

Aliasing Issues: Call by reference, Pointer programs

Claude Marché

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

18 janvier 2017

Reminder of the last lecture

- ▶ Additional features of the specification language
 - ▶ Abstract Types: e.g. sets, *maps*
 - ▶ Product Types: *records* and such
 - ▶ Sum Types, e.g. *lists*
- ▶ Programs on *lists*
- ▶ Computer Arithmetic: *bounded integers*, *floating-point numbers*
- ▶ Additional feature of the programming language
 - ▶ *Exceptions*
 - ▶ Function contracts extended with *exceptional post-conditions*

Home Work from previous lecture

- ▶ Re-implement and prove linear search in an array, using an exception to exit immediately when an element is found.
(see [lin_search_exc.mlw](#))

- ▶ Implement and prove binary search using also a immediate exit:

$low = 0$; $high = n - 1$;

while $low \leq high$:

 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 [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, sets)

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

Need for call by reference

Example: stacks of integers

```
type stack = list int
val s:ref stack

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

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

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

Call by Reference: example

```
val s1,s2: ref stack

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

let fun wrong(s5: ref 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 fun  $f(y_1 : \text{ref } \tau_1, \dots, y_k : \text{ref } \tau_k, x_1 : \tau'_1, \dots, x_n : \tau'_n)$ :
  ...
```

- ▶ Call:

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

where each z_j must be a reference

Operational Semantics

Intuitive semantics, by substitution:

$$\frac{\Pi' = \{x_i \leftarrow \llbracket t_i \rrbracket_{\Sigma, \Pi}\} \quad \Sigma, \Pi' \models \text{Pre} \quad \text{Body}' = \text{Body}[y_j \leftarrow z_j]}{\Sigma, \Pi, f(z_1, \dots, z_k, t_1, \dots, t_n) \rightsquigarrow \Sigma, \Pi, (\text{Old} : \text{frame}(\Pi', \text{Body}', \text{Post}))}$$

- ▶ 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:

$$\frac{\Sigma' = \Sigma[y_j \leftarrow \Sigma(z_j)] \quad \Pi' = \{x_i \leftarrow \llbracket t_i \rrbracket_{\Sigma, \Pi}\} \quad \Sigma, \Pi' \models Pre}{\Sigma, \Pi, f(z_1, \dots, z_k, t_1, \dots, t_n) \rightsquigarrow \Sigma', \Pi, (Old : \text{frame}(\Pi', Body, Post))}$$

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

Warning: not the same semantics !

Difference in the semantics

```
val g : ref int

let fun f(x:ref int):unit
  body x := 1; x := g+1

let fun 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 fun f(x:ref int, y:ref int):
  writes x, y
  ensures x = 1 ∧ y = 2
  body x := 1; y := 2

val g : ref int

let fun test():
  body
    f(g,g);
    assert g = 1 ∧ g = 2 (* ????)
```

- ▶ Aliasing of reference parameters

Aliasing Issues (2)

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

let fun p(x:ref int):
  writes g1, x
  ensures g1 = 1 ∧ x = 2
  body g1 := 1; x := 2

let fun test():
  body
    p(g2); assert g1 = 1 ∧ g2 = 2; (* OK *)
    p(g1); assert g1 = 1 ∧ g1 = 2; (* ??? *)
```

- ▶ Aliasing of a global variable and reference parameter

Aliasing Issues (3)

```
val g : ref int

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

let fun 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 g : ref int

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

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

- ▶ See file [stack2.mlw](#)

Typing: Alias-Freedom Conditions

For a function of the form

```
f(y1 : ref τ1, ..., yk : ref τk, ...) : τ:
  writes  $\vec{w}$ 
  reads  $\vec{r}$ 
```

Typing rule for a call to f :

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

- ▶ effective arguments z_j must be distinct
- ▶ effective arguments z_j must not be 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 fun 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

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: $e \rightarrow f$
- ▶ Update of a record field: $e \rightarrow f := e'$

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 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(field: ref (map loc  $\alpha$ ), l:loc) :  $\alpha$ 
  requires l  $\neq$  null
  reads field
  ensures result = select(field, l)

val upd(field: ref (map loc  $\alpha$ ), l:loc, v: $\alpha$ ):unit
  requires l  $\neq$  null
  writes field
  ensures field = store(field@Old, l, v)
```

Encoding:

- ▶ Access to record field: $e \rightarrow f$ becomes `acc(f, e)`
- ▶ Update of a record field:
 $e \rightarrow f := e'$ becomes `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 data: ref (map loc int)
val next: ref (map loc loc)

while p  $\neq$  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 Reversal in our Model

```
let fun reverse (l:loc) : loc =
  let p = ref l in
  let r = ref 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 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:map loc loc,
                  pM:list loc, q:loc) =
  match pM with
  | Nil → p = q
  | Cons h t →
    p ≠ null ∧ h=p ∧ list_seg(select(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 fun 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 p = ref l in
  let r = ref 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

$list_seg(p, next, p_M, null)$

$list_seg(r, next, r_M, null)$

$append(rev(p_M), r_M) = rev(l_M)$

Needed lemmas

To prove invariant $list_seg(p, next, p_M, null)$, 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 an instance of a general *frame property*

Frame property

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

Needed lemmas

- ▶ To prove invariant `list_seg(p, next, pM, 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, rM, null)`, we need the frame property
- ▶ Again, to apply the frame lemma, we need to show that `pM, rM` 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 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

Schedule

- ▶ Lecture on January 25th by Arthur Charguéraud
- ▶ February 1st: lab session, help with the project, same room as usual, bring your laptop
- ▶ Lectures on February 8th, 15th, 22th
- ▶ Monday February 27th, deadline for sending your project solution
- ▶ Written exam: March 1st, 16:00, room 2036