characteristic formulas

The characteristic formula  $\llbracket t \rrbracket$  of a term  $t$  is a predicate such that:

$$\forall H Q. \llbracket t \rrbracket H Q \Leftrightarrow \{H\} t \{Q\}$$

# Properties of characteristic formulas

The characteristic formula  $\llbracket t \rrbracket$  of a term  $t$  is a predicate such that:

$$\forall H Q. \llbracket t \rrbracket H Q \Leftrightarrow \{H\} t \{Q\}$$

Properties:

- $\llbracket t \rrbracket$  has type  $(\text{Heap} \rightarrow \text{Prop}) \rightarrow (\text{Val} \rightarrow \text{Heap} \rightarrow \text{Prop}) \rightarrow \text{Prop}$
- $\llbracket t \rrbracket$  characterizes the set of valid specifications for  $t$
- $\llbracket t \rrbracket$  is a higher-order logic formula built using  $\wedge, \vee, \Rightarrow, \exists, \dots$
- $\llbracket t \rrbracket$  is built automatically and compositionally
- $\llbracket t \rrbracket$  has size linear in the size of  $t$  and is easy to read

## Characteristic formula for sequence

$$\forall H Q. \llbracket t \rrbracket H Q \Leftrightarrow \{H\} t \{Q\}$$

$$\frac{\{H\} t_1 \{Q'\} \quad \{Q'()\} t_2 \{Q\}}{\{H\} (t_1 ; t_2) \{Q\}} \text{SEQ}$$

Goal:

$$\forall H Q. \llbracket t_1 ; t_2 \rrbracket H Q \Leftrightarrow \{H\} (t_1 ; t_2) \{Q\}$$

Exercise: define the characteristic formula for sequences.

# Characteristic formula for sequence

$$\forall H Q. \llbracket t \rrbracket H Q \Leftrightarrow \{H\} t \{Q\}$$

$$\frac{\{H\} t_1 \{Q'\} \quad \{Q' ()\} t_2 \{Q\}}{\{H\} (t_1 ; t_2) \{Q\}} \text{SEQ}$$

Goal:

$$\forall H Q. \llbracket t_1 ; t_2 \rrbracket H Q \Leftrightarrow \{H\} (t_1 ; t_2) \{Q\}$$

Exercise: define the characteristic formula for sequences.

$$\llbracket t_1 ; t_2 \rrbracket \equiv \lambda H. \lambda Q. \exists Q'. \llbracket t_1 \rrbracket H Q' \wedge \llbracket t_2 \rrbracket (Q' ()) Q$$



## Characteristic formula for let bindings

$$\frac{\{H\} t_1 \{Q'\} \quad \forall x. \{Q' x\} t_2 \{Q\}}{\{H\} (\text{let } x = t_1 \text{ in } t_2) \{Q\}}$$

Definition:

$$\llbracket \text{let } x = t_1 \text{ in } t_2 \rrbracket \equiv \lambda H Q. \exists Q'. \llbracket t_1 \rrbracket H Q' \wedge \forall x. \llbracket t_2 \rrbracket (Q' x) Q$$

# Characteristic formula for let bindings

$$\frac{\{H\} t_1 \{Q'\} \quad \forall x. \{Q' x\} t_2 \{Q\}}{\{H\} (\text{let } x = t_1 \text{ in } t_2) \{Q\}}$$

Definition:

$$\llbracket \text{let } x = t_1 \text{ in } t_2 \rrbracket \equiv \lambda H Q. \exists Q'. \llbracket t_1 \rrbracket H Q' \wedge \forall x. \llbracket t_2 \rrbracket (Q' x) Q$$

Technically,  $x$  has type var and:

$$\begin{aligned} \llbracket \text{let } x = t_1 \text{ in } t_2 \rrbracket \equiv \lambda H Q. \exists Q'. \llbracket t_1 \rrbracket H Q' \\ \wedge \forall (X : \text{Val}). \llbracket ([x \rightarrow X] t_2) \rrbracket (Q' X) Q \end{aligned}$$

# Characteristic formula for values

$$\frac{H \triangleright Q v}{\{H\} v \{Q\}} \text{VAL-FRAME}$$

Definition:

$$\llbracket v \rrbracket \equiv \lambda H Q. H \triangleright Q v$$

# Characteristic formula for conditionals

$$\frac{(b = \text{true} \Rightarrow \{H\} t_1 \{Q\}) \quad (b = \text{false} \Rightarrow \{H\} t_2 \{Q\})}{\{H\} (\text{if } b \text{ then } t_1 \text{ else } t_2) \{Q\}} \text{IF}$$

Exercise: define the characteristic formula for conditionals.

# Characteristic formula for conditionals

$$\frac{(b = \text{true} \Rightarrow \{H\} t_1 \{Q\}) \quad (b = \text{false} \Rightarrow \{H\} t_2 \{Q\})}{\{H\} (\text{if } b \text{ then } t_1 \text{ else } t_2) \{Q\}} \text{IF}$$

Exercise: define the characteristic formula for conditionals.

$$\begin{aligned} \llbracket \text{if } b \text{ then } t_1 \text{ else } t_2 \rrbracket &\equiv \lambda H Q. \quad (b = \text{true} \Rightarrow \llbracket t_1 \rrbracket H Q) \\ &\quad \wedge \quad (b = \text{false} \Rightarrow \llbracket t_2 \rrbracket H Q) \end{aligned}$$

# The App predicate

The goal of characteristic formulas is do proofs without involving triples.

Let “App” be an abstract predicate with the following interpretation:

$$\text{App } f v H Q \quad \Leftrightarrow \quad \{H\} (f v) \{Q\}$$

Remark:

$$\text{App} : \text{Val} \rightarrow \text{Val} \rightarrow \text{Hprop} \rightarrow (\text{Val} \rightarrow \text{Hprop}) \rightarrow \text{Prop}$$

# Reasoning about function calls

Interpretation of App:  $\text{App } f v H Q \Leftrightarrow \{H\} (f v) \{Q\}$

Interpretation of formulas:  $\llbracket f v \rrbracket H Q \Leftrightarrow \{H\} (f v) \{Q\}$

Definition:

$$\llbracket f v \rrbracket \equiv \lambda H Q. \text{App } f v H Q$$

# Reasoning about function calls

Interpretation of App:  $\text{App } f v H Q \Leftrightarrow \{H\} (f v) \{Q\}$

Interpretation of formulas:  $\llbracket f v \rrbracket H Q \Leftrightarrow \{H\} (f v) \{Q\}$

Definition:

$$\llbracket f v \rrbracket \equiv \lambda H Q. \text{App } f v H Q$$

Instances of App are used on calls and introduced on function definitions.



## Reasoning about function definitions

$$\frac{\forall f. Pf \Rightarrow \{H\} t_2 \{Q\} \quad Pf = (\forall x H' Q'. \{H'\} t_1 \{Q'\} \Rightarrow \{H'\} (f x) \{Q'\})}{\{H\} (\text{let rec } f x = t_1 \text{ in } t_2) \{Q\}} \text{FIX}$$

Definition:

$$\llbracket \text{let rec } f = \lambda x. t_1 \text{ in } t_2 \rrbracket \equiv \lambda H Q. \forall f. Pf \Rightarrow \llbracket t_2 \rrbracket H Q$$

$$\text{where } Pf \equiv (\forall x H' Q'. \llbracket t_1 \rrbracket H' Q' \Rightarrow \text{App } f x H' Q')$$

## Complete definition

$$\llbracket v \rrbracket \equiv \lambda H Q. H \triangleright Q v$$

$$\llbracket \text{let } x = t_1 \text{ in } t_2 \rrbracket \equiv \lambda H Q. \exists Q'. \llbracket t_1 \rrbracket H Q' \wedge \forall x. \llbracket t_2 \rrbracket (Q' x) Q$$

$$\llbracket \text{if } b \text{ then } t_1 \text{ else } t_2 \rrbracket \equiv \lambda H Q. \begin{aligned} & (b = \text{true} \Rightarrow \llbracket t_1 \rrbracket H Q) \\ & \wedge (b = \text{false} \Rightarrow \llbracket t_2 \rrbracket H Q) \end{aligned}$$

$$\llbracket v_1 v_2 \rrbracket \equiv \lambda H Q. \text{App } v_1 v_2 H Q$$

$$\llbracket \text{let rec } f = \lambda x. t_1 \text{ in } t_2 \rrbracket \equiv \lambda H Q. \forall f. P f \Rightarrow \llbracket t_2 \rrbracket H Q$$

$$\text{where } P f \equiv (\forall x H' Q'. \llbracket t_1 \rrbracket H' Q' \Rightarrow \text{App } f x H' Q')$$

The end!