

Aliasing Issues: Call by reference, Pointer programs

Claude Marché (François Bobot)

Cours MPRI 2-36-1 "Preuve de Programme"

January 12th, 2023

Reminder of the last lecture

- ▶ Ghost variables, ghost functions, lemma functions
- ▶ Additional features of the specification language
 - ▶ Sum Types, e.g. *lists*
- ▶ Programs on *lists*
- ▶ Additional feature of the programming language
 - ▶ *Exceptions*
 - ▶ Function contracts extended with *exceptional post-conditions*
- ▶ Fast WP using SSA
- ▶ Computer Arithmetic: *bounded integers*, *floating-point numbers*
- ▶ A few home work to do

Home Work: Bézout coefficients

- ▶ Extend the post-condition of Euclid's algorithm for GCD to express the Bézout property:

$$\exists a, b, \text{result} = x * a + y * b$$

- ▶ Prove the program by adding appropriate ghost local variables

Use canvas file [exo_bezout.mlw](#)

Home work: lemmas on exponentiation

Prove the helper lemmas stated for the fast exponentiation algorithm

See [power_int_lemma_functions.mlw](#)

Home Work

Prove Fermat's little theorem for case $p = 3$:

$$\forall x, \exists y. x^3 - x = 3y$$

using a lemma function

See [little_fermat_3.mlw](#)

Home Work: Binary Search with an exception

```
low = 0; high = a.length - 1;  
while low ≤ high:  
  let m be the middle of low and high  
  if a[m] = v then return m  
  if a[m] < v then continue search between m and high  
  if a[m] > v then continue search between low and m
```

See file [bin_search_exc.mlw](#)

Introducing Aliasing Issues

Compound data structures can be *modeled* using expressive specification languages

- ▶ Defined functions and predicates
- ▶ Product types (records)
- ▶ Sum types (lists, trees)
- ▶ Axiomatizations (arrays, machine integers)
- ▶ Ghost code, lemma functions

Important points:

- ▶ *pure* types, no internal “in-place” assignment
- ▶ Mutable variables = *references to pure types*

No Aliasing

Aliasing

Aliasing = two different “names” for the same mutable data

Two sub-topics of today's lecture:

- ▶ Call by reference
- ▶ Pointer programs

Outline

Call by Reference

The Framing Issue

Pointer Programs

Need for call by reference

Example: stacks of integers

```
type stack = list int

val ref s: stack

let push(x:int):unit
  writes s
  ensures s = Cons(x,s@old)
  body ...

let pop(): int
  requires s <> Nil
  writes s
  ensures result = head(s@old) /\ s = tail(s@old)
```

Need for call by reference

If we need two stacks in the same program:

- We don't want to write the functions twice!

We want to write

```
type stack = list int

let push(ref s: stack, x:int): unit
  writes s
  ensures s = Cons(x,s@old)
  ...

let pop(ref s: stack):int
  ...
```

Call by Reference: example

```
val ref s1,s2: stack

let test():
  writes s1, s2
  ensures result = 13 /\ head(s2) = 42
  body push(s1,13); push(s2,42); pop(s1)
```

- See file [stack1.mlw](#)

Aliasing problems

```

let test(ref s3,s4: stack) : unit
  writes s3, s4
  ensures { head(s3) = 13 /\ head(s4) = 42 }
  body push(s3,13); push(s4,42)

let wrong(ref s5: stack) : int
  writes s5
  ensures { head(s5) = 13 /\ head(s5) = 42 }
    something's wrong !?
  body test(s5,s5)

```

Aliasing is a major issue

Deductive Verification Methods like Hoare logic, Weakest Precondition Calculus implicitly require absence of aliasing

Syntax

- Declaration of functions: (references first for simplicity)

let $f(\text{ref } y_1 : \tau_1, \dots, \text{ref } y_k : \tau_k, x_1 : \tau'_1, \dots, x_n : \tau'_n) :$
 ...

- Call:

$f(z_1, \dots, z_k, e_1, \dots, e_n)$

where each z_i must be a mutable variable

Operational Semantics

Intuitive semantics, by substitution:

$$\frac{\pi = \{x_i \mapsto \llbracket t_i \rrbracket_{\Sigma, \pi}\} \quad \Sigma, \pi \models \text{Pre} \quad \text{Body}' = \text{Body}[y_j \leftarrow z_j]}{\Sigma, \Pi, f(t_1, \dots, t_n) \rightsquigarrow \Sigma, (\pi, \text{Post}) \cdot \Pi, (\text{Old} : \text{Body}')}$$

- The body is executed, where each occurrence of reference parameters are replaced by the corresponding reference argument.
- Not a “practical” semantics, but that’s not important. . .

Operational Semantics

Variant: Semantics by copy/restore:

$$\frac{\pi = \{y_j \mapsto \Sigma(z_j), x_i \mapsto \llbracket t_i \rrbracket_{\Sigma, \pi}\} \quad \Sigma, \pi \models \text{Pre}}{\Sigma, \Pi, f(t_1, \dots, t_n) \rightsquigarrow \Sigma, (\pi, \text{Post}) \cdot \Pi, (\text{Old} : \text{Body})}$$

$$\frac{\Sigma, \pi \models \text{Post}[\text{result} \leftarrow v] \quad \Sigma' = \Sigma[z_j \leftarrow \pi(y_j)]}{\Sigma, (\pi, \text{Post}) \cdot \Pi, v \rightsquigarrow \Sigma', \Pi, v}$$

Warning: not the same semantics !

Difference in the semantics

```
val ref g : int

let f(ref x: int):unit
  body x <- 1; x <- g+1

let test():unit
  body g <- 0; f(g)
```

After executing test:

- ▶ Semantics by substitution: $g = 2$
- ▶ Semantics by copy/restore: $g = 1$

Aliasing Issues (1)

```
let f(ref x: int, ref y: int):
  writes x, y
  ensures x = 1 /\ y = 2
  body x <- 1; y <- 2

val ref g : int

let test():
  body
    f(g,g);
    assert g = 1 /\ g = 2 (* what's wrong? *)
```

- ▶ Aliasing of reference parameters

Aliasing Issues (2)

```
val ref g1 : int
val ref g2 : int

let p(ref x: int):
  writes g1, x
  ensures g1 = 1 /\ x = 2
  body g1 <- 1; x <- 2

let test():
  body
    p(g2); assert g1 = 1 /\ g2 = 2; (* OK *)
    p(g1); assert g1 = 1 /\ g1 = 2; (* what's wrong? *)
```

- ▶ Aliasing of a global variable and reference parameter

Aliasing Issues (3)

```
val ref g : int

val fun f(ref x: int):unit
  writes x
  ensures x = g + 1
  (* body x <- 1; x <- g+1 *)

let test():unit
  ensures { g = 1 or 2 ? }
  body g <- 0; f(g)
```

- ▶ Aliasing of a read reference and a written reference

Aliasing Issues (3)

New need in specifications

Need to *specify read references in contracts*

```
val ref g : int

val f(ref x: int):unit
  reads g          (* new clause in contract *)
  writes x
  ensures x = g + 1
  (* body x <- 1; x <- g+1 *)

let test():unit
  ensures { g = ? }
  body g <- 0; f(g)
```

► See file [stack2.mlw](#)

Typing: Alias-Freedom Conditions

For a function of the form

$f(\text{ref } y_1 : \tau_1, \dots, \text{ref } y_k : \tau_k, \dots) : \tau$
writes $\vec{w}$
reads $\vec{r}$

Typing rule for a call to f :

$$\frac{\dots \quad \forall i, j, i \neq j \rightarrow z_i \neq z_j \quad \forall i, j, z_i \neq w_j \quad \forall i, j, z_i \neq r_j}{\dots \vdash f(z_1, \dots, z_k, \dots) : \tau}$$

- effective arguments z_j must be distinct
- effective arguments z_j must not be already directly read nor written by f

Proof Rules

Thanks to restricted typing:

- Semantics by substitution and by copy/restore coincide
- Hoare rules remain correct
- WP rules remain correct

New references

- Need to return newly created references
- Example: stack continued

```
let create():ref stack
  ensures result = Nil
  body (ref Nil)
```

- Typing should require that a returned reference is always *fresh*

More on aliasing control using static typing: [\[Filliâtre, 2016\]](#)

Outline

Call by Reference

The Framing Issue

Pointer Programs

Introduction to Framing

(Example from exam 2017)

- ▶ Consider polynomials of the form $\sum_{i=0}^n c_i X^i$
- ▶ Representation: array of real numbers, len $n + 1$, i -th cell is c_i

Example: $P_0 = X^3 + 4X - 7$ is represented as array $[-7; 4; 0; 1]$

Polynomial Evaluation

Function eval

Formally interprets an array of reals as a polynomial function

```
let rec function eval_aux (p:array real) (x:real)
                        (i j:int) : real
= if j <= i then 0.0 else
  p[i] + x * eval_aux p x (i+1) j

function eval (p:array real) (x:real) : real =
  eval_aux p x 0 p.length
```

Example

```
eval P0 0.5
= eval_aux [-7; 4; 0; 1] 0.5 0 4
= (-7) + 0.5 * eval_aux [-7; 4; 0; 1] 0.5 1 4
= ⋮
= (-7) + 0.5 * (4 + 0.5 * (0 + 0.5 * 1))
```

Adding a constant to a polynomial

Function add_const

Adds a constant to a polynomial

```
let add_const (p:array real) (c:real) : unit
  requires { p.length >= 1 }
  writes { p }
  ensures { forall x. eval p x = eval (old p) x + c }
= p[0] <- p[0] + c
```

As such, this function is not proved automatically, why?

Need for a framing property

Let p' denote the array after assignment. Proving the post-condition requires to establish:

$$\text{eval } p' \ x = \text{eval } p \ x + c$$

that is, after unfolding eval:

$$\text{eval_aux } p' \ x \ 0 \ l = \text{eval_aux } p \ x \ 0 \ l + c$$

By expanding using the definition of eval_aux:

$$p'[0] + \text{eval_aux } p' \ x \ 1 \ l = p[0] + \text{eval_aux } p \ x \ 1 \ l + c$$

After simplification:

$$\text{eval_aux } p' \ x \ 1 \ l = \text{eval_aux } p \ x \ 1 \ l$$

Framing

To prove that p' is equal to p on the range $1 \dots l$, a *frame property* is needed

Frame property

Frame property for eval_aux

For any arrays p and q , if

$$\forall k. i \leq k < j \rightarrow p[k] = q[k]$$

then

$$\text{eval_aux } p \ x \ i \ j = \text{eval_aux } q \ x \ i \ j$$

A lemma function can be stated as follows to enforce a proof by induction on $j - i$:

```
let rec lemma eval_aux_frame (p q:array real) (x:real) (i j:int)
  requires { forall k. i <= k < j -> p[k] = q[k] }
  variant { j - i }
  ensures { eval_aux p x i j = eval_aux q x i j }
= if j > i then eval_frame p q x (i+1) j
```

Property needed very often, e.g. for addition of polynomials

Frame properties in general

For a predicate P , the *frame* of P is the set of memory locations $fr(P)$ that P depends on.

Frame property

P is invariant under mutations outside $fr(P)$

$$\frac{H \vdash P \quad H \cap fr(P) = H' \cap fr(P)}{H' \vdash P}$$

See also [\[Kassios, 2006\]](#)

Outline

Call by Reference

The Framing Issue

Pointer Programs

Pointer programs

- ▶ We drop the hypothesis “no reference to reference”
- ▶ Allows to program on *linked data structures*. Example (in the C language):

```
struct List { int data; linked_list next; }  
*linked_list;  
while (p <> NULL) { p->data++; p = p->next }
```

- ▶ “In-place” assignment
- ▶ References are now *values* of the language: “pointers” or “memory addresses”

We need to handle aliasing problems differently

Syntax

- ▶ For simplicity, we assume a language with pointers to records
- ▶ Access to record field: $e.f$
- ▶ Update of a record field: $e.f \leftarrow e'$

Operational Semantics

- ▶ New kind of values: *loc* = the type of pointers
- ▶ A special value *null* of type *loc* is given
- ▶ A program state is now a pair of
 - ▶ a *store* which maps variables identifiers to values
 - ▶ a *heap* which maps pairs (loc, field name) to values
- ▶ Memory access and updates should be proved safe (no “null pointer dereferencing”)
- ▶ For the moment we forbid allocation/deallocation
[See lecture next next week]

Component-as-array trick

[Bornat, 2000]

If

- ▶ a program is *well-typed*
- ▶ The set of *all field names are known*

then the heap can be also seen as *a finite collection of maps*, one for each field name:

- ▶ map for a field of type τ maps loc to values of type τ

This “trick” allows to *encode pointer programs* into our previous programming language:

- ▶ Use maps indexed by locs (instead of integers for arrays)

Component-as-array model

```
type loc
constant null : loc

val acc(ref field: loc -> 'a, l:loc) : 'a
  requires l <> null
  reads field
  ensures result = select(field,l)

val upd(ref field: loc -> 'a, l:loc, v:'a):unit
  requires l <> null
  writes field
  ensures field = store(field@old,l,v)
```

Encoding:

- ▶ Access to record field: $e.f$ becomes $\text{acc}(f,e)$
- ▶ Update of a record field:
 $e.f \leftarrow e'$ becomes $\text{upd}(f,e,e')$

Example

▶ In C

```
struct List { int data; linked_list next; }
*linked_list;

while (p <> NULL) { p->data++; p = p->next }
```

▶ Encoded as

```
val ref data: loc -> int
val ref next: loc -> loc
val ref p : loc

while p <> null do
  upd(data,p,acc(data,p)+1);
  p <- acc(next,p)
```

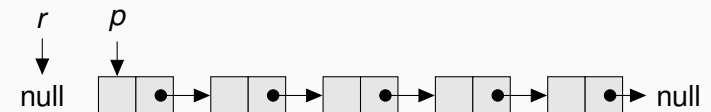
In-place List Reversal

A la C/Java:

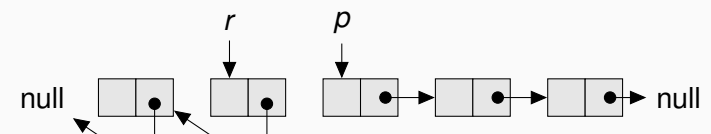
```
linked_list reverse(linked_list l) {
  linked_list p = l;
  linked_list r = null;
  while (p != null) {
    linked_list n = p->next;
    p->next = r;
    r = p;
    p = n
  }
  return r;
}
```

In-place List Reversal

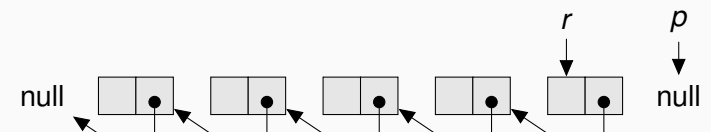
initial step:



intermediate step:



final state:



In-place Reversal in our Model

```

let reverse (l:loc) : loc =
  let ref p = l in
  let ref r = null in
  while p <> null do
    let n = acc(next,p) in
    upd(next,p,r);
    r <- p;
    p <- n
  done;
  r

```

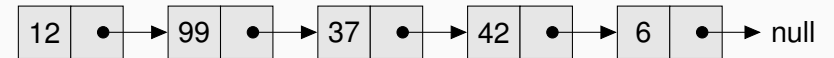
Goals:

- Specify the expected behavior of `reverse`
- Prove the implementation

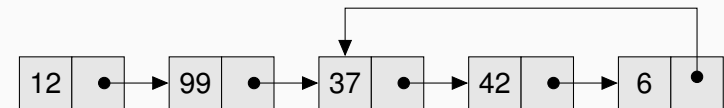
Specifying reverse

Three possibilities for a shape of a linked list:

- null terminated, e.g.:



- cyclic, e.g.:



- or... infinite ! (not forbidden in our model)

Specifying the function

Predicate `list_seg(p, next, pM, q)` :

p points to a list of nodes *p_M* that ends at *q*

$$p = p_0 \xrightarrow{\text{next}} p_1 \cdots \xrightarrow{\text{next}} p_k \xrightarrow{\text{next}} q$$

$$p_M = \text{Cons}(p_0, \text{Cons}(p_1, \dots \text{Cons}(p_k, \text{Nil}) \dots))$$

p_M is the *model list* of *p*

```

predicate list_seg (p:loc, next: loc -> loc,
                    pM:list loc, q:loc) =
  match pM with
  | Nil -> p = q
  | Cons h t ->
    p <> null /\ h=p /\ list_seg((next p),next,t,q)

```

Specification

- pre: input *l* well-formed:

$$\exists l_M. \text{list_seg}(l, \text{next}, l_M, \text{null})$$

- post: output well-formed:

$$\exists r_M. \text{list_seg}(\text{result}, \text{next}, r_M, \text{null})$$

and

$$r_M = \text{rev}(l_M)$$

Issue: quantification on *l_M* is global to the function

- Use *ghost* variables

Annotated In-place Reversal

```

let reverse (l:loc) (ghost lM:list loc) : loc =
  requires list_seg(l,next,lM,null)
  writes next
  ensures list_seg(result,next,rev(lM),null)
  body
    let ref p = l in
    let ref r = null in
    while p <> null do
      let n = acc(next,p) in
      upd(next,p,r);
      r <- p;
      p <- n
    done;
    r

```

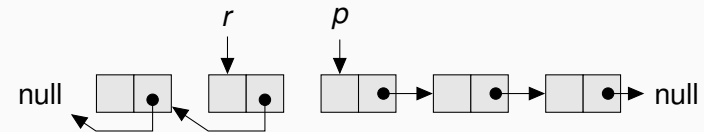
See file [linked_list_rev.mlw](#)

In-place Reversal: loop invariant

```

while (p <> null) do
  let n = acc(next,p) in
  upd(next,p,r);
  r <- p;
  p <- n

```



Local ghost variables p_M, r_M

$\text{list_seg}(p, \text{next}, p_M, \text{null})$

$\text{list_seg}(r, \text{next}, r_M, \text{null})$

$\text{append}(\text{rev}(p_M), r_M) = \text{rev}(l_M)$

Needed lemmas

To prove invariant $\text{list_seg}(p, \text{next}, p_M, \text{null})$, we need to show that list_seg remains true when next is updated:

```

lemma list_seg_frame: forall next1 next2:map loc loc,
  p q r v: loc, pM:list loc.
  list_seg(p,next1,pM,q) /\
  next2 = store(next1,r,v) /\
  not mem(r,pM) -> list_seg(p,next2,pM,q)

```

This is again an instance of the general *frame property*

Needed lemmas

- ▶ To prove invariant $\text{list_seg}(p, \text{next}, p_M, \text{null})$, we need to show that list_seg remains true when next is updated:
- ▶ But to apply the frame lemma, we need to show that a path going to null cannot contain repeated elements

```

lemma list_seg_no_repet:
  forall next:map loc loc, p: loc, pM:list loc.
  list_seg(p,next,pM,null) -> no_repet(pM)

```

Needed lemmas

- ▶ To prove invariant `list_seg(r, next, r_M, null)`, we need the frame property
- ▶ Again, to apply the frame lemma, we need to show that p_M, r_M remain *disjoint*: it is an additional invariant

Exercise

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?

See [linked_list_app.mlw](#)

Bibliography

Aliasing control using static typing

[Filliâtre, 2016] J.-C. Filliâtre, L. Gondelman, A. Paskevich. A Pragmatic Type System for Deductive Verification, 2016. (see also Gondelman's PhD thesis)

Component-as-array modeling

[Bornat, 2000] Richard Bornat, Proving Pointer Programs in Hoare Logic, *Mathematics of Program Construction*, 102–126, 2000

[Kassios, 2006] I. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions, *International Symposium on Formal Methods*.

Advertising next next lectures

- ▶ Reasoning on pointer programs using the component-as-array trick is complex
 - ▶ need to state and prove *frame* lemmas
 - ▶ need to specify many *disjointness* properties
 - ▶ even harder is the handling of *memory allocation*
- ▶ *Separation Logic* is another approach to reason on heap memory
 - ▶ memory resources *explicit* in formulas
 - ▶ frame lemmas and disjointness properties are internalized