Aliasing: Call by Reference, Pointer Programs

Claude Marché

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

20 février 2013

Reminder of former lectures

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*

Today's lecture

Main topic: *Aliasing*

Two sub-topics:

- Call by reference
- Pointer programs

Outline

Need for call by reference

Example: stacks of integers

```
type stack = list int

val s:ref stack

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

function 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

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

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

Call by Reference: example

```
val s1,s2: ref stack

function test():
  ensures result = 13 ∧ head(s2) = 42
  body push(s1,13); push(s2,42); pop(s1)
```

- See file `stack1.mlw`

A note about program modules

Call by reference allows to structure programs into *modules*:

- Encapsulate types, variables and functions
- A program *importing* a module sees
  - the types
  - the *contracts* of the functions
  - the declarations of global variables

Modules: Why3 syntax

```
module Stack
  use import list.List
  type stack = list int

  val push (s:ref stack) (x:int): unit
    writes { s }
    ensures { !s = Cons x (old !s) }
  ...
end

module Test
  use import Stack
  ...
```

- See file `stack2.mlw`
- See Why3 Manual for more on modules (**use**, **import**, **export**, **theory**)

Aliasing problems

```
function test(s1,s2: ref stack) : unit
  ensures { head(s1) = 42 ∧ head(s2) = 13 }
  body push(s1,42); push(s2,13)

function wrong(s: ref stack) : int
  ensures { head(s) = 42 ∧ head(s) = 13 }
              something's wrong !?
  body test(s,s)
```

Aliasing is a major issue

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

Outline

Syntax

- Declaration of functions: (references first for simplicity)

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

- Call:

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

  where each z_i must be a reference

Operational Semantics

Intuitive semantics, by substitution:

$$\frac{\Pi' = \{x_i \leftarrow [\![t_i]\!]_{\Sigma,\Pi}\} \quad \Sigma, \Pi' \models \mathit{Pre} \quad \mathit{Body}' = \mathit{Body}[y_j \leftarrow z_j]}{\Sigma, \Pi, f(z_1, \ldots, z_k, t_1, \ldots, t_n) \rightsquigarrow \Sigma, \Pi, (\mathit{Old} : \texttt{frame}\ \Pi', \mathit{Body}', \mathit{Post})}$$

- 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:

$$\frac{\Sigma' = \Sigma[y_j \leftarrow \Sigma(z_j)] \quad \Pi' = \{x_i \leftarrow [\![t_i]\!]_{\Sigma,\Pi}\} \quad \Sigma, \Pi' \models \mathit{Pre}}{\Sigma, \Pi, f(z_1, \ldots, z_k, t_1, \ldots, t_n) \rightsquigarrow \Sigma', \Pi, (\mathit{Old} : \mathtt{frame}\ \Pi', \mathit{Body}, \mathit{Post})}$$

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

Operational Semantics

Variant: Semantics by copy/restore:

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

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

Warning: not the same semantics !

Difference in the semantics

```
val g : ref int

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

function test():unit
  body g:=0; f(g)
```

After executing test:

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

Aliasing Issues (1)

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

val g : ref int

function 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

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

function 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

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

function 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

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

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

Typing: Alias-Freedom Conditions

For a function of the form

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

Typing rule for a call to f:

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

- 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

Outline

New references

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

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

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

See file `stack3.mlw`

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; list next; } *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 assumed a language with pointers to records
- Access to record field: `e→f`
- Update of a record field: `e→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

Component-as-array trick

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 Why3 programs

- Use maps indexed by locs instead of integers

Component-as-array model

```
type loc
constant null : loc

function acc(field: ref (map loc α),l:loc) : α
  requires l ≠ null
  reads field
  ensures result = select(field,l)

function upd(field: ref (map loc α),l:loc,v:α):unit
  requires l ≠ null
  writes field
  ensures field = store(field@Old,l,v)
```

Encoding:

- Access to record field: `e→f` becomes `acc(f,e)`
- Update of a record field:
  `e→f := e'` becomes `upd(f,e,e')`

Example

- In C

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

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

- In Why3

```
val data: ref (map loc int)
val next: ref (map loc loc)

while p ≠ null do
   upd(data,p,acc(data,p)+1);
   p := acc(next,p)
```

In-place List Reversal

A la C/Java:

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

In-place Reversal in our Model

```
function reverse (l:loc) : loc =
   let p = ref l in
   let r = ref null in
   while (p ≠ null) do
     let n = acc(next,p) in
     store(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`$(p, next, p_M, q)$: p points to a list of nodes p_M that ends at q.

$$p = p_0 \overset{next}{\rightarrow} p_1 \cdots \overset{next}{\rightarrow} p_k \overset{next}{\rightarrow} \mathrm{q}$$

$$p_M = Cons(p_0, Cons(p_1, \cdots Cons(p_k, Nil) \cdots))$$

p_M is the *model list* of p

```
inductive list_seg(loc,map loc loc,list loc,loc) =
| list_seg_nil:
  forall p:loc, next:map loc loc. list_seg(p,next,Nil,p)
| list_seg_cons:
  forall p q:loc, next:map loc loc, pM:list loc.
    p ≠ null ∧ list_seg(select(next,p),next,pM,q) →
      list_seg(p,next,Cons(p,pM),q)
```

Specification

- pre: input l well-formed:

$$\exists l_M.\texttt{list\_seg}(l, next, l_M, null)$$

- post: output well-formed:

$$\exists r_M.\texttt{list\_seg}(result, next, r_M, null)$$

  and

$$r_M = rev(l_M)$$

Issue: quantification on l_M is global to the function

- Use *ghost* variables

Annotated In-place Reversal

```
function reverse (l:loc) (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
     store(next,p,r);
     r := p;
     p := n
  done;
  r
```

In-place Reversal: loop invariant

```
while (p ≠ null) do
  let n = acc(next,p) in
  store(next,p,r);
  r := p;
  p := n
```

In-place Reversal: loop invariant

```
while (p ≠ null) do
  let n = acc(next,p) in
  store(next,p,r);
  r := p;
  p := n
```

Local ghost variables p_M, r_M

$$\texttt{list\_seg}(p, next, p_M, null)$$

$$\texttt{list\_seg}(r, next, r_M, null)$$

In-place Reversal: loop invariant

```
while (p ≠ null) do
  let n = acc(next,p) in
  store(next,p,r);
  r := p;
  p := n
```

Local ghost variables p_M, r_M

$$\texttt{list\_seg}(p, next, p_M, null)$$

$$\texttt{list\_seg}(r, next, r_M, null)$$

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

See file `linked_list_rev.mlw`

Needed lemmas

- Need to show that `list_seg` remains true when `next` is updated:

```
lemma list_seg_frame: forall next1 next2:map loc loc,
  p q v: loc, pM:list loc.
  list_seg(p,next1,pM,null) ∧
  next2 = store(next1,q,v) ∧
  ¬ mem(q,pM) → list_seg(p,next2,pM,null)
```

- For that, need to show that p_M, r_M are *disjoint*
- For that, need to show that a rep list do not contain repeted 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)
```

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;
  p := l1;
  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?