Aliasing: Call by Reference, Pointer Programs

Claude Marché

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

20 février 2013
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

Call by Reference
  Syntax, Semantics, Typing
  About Creation of References

Pointer Programs
Need for call by reference

Example: stacks of integers

```plaintext
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

```plaintext
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

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

function wrong(s: ref stack) : int
    ensures { head(s) = 42 \land head(s) = 13 }
    body test(s, s)

Aliasing is a major issue

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

Call by Reference
  Syntax, Semantics, Typing
    About Creation of References

Pointer Programs
Syntax

- Declaration of functions: (references first for simplicity)

  function \( f(\mathit{y}_1 : \text{ref } \tau_1, \ldots, \mathit{y}_k : \text{ref } \tau_k, \mathit{x}_1 : \tau'_1, \ldots, \mathit{x}_n : \tau'_n) : \cdots \)

- Call:

  \( f(\mathit{z}_1, \ldots, \mathit{z}_k, \mathit{e}_1, \ldots, \mathit{e}_n) \)

  where each \( \mathit{z}_i \) must be a reference
Intuitive semantics, by substitution:

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

- 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:

\[
\Sigma' = \Sigma[y_j \leftarrow \Sigma(z_j)] \quad \Pi' = \{x_i \leftarrow [t_i]_{\Sigma, \Pi}\} \quad \Sigma, \Pi' \models Pre
\]

\[
\Sigma, \Pi, f(z_1, \ldots, z_k, t_1, \ldots, t_n) \leadsto \Sigma', \Pi, (\text{Old : frame } \Pi', \text{Body}, \text{Post})
\]

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

\[
\Sigma, \Pi, (\text{frame } \Pi', v, P) \leadsto \Sigma', \Pi, v
\]
Operational Semantics

Variant: Semantics by copy/restore:

\[ \Sigma' = \Sigma[y_j \leftarrow \Sigma(z_j)] \quad \Pi' = \{x_i \leftarrow [t_i]_{\Sigma, \Pi}\} \quad \Sigma, \Pi' \models Pre \]

\[ \Sigma, \Pi, f(z_1, \ldots, z_k, t_1, \ldots, t_n) \leadsto \Sigma', \Pi, (\text{Old : frame } \Pi', \text{Body}, \text{Post}) \]

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

\[ \Sigma, \Pi, (\text{frame } \Pi', v, P) \leadsto \Sigma', \Pi, v \]

Warning: not the same semantics!
Difference in the semantics

```ml
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
function \( f(x: \text{ref int}, y: \text{ref int}):\)
\begin{align*}
\text{writes} & \quad x \quad y \\
\text{ensures} & \quad x = 1 \land y = 2 \\
\text{body} & \quad x := 1; \quad y := 2
\end{align*}

val \( g : \text{ref int} \)

function \( \text{test}():: \)
\begin{align*}
\text{body} \\
& \quad f(g,g); \\
\text{assert} \quad g = 1 \land g = 2 \quad (* \quad ???? \quad *)
\end{align*}

- Aliasing of reference parameters
Aliasing Issues (2)

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

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

function test():
  body
  p(g2); assert g1 = 1 \land g2 = 2; (* OK *)
  p(g1); assert g1 = 1 \land g1 = 2; (* ??? *)
```

- Aliasing of a global variable and reference parameter
Aliasing Issues (3)

```plaintext
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
New need in specifications
Need to *specify read references in contracts*

```ocaml
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

\[ f(y_1 : \tau_1, \ldots, y_k : \tau_k, \ldots) : \tau : \]

writes \( \vec{w} \)

reads \( \vec{r} \)

Typing rule for a call to \( f \):

\[ \ldots \quad \forall i, j, i \neq j \rightarrow z_i \neq z_j \quad \forall i, j, z_i \neq w_j \quad \forall i, j, z_i \neq r_j \]

\[ \ldots \vdash f(z_1, \ldots, z_k, \ldots) : \tau \]

- 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

Call by Reference
  Syntax, Semantics, Typing
  About Creation of References

Pointer Programs
New references

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

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

- Typing should require that a returned reference is always fresh

See file stack3.mlw
Outline

Call by Reference
  Syntax, Semantics, Typing
  About Creation of References

Pointer Programs
Pointer programs

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

```c
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
For simplicity, we assumed 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: $\textit{loc} =$ the type of pointers
- A special value $\textit{null}$ of type loc is given
- A program state is now a pair of
  - a $\textit{store}$ which maps variables identifiers to values
  - a $\textit{heap}$ 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 $\tau$ maps loc to values of type $\tau$

This “trick” allows to encode pointer programs into Why3 programs
- Use maps indexed by locs instead of integers
Component-as-array model

```plaintext

```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: \( e \to f \) becomes \( \text{acc}(f,e) \)
- Update of a record field:
  \( e \to f := e' \) becomes \( \text{upd}(f,e,e') \)
Example

- In C

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

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

- In Why3

```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:

```c
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 \( \text{list\_seg}(p, next, p_M, q) \): \( p \) points to a list of nodes \( p_M \) that ends at \( q \).

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

\[
p_M = \text{Cons}(p_0, \text{Cons}(p_1, \cdots \text{Cons}(p_k, \text{Nil}) \cdots ))
\]

\( p_M \) is the model list of \( p \)

```ocaml
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 \neq \text{null} \land \text{list\_seg}(
      \text{select}(next,p),next,pM,q)
    \rightarrow \text{list\_seg}(p,next,\text{Cons}(p,pM),q)
```

**Specification**

- **pre**: input $l$ well-formed:
  \[ \exists l_M. \text{list}_\text{seg}(l, next, l_M, null) \]

- **post**: output well-formed:
  \[ \exists r_M. \text{list}_\text{seg}(\text{result}, next, r_M, null) \]

  and

  \[ r_M = \text{rev}(l_M) \]

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

- Use *ghost* variables
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 $\neq$ null) do
    let n = acc(next,p) in
    store(next,p,r);
    r := p;
    p := n
  done;
  r
In-place Reversal: loop invariant

\[
\textbf{while } (p \neq \text{null}) \textbf{ do} \\
\quad \textbf{let } n = \text{acc}(\text{next}, p) \textbf{ in} \\
\quad \text{store}(\text{next}, p, r); \\
\quad r := p; \\
\quad p := n
\]
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 \( p_M, r_M \)

list_seg(p, next, p_M, null)

list_seg(r, next, r_M, null)

See file linked_list_rev.mlw
In-place Reversal: loop invariant

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

Local ghost variables $p_M, r_M$

```
list_seg(p, next, p_M, null)
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:

  ```latex
  \textbf{lemma} \ list\_seg\_frame: \textbf{forall} \ next1 \ next2: map \ loc \ loc, \ p \ q \ v: \ loc, \ pM: list \ loc.
  \neg \ \text{mem}(q, pM) \rightarrow \ 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

  ```latex
  \textbf{lemma} \ list\_seg\_no\_repet:
  \textbf{forall} \ next: map \ loc \ loc, \ p: \ loc, \ pM: list \ loc.
  \neg \ \text{mem}(q, pM) \rightarrow \ no\_repet(pM)
  ```
Exercise

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

```plaintext
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?