

Pointer Programs, Separation Logic

Claude Marché

Cours MPRI 2-36-1 “Preuve de Programme”

February 27, 2013

Exercise of the last lecture

The algorithm that appends two lists *in place* follows this pseudo-code:

```
append(l1, l2 : loc) : loc
  if l1 is empty then return l2;
  let ref p = l1 in
  while p → next is not null do p := p → next;
  p → next := l2;
  return l1
```

1. Specify a post-condition giving the list models of both **result** and l2 (add any ghost variable needed)
2. Which pre-conditions and loop invariants are needed to prove this function?

Solution to the Exercise

```
append(l1,l2 : loc) (ghost l1M l2M:list loc) : loc
  requires list_seg(l1,next,l1M,null)
  requires list_seg(l2,next,l2M,null)
  requires disjoint(l1M,l2M)
  writes next
  ensures
    list_seg(result,next,list_append(l1M,l2M),null)
```

Invariants: see `linked_list_app.mlw`

Solution to the Exercise

Needed lemma:

```
lemma list_seg_append:  
  forall next:map loc loc,  
    p q r: loc, pM qM:list loc.  
  list_seg p next pM q  $\wedge$  list_seg q next qM r  $\rightarrow$   
  list_seg p next (append pM qM) r
```

Today's lecture

- ▶ Another approach to Pointer programs: Separation Logic

Outline

Basics of Separation Logic

Case of References à la OCaml

Case of Linked Lists

Separation Logic

No more encoding of pointers and memory heap

- ▶ programming language explicitly extended with pointers to records
- ▶ annotation language extended with
 - ▶ atoms specifying *available memory resources*
 - ▶ *separating* conjunction

Predicates like former *disjoint* and *no_repet* are in some sense *internalized*

Syntax of programs

- ▶ Set of declarations of record types:

$\text{record } S = \{f_1 : \tau_1; \dots f_n : \tau_n\}$

- ▶ Types: $\tau ::= \text{int} \mid \text{real} \mid \text{bool} \mid \text{unit} \mid S$

- ▶ Expressions:

$e ::= \dots$

former expression constructs
without mutable variables

$e \rightarrow f$	field access
$e \rightarrow f := e$	field update
$\text{new } S$	allocation
$\text{dispose } e$	deallocation

Semantics: Heaps

- ▶ Before: one heap, a *total* map from (loc,field) to values
- ▶ Now: many heaps, *partial* maps from (loc,field) to values

Notations: if h, h_1, h_2 are such partial maps:

- ▶ $\text{dom}(h)$: domain of h , i.e. the set of (loc,field) where it is defined
- ▶ $h = h_1 \oplus h_2$: h is the disjoint union of h_1 and h_2 , i.e.
 - ▶ $\text{dom}(h_1) \cap \text{dom}(h_2) = \emptyset$
 - ▶ $\text{dom}(h) = \text{dom}(h_1) \cup \text{dom}(h_2)$
 - ▶ $h(l, f) = h_1(l, f)$ if $l, f \in \text{dom}(h_1)$
 - ▶ $h(l, f) = h_2(l, f)$ if $l, f \in \text{dom}(h_2)$

Operational semantics

Relation

$$h, \Pi, e \rightsquigarrow h', \Pi', e'$$

where h is a partial heap

- ▶ Field access:

$$h, \Pi, (l \rightarrow f) \rightsquigarrow h, \Pi, v$$

if $l, f \in \text{dom}(h)$ and $h(l, f) = v$

blocks otherwise (“seg fault”)

- ▶ Field update:

$$h, \Pi, (l \rightarrow f := v) \rightsquigarrow h', \Pi, ()$$

if $l, f \in \text{dom}(h)$ and $h' = h\{(l, f) \leftarrow v\}$

blocks otherwise (“seg fault”)

Operational semantics

► Allocation

$$h, \Pi, (\text{new } S) \rightsquigarrow h', \Pi, l$$

where l fresh and $h' = h \oplus \{(l, f) \leftarrow \text{def}(\tau) \mid f : \tau \in S\}$
($\text{def}(\text{int}) = 0$, $\text{def}(S) = \text{null}$, etc.)

never blocks (no “memory overflow”)

► Deallocation

$$h, \Pi, (\text{dispose } l) \rightsquigarrow h', \Pi, ()$$

if

- for all field f of S , $(l, f) \in \text{dom}(h)$
- $h'(l', f') = h(l', f')$ if $l' \neq l$, undef otherwise

blocks otherwise (“seg fault”)

Example of execution

```
record List = { data : int, next: List }
```

```
[], []
```

```
let x = new List in
```

```
x→next := new List;
```

```
dispose(x→next);
```

```
x→next→data := 1
```

Example of execution

```
record List = { data : int, next: List }
```

```
[], []
```

```
let x = new List in
```

```
[(!0, data) = 0, (!0, next) = null], [x = !0]
```

```
x→next := new List;
```

```
dispose(x→next);
```

```
x→next→data := 1
```

Example of execution

```
record List = { data : int, next: List }
```

```
[], []
```

```
let x = new List in
```

```
[(!0, data) = 0, (!0, next) = null], [x = !0]
```

```
x→next := new List;
```

```
[(!0, data) = 0, (!0, next) = !1, (!1, data) = 0, (!1, next) = null],  
  [x = !0]
```

```
dispose(x→next);
```

```
x→next→data := 1
```

Example of execution

```
record List = { data : int, next: List }
```

```
[], []
```

```
let x = new List in
```

```
[(!0, data) = 0, (!0, next) = null], [x = !0]
```

```
x→next := new List;
```

```
[(!0, data) = 0, (!0, next) = !1, (!1, data) = 0, (!1, next) = null],  
  [x = !0]
```

```
dispose(x→next);
```

```
[(!0, data) = 0, (!0, next) = !1], [x = !0]
```

```
x→next→data := 1
```

Example of execution

```
record List = { data : int, next: List }
```

```
[], []
```

```
let x = new List in
```

```
[(!0, data) = 0, (!0, next) = null], [x = !0]
```

```
x→next := new List;
```

```
[(!0, data) = 0, (!0, next) = !1, (!1, data) = 0, (!1, next) = null],  
  [x = !0]
```

```
dispose(x→next);
```

```
[(!0, data) = 0, (!0, next) = !1], [x = !0]
```

```
x→next→data := 1
```

```
“seg fault”
```


Logic Formulas

The set of terms remains *unchanged*

$e \rightarrow f$ is not a term in the logic

New grammar for formulas:

$P, Q ::=$...	former formula constructs
	emp	empty heap
	$t_1 \xrightarrow{f} t_2$	memory chunk
	$P * Q$	separating conjunction

where

- ▶ t_1 is a term of type S for some record type S
- ▶ f is a field of type τ in S
- ▶ t_2 is a term of type τ

The 3 new constructs allow to describe *finite portions of the memory heap*

Example

```
record S { f : int }
```

```
function reset_f(x:S):unit
```

```
  requires ?   how do we say “allocated”
```

```
  ensures ?   we can't write “x->f = 0” !
```

```
  body x->f := 0
```

Semantics of formulas

Interpretation $\llbracket P \rrbracket_{h,\Pi}$

- ▶ Special formula **emp**:

$\llbracket \text{emp} \rrbracket_{h,\Pi}$ valid iff $h = \emptyset$

- ▶ Memory chunk: $\llbracket t_1 \xrightarrow{f} t_2 \rrbracket_{h,\Pi}$ iff

- ▶ $\llbracket t_1 \rrbracket_{\Pi} = l$ for some location l
- ▶ $\text{dom}(h) = \{(l, f)\}$
- ▶ $h(l, f) = \llbracket t_2 \rrbracket_{\Pi}$

- ▶ Separating conjunction: $\llbracket P * Q \rrbracket_{h,\Pi}$ is valid iff there exists h_1, h_2 such that

- ▶ $h = h_1 \oplus h_2$
- ▶ $\llbracket P \rrbracket_{h_1,\Pi}$ is valid
- ▶ $\llbracket Q \rrbracket_{h_2,\Pi}$ is valid

Examples

$$\Pi = [x = l_0]$$

$$h_1 = [(l_0, next) = l_1]$$

$$h_2 = [(l_0, next) = l_1, (l_0, data) = 42]$$

$$h_3 = [(l_0, next) = l_1, (l_1, next) = null]$$

valid in ?	h_1	h_2	h_3
$x \xrightarrow{next} l_1$			
$x \xrightarrow{next} l_1 * x \xrightarrow{data} 42$			
$x \xrightarrow{next} l_1 * l_1 \xrightarrow{next} null$			
emp			

Examples

$$\Pi = [x = l_0]$$

$$h_1 = [(l_0, next) = l_1]$$

$$h_2 = [(l_0, next) = l_1, (l_0, data) = 42]$$

$$h_3 = [(l_0, next) = l_1, (l_1, next) = null]$$

valid in ?	h_1	h_2	h_3
$x \xrightarrow{next} l_1$	Y	N	N
$x \xrightarrow{next} l_1 * x \xrightarrow{data} 42$	N	Y	N
$x \xrightarrow{next} l_1 * l_1 \xrightarrow{next} null$	N	N	Y
emp	N	N	N

Properties of Separating Conjunction

- ▶ $(P * Q) * R \leftrightarrow P * (Q * R)$
- ▶ $P * Q \leftrightarrow Q * P$
- ▶ $\text{emp} * P \leftrightarrow P$
- ▶ if P, Q memory-free formulas, $P * Q \leftrightarrow P \wedge Q$

Caution! P not equivalent to $P * P$
(*linearity* of the separating conjunction)

Issue regarding memory-free formulas

- ▶ Formula $x \xrightarrow{\text{next}} l_1$ is valid only in heap h_1
- ▶ Conjunction $x \xrightarrow{\text{next}} l_1 * \text{true}$ is valid also in heap h_1 and h_2 .
- ▶ Variant of the semantics: consider that a memory-free formula is valid only in an empty heap.

$\llbracket \phi \rrbracket_{h, \Pi}$ valid iff $h = \emptyset$ and $\llbracket \phi \rrbracket_{\Pi}$

For now, we adopt this semantics

A language fragment: Symbolic Heaps

Classical fragment of Separation Logic: *Symbolic Heaps*

Only formulas of the form

$$\exists v_1, \dots, v_n, P_1 * P_2 * \dots * P_k * \phi$$

where ϕ is a memory-free formula and the P_i are memory chunks

For now, we consider this fragment only

Simple Example, continued

```
record S { f : int }  
  
function reset_f(x:S):unit  
  requires  
  ensures  
  body x→f := 0
```

Simple Example, continued

```
record S { f : int }
```

```
function reset_f(x:S):unit
```

```
  requires
```

```
  ensures  $x \xrightarrow{f} 0$ 
```

```
  body  $x \rightarrow f := 0$ 
```

Simple Example, continued

```
record S { f : int }
```

```
function reset_f(x:S):unit
```

```
  requires    $\exists v. x \xrightarrow{f} v$ 
```

```
  ensures    $x \xrightarrow{f} 0$ 
```

```
  body x → f := 0
```

Another Simple Example

```
record S { f : int }
```

```
function incr_f(x:S)                                :unit
```

```
  requires
```

```
  ensures
```

```
  body x→f := x→f + 1
```

Another Simple Example

```
record S { f : int }
```

```
function incr_f(x:S) (ghost v:int):unit
```

```
  requires    $x \xrightarrow{f} v$ 
```

```
  ensures    $x \xrightarrow{f} v + 1$ 
```

```
  body  $x \rightarrow f := x \rightarrow f + 1$ 
```

Syntactic Sugar

- ▶ implicit existential quantification
- ▶ quantified variables starting with **'_'**

```
function reset_f(x:S):unit  
  requires    $x \xrightarrow{f} \_$   
  ensures    $x \xrightarrow{f} 0$   
  body  $x \rightarrow f := 0$ 
```

```
function incr_f(x:S) :unit  
  requires    $x \xrightarrow{f} \_v$   
  ensures    $x \xrightarrow{f} \_v + 1$   
  body  $x \rightarrow f := x \rightarrow f + 1$ 
```

Separation Logic Rules

Hoare style rules

- ▶ Field access

$$\frac{}{\{l \overset{f}{\mapsto} v\} l \rightarrow f \{l \overset{f}{\mapsto} v * \text{result} = v\}}$$

- ▶ Field update

$$\frac{}{\{l \overset{f}{\mapsto} v\} l \rightarrow f := v' \{l \overset{f}{\mapsto} v' * \text{result} = ()\}}$$

- ▶ Frame rule

$$\frac{\{P\} e \{Q\}}{\{P * R\} e \{Q * R\}}$$

Separation Logic Rules

► Allocation

$$\{\text{emp}\} \text{new } S\{ _ / \stackrel{f_1}{\mapsto} \text{def}(\tau_1) * \dots * _ / \stackrel{f_n}{\mapsto} \text{def}(\tau_n) * \text{result} = _ / \}$$

► Deallocation

$$\{ _ / \stackrel{f_1}{\mapsto} v_1 * \dots * _ / \stackrel{f_n}{\mapsto} v_n \} \text{dispose } _ / \{ \text{emp} \}$$

Separation Logic and Symbolic Execution

Rules are close to operational semantics: can be used as a kind of *Symbolic execution*

Example:

```
{emp}
```

```
let x = new List in
```

```
x->data := 42
```

```
x->next := new List
```

Separation Logic and Symbolic Execution

Rules are close to operational semantics: can be used as a kind of *Symbolic execution*

Example:

```
{emp}  
let x = new List in  
{( $\_l \xrightarrow{data} 0$ ) * ( $\_l \xrightarrow{next} null$ ) * (x =  $\_l$ )}
```

x->data := 42

x->next := new List

Separation Logic and Symbolic Execution

Rules are close to operational semantics: can be used as a kind of *Symbolic execution*

Example:

```
{emp}  
let x = new List in  
{( $\_l \xrightarrow{data} 0$ ) * ( $\_l \xrightarrow{next} null$ ) * (x =  $\_l$ )}  
{( $x \xrightarrow{data} 0$ ) * ( $x \xrightarrow{next} null$ )}  
x->data := 42
```

```
x->next := new List
```

Separation Logic and Symbolic Execution

Rules are close to operational semantics: can be used as a kind of *Symbolic execution*

Example:

```
{emp}
let x = new List in
{( $\_l \overset{data}{\mapsto} 0$ ) * ( $\_l \overset{next}{\mapsto} null$ ) * ( $x = \_l$ )}
{( $x \overset{data}{\mapsto} 0$ ) * ( $x \overset{next}{\mapsto} null$ )}
x->data := 42
{( $x \overset{data}{\mapsto} 42$ ) * ( $x \overset{next}{\mapsto} null$ )} (frame rule)
x->next := new List
```

Separation Logic and Symbolic Execution

Rules are close to operational semantics: can be used as a kind of *Symbolic execution*

Example:

{emp}

let x = new List in

$\{(_l \xrightarrow{\text{data}} 0) * (_l \xrightarrow{\text{next}} \text{null}) * (x = _l)\}$

$\{(x \xrightarrow{\text{data}} 0) * (x \xrightarrow{\text{next}} \text{null})\}$

x->data := 42

$\{(x \xrightarrow{\text{data}} 42) * (x \xrightarrow{\text{next}} \text{null})\}$ (frame rule)

x->next := new List

$\{(x \xrightarrow{\text{data}} 42) * (x \xrightarrow{\text{next}} _l) * (_l \xrightarrow{\text{data}} 0) * (_l \xrightarrow{\text{next}} \text{null})\}$

Outline

Basics of Separation Logic

Case of References à la OCaml

Case of Linked Lists

References as mutable records

- ▶ We have no mutable variables anymore
- ▶ But they can be simulated using a pointer to a record with one field only

```
record Ref  $\alpha$  = { contents :  $\alpha$  }
```

```
function ref (x:  $\alpha$ ) : Ref  $\alpha$   
  body let r = new Ref in r $\rightarrow$ contents := x; r
```

```
function (!) (r: Ref  $\alpha$ ) :  $\alpha$   
  body r $\rightarrow$ contents
```

```
function (:=) (r: Ref  $\alpha$ ) (x: $\alpha$ ) : unit  
  body r $\rightarrow$ contents := x
```

Specifications

- For readability we abbreviate $r \xrightarrow{\text{contents}} t$ into $r \mapsto t$

```
function ref (x:  $\alpha$ ) : Ref  $\alpha$   
  requires  
  ensures
```

```
function (!) (r: Ref  $\alpha$ ) :  $\alpha$   
  requires  
  ensures
```

```
function (:=) (r: Ref  $\alpha$ ) (x: $\alpha$ ) : unit  
  requires  
  ensures
```


Specifications

- For readability we abbreviate $r \overset{\text{contents}}{\mapsto} t$ into $r \mapsto t$

```
function ref (x:  $\alpha$ ) : Ref  $\alpha$   
  requires emp  
  ensures
```

```
function (!) (r: Ref  $\alpha$ ) :  $\alpha$   
  requires  
  ensures
```

```
function (:=) (r: Ref  $\alpha$ ) (x: $\alpha$ ) : unit  
  requires  
  ensures
```

Specifications

- For readability we abbreviate $r \overset{\text{contents}}{\mapsto} t$ into $r \mapsto t$

```
function ref (x:  $\alpha$ ) : Ref  $\alpha$   
  requires  emp  
  ensures  result  $\mapsto$  x
```

```
function (!) (r: Ref  $\alpha$ ) :  $\alpha$   
  requires  
  ensures
```

```
function (:=) (r: Ref  $\alpha$ ) (x:  $\alpha$ ) : unit  
  requires  
  ensures
```

Specifications

- For readability we abbreviate $r \overset{\text{contents}}{\mapsto} t$ into $r \mapsto t$

```
function ref (x:  $\alpha$ ) : Ref  $\alpha$   
  requires emp  
  ensures result  $\mapsto x$ 
```

```
function (!) (r: Ref  $\alpha$ ) :  $\alpha$   
  requires r  $\mapsto \_V$   
  ensures
```

```
function (:=) (r: Ref  $\alpha$ ) (x:  $\alpha$ ) : unit  
  requires  
  ensures
```

Specifications

- For readability we abbreviate $r \overset{\text{contents}}{\mapsto} t$ into $r \mapsto t$

```
function ref (x:  $\alpha$ ) : Ref  $\alpha$   
  requires emp  
  ensures result  $\mapsto$  x
```

```
function (!) (r: Ref  $\alpha$ ) :  $\alpha$   
  requires r  $\mapsto$  _v  
  ensures r  $\mapsto$  _v * result = _v
```

```
function (:=) (r: Ref  $\alpha$ ) (x:  $\alpha$ ) : unit  
  requires  
  ensures
```

Specifications

- For readability we abbreviate $r \overset{\text{contents}}{\mapsto} t$ into $r \mapsto t$

```
function ref (x:  $\alpha$ ) : Ref  $\alpha$   
  requires emp  
  ensures result  $\mapsto$  x
```

```
function (!) (r: Ref  $\alpha$ ) :  $\alpha$   
  requires r  $\mapsto$  _v  
  ensures r  $\mapsto$  _v * result = _v
```

```
function (:=) (r: Ref  $\alpha$ ) (x:  $\alpha$ ) : unit  
  requires r  $\mapsto$  _  
  ensures
```

Specifications

- For readability we abbreviate $r \xrightarrow{\text{contents}} t$ into $r \mapsto t$

```
function ref (x:  $\alpha$ ) : Ref  $\alpha$   
  requires   emp  
  ensures   result  $\mapsto x$ 
```

```
function (!) (r: Ref  $\alpha$ ) :  $\alpha$   
  requires   r  $\mapsto \_v$   
  ensures   r  $\mapsto \_v * \text{result} = \_v$ 
```

```
function (:=) (r: Ref  $\alpha$ ) (x:  $\alpha$ ) : unit  
  requires   r  $\mapsto \_$   
  ensures   r  $\mapsto x$ 
```

Outline

Basics of Separation Logic

Case of References à la OCaml

Case of Linked Lists

Case of Linked Lists

Inductive predicate *ls* (list segment)

inductive *ls*(*List*, *List*) =

| *ls_nil*: $\forall x : \text{List}, \text{ls}(x, x)$

| *ls_cons*: $\forall xyz : \text{List}, (x \xrightarrow{\text{next}} y) * \text{ls}(y, z) \rightarrow \text{ls}(x, z)$

- State lemmas, e.g.

$$\text{ls}(x, y) \leftrightarrow x = y \vee \exists z, (x \xrightarrow{\text{next}} z) * \text{ls}(z, y)$$

- *Symbolic execution rules*, e.g.

$$\text{ls}(x, y) * x \neq y \rightsquigarrow (x \xrightarrow{\text{next}} _z) * \text{ls}(_z, y) * x \neq y$$

Example: in-place list reversal

```
function reverse (l:List) : List =  
  requires  
  ensures  
  body  
    let p = ref l in  
    let r = ref null in  
    while !p  $\neq$  null do  
      invariant  
      let n = !p $\rightarrow$ next in  
      !p $\rightarrow$ next := r;  
      r := !p;  
      p := n;  
  done;  
  !r
```

Example: in-place list reversal

```
function reverse (l:List) : List =  
  requires   ls(l, null)  
  ensures  
  body  
    let p = ref l in  
    let r = ref null in  
    while !p  $\neq$  null do  
      invariant  
      let n = !p $\rightarrow$ next in  
      !p $\rightarrow$ next := r;  
      r := !p;  
      p := n;  
  done;  
  !r
```

Example: in-place list reversal

```
function reverse (l:List) : List =  
  requires   ls(l, null)  
  ensures   ls(result, null)  
  body  
    let p = ref l in  
    let r = ref null in  
    while !p  $\neq$  null do  
      invariant  
      let n = !p  $\rightarrow$  next in  
      !p  $\rightarrow$  next := r;  
      r := !p;  
      p := n;  
  done;  
  !r
```

Example: in-place list reversal

```
function reverse (l:List) : List =  
  requires   ls(l, null)  
  ensures   ls(result, null)  
  body  
    let p = ref l in  
    let r = ref null in  
    while !p  $\neq$  null do  
      invariant   p  $\mapsto$  lp * ls(lp, null) * r  $\mapsto$  lr * ls(lr, null)  
      let n = !p  $\rightarrow$  next in  
      !p  $\rightarrow$  next := r;  
      r := !p;  
      p := n;  
  done;  
  !r
```

Example: in-place list reversal

$\{ls(l, null)\}$

let $p = \text{ref } l$ in

let $r = \text{ref } null$ in

while ($!p \text{ <> } null$) do

invariant $p \mapsto l_p * ls(l_p, null) * r \mapsto l_r * ls(l_r, null)$

Example: in-place list reversal

$\{ls(l, null)\}$

let $p = \text{ref } l$ in

$\{p \mapsto l * ls(l, null)\}$

let $r = \text{ref } null$ in

while ($!p \text{ <> } null$) do

invariant $p \mapsto l_p * ls(l_p, null) * r \mapsto l_r * ls(l_r, null)$

Example: in-place list reversal

$\{ls(l, null)\}$

let $p = \text{ref } l$ in

$\{p \mapsto l * ls(l, null)\}$

let $r = \text{ref } null$ in

$\{p \mapsto l * ls(l, null) * r \mapsto null\}$

while ($!p <> null$) do

invariant $p \mapsto l_p * ls(l_p, null) * r \mapsto l_r * ls(l_r, null)$

Example: in-place list reversal

$\{ls(l, null)\}$

let $p = \text{ref } l$ in

$\{p \mapsto l * ls(l, null)\}$

let $r = \text{ref null}$ in

$\{p \mapsto l * ls(l, null) * r \mapsto null\}$

$\{p \mapsto l * ls(l, null) * r \mapsto null * ls(null, null)\}$ (symb. exec. rule)

while ($!p <> null$) do

invariant $p \mapsto l_p * ls(l_p, null) * r \mapsto l_r * ls(l_r, null)$

Example: in-place list reversal

while (!p <> null) do

invariant $p \mapsto l_p * ls(l_p, null) * r \mapsto l_r * ls(l_r, null)$

let n = !p->next in

!p->next := !r;

r := !p;

p := n

Example: in-place list reversal

while (!p <> null) do

invariant $p \mapsto l_p * ls(l_p, null) * r \mapsto l_r * ls(l_r, null)$

$\{p \mapsto l_p * ls(l_p, null) * r \mapsto l_r * ls(l_r, null) * l_p \neq null\}$

let n = !p->next in

!p->next := !r;

r := !p;

p := n

Example: in-place list reversal

while (!p <> null) do

invariant $p \mapsto l_p * ls(l_p, null) * r \mapsto l_r * ls(l_r, null)$

$\{p \mapsto l_p * ls(l_p, null) * r \mapsto l_r * ls(l_r, null) * l_p \neq null\}$

$\{p \mapsto l_p * l_p \xrightarrow{next} _q * ls(_q, null) * r \mapsto l_r * ls(l_r, null) * l_p \neq null\}$

let n = !p->next in

!p->next := !r;

r := !p;

p := n

Example: in-place list reversal

while (!p <> null) do

invariant $p \mapsto l_p * ls(l_p, null) * r \mapsto l_r * ls(l_r, null)$

$\{p \mapsto l_p * ls(l_p, null) * r \mapsto l_r * ls(l_r, null) * l_p \neq null\}$

$\{p \mapsto l_p * l_p \xrightarrow{next} _q * ls(_q, null) * r \mapsto l_r * ls(l_r, null) * l_p \neq null\}$

let n = !p->next in

$\{p \mapsto l_p * l_p \xrightarrow{next} n * ls(n, null) * r \mapsto l_r * ls(l_r, null) * l_p \neq null\}$

!p->next := !r;

r := !p;

p := n

Example: in-place list reversal

while (!p <> null) do

invariant $p \mapsto l_p * ls(l_p, null) * r \mapsto l_r * ls(l_r, null)$

$\{p \mapsto l_p * ls(l_p, null) * r \mapsto l_r * ls(l_r, null) * l_p \neq null\}$

$\{p \mapsto l_p * l_p \xrightarrow{next} _q * ls(_q, null) * r \mapsto l_r * ls(l_r, null) * l_p \neq null\}$

let n = !p->next in

$\{p \mapsto l_p * l_p \xrightarrow{next} n * ls(n, null) * r \mapsto l_r * ls(l_r, null) * l_p \neq null\}$

!p->next := !r;

$\{p \mapsto l_p * l_p \xrightarrow{next} l_r * ls(n, null) * r \mapsto l_r * ls(l_r, null) * l_p \neq null\}$

r := !p;

p := n

Example: in-place list reversal

while (!p <> null) do

invariant $p \mapsto l_p * ls(l_p, null) * r \mapsto l_r * ls(l_r, null)$

$\{p \mapsto l_p * ls(l_p, null) * r \mapsto l_r * ls(l_r, null) * l_p \neq null\}$

$\{p \mapsto l_p * l_p \xrightarrow{next} _q * ls(_q, null) * r \mapsto l_r * ls(l_r, null) * l_p \neq null\}$

let n = !p->next in

$\{p \mapsto l_p * l_p \xrightarrow{next} n * ls(n, null) * r \mapsto l_r * ls(l_r, null) * l_p \neq null\}$

!p->next := !r;

$\{p \mapsto l_p * l_p \xrightarrow{next} l_r * ls(n, null) * r \mapsto l_r * ls(l_r, null) * l_p \neq null\}$

r := !p;

$\{p \mapsto l_p * l_p \xrightarrow{next} l_r * ls(n, null) * r \mapsto l_p * ls(l_r, null) * l_p \neq null\}$

p := n

Example: in-place list reversal

while (!p <> null) do

invariant $p \mapsto l_p * ls(l_p, null) * r \mapsto l_r * ls(l_r, null)$

$\{p \mapsto l_p * ls(l_p, null) * r \mapsto l_r * ls(l_r, null) * l_p \neq null\}$

$\{p \mapsto l_p * l_p \xrightarrow{next} _q * ls(_q, null) * r \mapsto l_r * ls(l_r, null) * l_p \neq null\}$

let n = !p->next in

$\{p \mapsto l_p * l_p \xrightarrow{next} n * ls(n, null) * r \mapsto l_r * ls(l_r, null) * l_p \neq null\}$

!p->next := !r;

$\{p \mapsto l_p * l_p \xrightarrow{next} l_r * ls(n, null) * r \mapsto l_r * ls(l_r, null) * l_p \neq null\}$

r := !p;

$\{p \mapsto l_p * l_p \xrightarrow{next} l_r * ls(n, null) * r \mapsto l_p * ls(l_r, null) * l_p \neq null\}$

p := n

$\{p \mapsto n * l_p \xrightarrow{next} l_r * ls(n, null) * r \mapsto l_p * ls(l_r, null) * l_p \neq null\}$

Example: in-place list reversal

while (!p <> null) do

invariant $p \mapsto l_p * ls(l_p, null) * r \mapsto l_r * ls(l_r, null)$

$\{p \mapsto l_p * ls(l_p, null) * r \mapsto l_r * ls(l_r, null) * l_p \neq null\}$

$\{p \mapsto l_p * l_p \xrightarrow{next} _q * ls(_q, null) * r \mapsto l_r * ls(l_r, null) * l_p \neq null\}$

let n = !p->next in

$\{p \mapsto l_p * l_p \xrightarrow{next} n * ls(n, null) * r \mapsto l_r * ls(l_r, null) * l_p \neq null\}$

!p->next := !r;

$\{p \mapsto l_p * l_p \xrightarrow{next} l_r * ls(n, null) * r \mapsto l_r * ls(l_r, null) * l_p \neq null\}$

r := !p;

$\{p \mapsto l_p * l_p \xrightarrow{next} l_r * ls(n, null) * r \mapsto l_p * ls(l_r, null) * l_p \neq null\}$

p := n

$\{p \mapsto n * l_p \xrightarrow{next} l_r * ls(n, null) * r \mapsto l_p * ls(l_r, null) * l_p \neq null\}$

$\{p \mapsto n * ls(n, null) * r \mapsto l_p * ls(l_p, null)\}$

Example: in-place list reversal

```
while (!p <> null) do  
  invariant  $p \mapsto l_p * ls(l_p, null) * r \mapsto l_r * ls(l_r, null)$   
done
```

!r

Example: in-place list reversal

```
while (!p <> null) do
  invariant  $p \mapsto l_p * ls(l_p, null) * r \mapsto l_r * ls(l_r, null)$ 
done
 $\{p \mapsto l_p * ls(l_p, null) * r \mapsto l_r * ls(l_r, null) * l_p = null\}$ 

!r
```

Example: in-place list reversal

```
while (!p <> null) do
  invariant  $p \mapsto l_p * ls(l_p, null) * r \mapsto l_r * ls(l_r, null)$ 
done
 $\{p \mapsto l_p * ls(l_p, null) * r \mapsto l_r * ls(l_r, null) * l_p = null\}$ 
 $\{p \mapsto null * r \mapsto l_r * ls(l_r, null)\}$ 
!r
```

Example: in-place list reversal

```
while (!p <> null) do
  invariant  $p \mapsto l_p * ls(l_p, null) * r \mapsto l_r * ls(l_r, null)$ 
done
 $\{p \mapsto l_p * ls(l_p, null) * r \mapsto l_r * ls(l_r, null) * l_p = null\}$ 
 $\{p \mapsto null * r \mapsto l_r * ls(l_r, null)\}$ 
!r
 $\{ls(result, null)\}$  (implicit garbage collecting!)
```

Exercise: in-place append

```
append(l1, l2 : loc) : loc
  if l1 is empty then return l2;
  p := l1;
  while p→next is not null do p := p→ next;
  p → next := l2;
  return l1
```

Final Remarks on Separation Logic

- ▶ Internalize disjointness and frame properties
- ▶ Extensions/Applications
 - ▶ preservation of data invariants
 - ▶ concurrent programs
- ▶ Negative point: low level of automation
 - ▶ No simple equivalent of WP
(some equivalent of WP using *magic wand*)
 - ▶ SMT solvers cannot be used directly

Exercise from Last year's exam

The following program takes a list l of integers as input and returns two lists (l_1, l_2) where l_1 contains the nonnegative elements of l , and l_2 contains the negative ones.

```
record IntList = { data : int ; next: IntList; }  
function split(l:IntList):(IntList,IntList)  
  body  
    let l1 = ref null in let l2 = ref null in  
    let p = ref l in  
    while p  $\neq$  null do  
      let n = p in  
      p := p→next;  
      if n→data  $\geq$  0 then  
        n→next := l1; l1 := n  
      else  
        n→next := l2; l2 := n  
    done;  
    (l1,l2)
```

Informally, the specification we want is

- ▶ the input list must be null-terminated
- ▶ the two output lists are null-terminated
- ▶ the list l_1 contains only nonnegative values
- ▶ the list l_2 contains only negative values
- ▶ all values appearing in l_1 and l_2 must already appear in the input list.

Notice that this specification does not require that all values of the input list should appear either in l_1 or l_2 .

1. Propose an equivalent program without pointers, using the Component-as-Array model, with appropriate pre- and postconditions. It is recommended to use the predicate $\text{mem}(x:\alpha, l:\text{list } \alpha)$ that tells whether a given element x appears in a pure logic list l .
2. Propose an appropriate loop invariant for the program of the previous question. Explain informally why your loop invariant is enough to prove the program, in particular regarding separation issues.

We now consider the Separation Logic approach instead of the Component-as-Array model.

1. Propose inductive predicates to represent list segments containing respectively any values, nonnegative values, and negative values.
2. Specify the program in Separation Logic. Propose a loop invariant and explain informally how the proof proceeds, in particular regarding separation issues.

Next week

- ▶ March 4: deadline for the project
- ▶ March 6: Exam
 - ▶ Room 1008
 - ▶ 16:15-19:15

Allowed documents

lecture notes, personal notes, books, etc. but **no electronic device**