Aliasing: Call by Reference, Pointer Programs

Claude Marché

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

23 janvier 2012
Reminder of the last lecture

Modeling compound data structures using expressive specification languages

- Defined functions and predicates
- Product types (records)
- Sum types (lists, trees)
- Axiomatizations (arrays)

Important points:

- *pure* types, no internal “in-place” assignment
- Mutable variables = references to pure types
Exercice: Linear Search

val a: ref(map real)
val idx : ref int

procedure search (n:int, v:real):
  requires 0 ≤ n
  writes idx
  ensures ?
  body ?

1. Formalize postcondition: if \( v \) occurs in \( a \), between 0 and \( n − 1 \) then \( idx \) is an index where \( v \) occurs, otherwise \( idx \) set to \(-1\)

2. Implement and prove linear search:
   for each \( i \) from 0 to \( n − 1 \): if \( a[i] = v \) then \( idx := i; \) exit
Exercice: Selection Sort

val a : ref(map real)

procedure sort(n:int):
  requires 0 ≤ n
  writes a
  ensures ?
  body ?

1. Formalize Postconditions:
   - array in increasing order between 0 and n – 1
   - array at exit is a permutation of the array at entrance

2. Implement and prove selection sort algorithm:
   for each i from 0 to n – 1:
     finds index idx of the min element between i and n – 1
     swap elements at indexes i and idx
Today’s lecture

Main topic: *Aliasing*

- Call by reference
- Pointer programs
Outline

Call by Reference

- Call by Reference and Modules
- Syntax, Semantics, Proof Rules
- Expressions with Side Effects
- Creation of References

Pointer Programs
Need for call by reference

Example: stacks of integers

```plaintext
type stack = list int

val s:ref stack

push(x:int):
  writes s
  ensures s = Cons(x,s@Old)

body ...
```

If we need two stacks in the same program:

- We don’t want to write the procedure twice
- We want a module for stacks
Call by Reference

Call by Reference and Modules
Syntax, Semantics, Proof Rules
Expressions with Side Effects
Creation of References

Pointer Programs
Call by Reference

```plaintext
(type stack = list int

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

val s1,s2: ref stack

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

Allows to program *modules*:

- Encapsulate types, variables and procedures
- A program *importing* a module sees
  - the types
  - the *contracts* of the procedures
  - the declarations of global variables
```ml
module Stack
  use import list.List
  type stack = list int
  val push (s: ref stack) (x:int):
    { true }
    unit writes s
    { !s = Cons x (old !s) } 
end

module Test
  use import module Stack
...
```

- See file `stack1.mlw`
- See Why3 Manual for more on modules (`use`, `import`, `export`, `theory`)
Outline

Call by Reference
  Call by Reference and Modules
  Syntax, Semantics, Proof Rules
  Expressions with Side Effects
  Creation of References

Pointer Programs
Syntax

- Declaration of procedures: (references first for simplicity)

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

- Call:

  \[ p(z_1, \ldots, z_k, e_1, \ldots, e_n) \]

  where each \( z_i \) must be a reference
Operational Semantics

Intuitive semantics, by substitution:

\[
\Gamma' = \{ x_i \leftarrow \llbracket e_i \rrbracket_{\Sigma, \Gamma} \} \quad [\text{pre}]_{\Sigma, \Gamma'} \text{ holds} \quad \text{Body}' = \text{Body}[y_j \leftarrow z_j]
\]

\[
\Sigma, \Gamma, p(z_1, \ldots, z_k, e_1, \ldots, e_n) \leadsto \Sigma, \Gamma', (\text{old} : \text{Body}'; \text{return}(\text{post}, \Gamma))
\]

- The body is executed, where each occurrence of reference parameters are replaced by the corresponding reference argument.
- Not a “practical” semantics…
Operational Semantics

Semantics by copy/restore:

\[ \Sigma' = \Sigma[y_j \leftarrow \Sigma(z_j)] \quad \Gamma' = \{ x_i \leftarrow \llbracket e_i \rrbracket_{\Sigma, \Gamma} \} \quad \llbracket \text{pre} \rrbracket_{\Sigma, \Gamma'} \text{ holds} \]

\[ \Sigma, \Gamma, p(z_1, \ldots, z_k, e_1, \ldots, e_n) \leadsto \Sigma, \Gamma', (\text{Old : Body}; \text{return}(\text{post}, \Gamma)) \]

\[ \llbracket \text{post} \rrbracket_{\Sigma, \Gamma'} \text{ holds} \quad \Sigma' = \Sigma[z_j \leftarrow \Sigma(y_j)] \]

\[ \Sigma, \Gamma', \text{return}(\text{post}, \Gamma) \leadsto \Sigma', \Gamma, \text{skip} \]
Operational Semantics

Semantics by copy/restore:

\[ \Sigma' = \Sigma[y_j \leftarrow \Sigma(z_j)] \quad \Gamma' = \{ x_i \leftarrow [e_i]_{\Sigma, \Gamma} \} \quad \llbracket \text{pre} \rrbracket_{\Sigma, \Gamma'} \text{ holds} \]
\[ \Sigma, \Gamma, p(z_1, \ldots, z_k, e_1, \ldots, e_n) \rightsquigarrow \Sigma, \Gamma', (\text{Old} : \text{Body}; \text{return} (\text{post}, \Gamma)) \]

\[ \llbracket \text{post} \rrbracket_{\Sigma, \Gamma'} \text{ holds} \quad \Sigma' = \Sigma[z_j \leftarrow \Sigma(y_j)] \]
\[ \Sigma, \Gamma', \text{return} (\text{post}, \Gamma) \rightsquigarrow \Sigma', \Gamma, \text{skip} \]

Warning: not the same semantics!
### Aliasing Issues

```plaintext
procedure p(x:ref int, y:ref int):
    writes x y
    ensures x = 1 ∧ y = 2
    body x := 1; y := 2

val g : ref int

procedure test():
    body
    p(g,g);
    assert g = 1 ∧ g = 2 (∗ ❀ ❀ ❀ ❀ ❀ ❀)
```

- **Aliasing** of reference parameters
- makes Hoare rule for assignment wrong
- makes WP rule wrong too
Aliasing Issues

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

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

procedure 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

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

procedure test():
    requires { g = 0 }
body
    p(g);
    assert g = 0; (* g isn’t written by p *)
    assert g = 1; (* instance of post of p: g = 0 + 1 *)
```

- Aliasing of read only and written reference
- Need to specify read references in contracts
Typing: Alias-Freedom Conditions

For a procedure of the form

\[ p(y_1 : \tau_1, \ldots, y_k : \tau_k, \ldots) : \]
  writes \( \vec{w} \)
  reads \( \vec{r} \)

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

\[
\forall ij, i \neq j \rightarrow z_i \neq z_j \quad \forall i, j, z_i \neq w_j \quad \forall i, j, z_i \neq r_j \\
\vdash p(z_1, \ldots, z_k, \ldots) : wf
\]

- effective arguments \( z_j \) must be distinct
- effective arguments \( z_j \) must not be read nor written by \( p \)
Proof Rules

Thanks to restricted typing:

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

Example: prove

```plaintext
requires s = Nil
ensures s = Cons(42,Cons(13,Nil))
body push(s,13); push(s,42)
```
Outline

Call by Reference
Call by Reference and Modules
Syntax, Semantics, Proof Rules
Expressions with Side Effects
Creation of References

Pointer Programs
Functions with Side Effects

- Goal: we would like subprograms that returns a value
- Example: stack continued

```
pop(s: ref stack): int
  requires s ≠ Nil
  writes s
  ensures result = head(s@Old) ∧ s = tail(s@Old)
```

- Keyword `result` denotes the returned value
Expressions with Side Effects

- Subprogram returning a value is used in an *expression*, not a *statement*

  Example:

  ```
  push(s,13); push(s,42);
  let x = pop(s)+pop(s) in
  assert x = ?
  ```

- Need change in syntax
- Need precision in semantics: evaluation order
Syntax

- former pure expressions are now called *terms*
- no difference between expressions and statements anymore
- former statements are now expressions of type *unit*
- the type unit is a new base type in logic, inhabited by only one constant denoted ()
- `skip` is identified with ()
Operational Semantics

- one-step execution has the form

\[
\Sigma, \Pi, e \rightsquigarrow \Sigma', \Pi', e'
\]

- Constants of the logic are *values*, they do not reduce

- Rules, show explicit order of evaluation, e.g.

\[
\Sigma, \Pi, e_1 \rightsquigarrow \Sigma', \Pi', e'_1 \\
\Sigma, \Pi, e_1 + e_2 \rightsquigarrow \Sigma', \Pi', e'_1 + e_2
\]

\[
\Sigma, \Pi, e_2 \rightsquigarrow \Sigma', \Pi', e'_2 \\
\Sigma, \Pi, v_1 + e_2 \rightsquigarrow \Sigma', \Pi', v_1 + e'_2
\]

\[
v = v_1 + v_2 \\
\Sigma, \Pi, v_1 + v_2 \rightsquigarrow \Sigma, \Pi, v
\]
Operational Semantics

- **Assignment**

\[
\Sigma, \Pi, x := v \leadsto \Sigma[x \leftarrow v], \Pi, ()
\]

- **Subprogram call**

\[
\Sigma', \Gamma = \{ x_i \leftarrow [e_i]_{\Sigma, \Gamma} \} \quad \text{if } [pre]_{\Sigma, \Gamma'} \text{ holds}
\]

\[
\Sigma, \Gamma, p(z_1, \ldots, z_k, v_1, \ldots, v_n) \leadsto \\
\Sigma, \Pi', \text{let } \text{result} = \text{Body in return(result, Post, } \Pi) \]

(non reference arguments must be values)

\[
[\text{let } \text{result} = v \text{ in Post}]_{\Sigma, \Gamma'} \text{ holds} \\
\Sigma' = \Sigma[z_j \leftarrow \Sigma(y_j)]
\]

\[
\Sigma, \Pi', \text{return(v, Post, } \Pi) \leadsto \Sigma', \Pi, v
\]
Weakest Preconditions

- Pure terms: \( WP(t, Q) = Q[result \leftarrow t] \)
- Assignment:
  \[ WP(x := e, Q) = WP(e, Q[result \leftarrow (); x \leftarrow result]) \]
- Let binding:
  \[
  WP(\text{let } x = e_1 \text{ in } e_2, Q) = \\
  WP(e_1, WP(e_2, Q)[x \leftarrow result])
  \]
- Conditional
  \[
  WP(\text{if } e_1 \text{ then } e_2 \text{ else } e_3, Q) = \\
  WP(e_1, \text{if } result \text{ then } WP(e_2, Q) \text{ else } WP(e_3, Q))
  \]
Weakest Preconditions

- While loop

\[ WP(\text{while } e_1 \text{ invariant } I \text{ variant } t \text{ do } e_2, Q) = \]
\[ I \land \forall \vec{y}, I \rightarrow WP(e_1, \text{if result then } WP(e_2, I \land t < t@L) \text{ else } Q)[x@L ←] \]

- Subprogram call:

\[ WP(p(t), Q) = \text{Pre}[x ← t] \land \]
\[ \forall \vec{y}, \text{result.}(\text{Post}[x ← t] \rightarrow Q)[t@Old ← t@Here] \]
Example

Prove

```plaintext
push(s, 13);
push(s, 42);
let x = pop(s) in assert x = 42;
let x = pop(s) in assert x = 13
```

See file `stack2.mlw`
Outline

Call by Reference
  Call by Reference and Modules
  Syntax, Semantics, Proof Rules
  Expressions with Side Effects
  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: requires that a reference result is always fresh

See file `stack3.mlw`
Final Note about Modules

▶ In a module, abstract subprogram can be replaced by implementation, even after client programs have been proved.

See file stack4.mlw

▶ About the project: At this point of the course, everything needed for the project has been introduced
Outline

Call by Reference

Call by Reference and Modules
Syntax, Semantics, Proof Rules
Expressions with Side Effects
Creation of References

Pointer Programs
Pointer programs

- We drop the hypothesis “no reference to reference”
- Allows to program on *linked data structures*
- In-place assignment
- Example (C style):

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

- 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: \(\text{loc}\) = the type of pointers
- A special value \(\text{null}\) of type \(\text{loc}\) 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 (\text{loc}, field name) to values
- For the moment we forbid allocation/deallocation
- We also assume all memory access safe
Component-as-array trick

If:
- Programs is well-typed
- 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
Example

- In C

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

- In Why3

```why3
type loc
function null : loc
val data: map loc int
val next: map loc loc
while p <> null do
  data := store(data,p,select(data,p)+1);
  p := select(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;
}
```
function reverse (l:loc) : loc =
    let p = ref l in
    let r = ref null in
    while (p ≠ null) do
        let n = select(next,p) in
        next := store(next,p,r);
        r := p;
        p := n
    done;
    r
Rule of the game:
  ▶ Specify the expected behavior of reverse
  ▶ Prove the implementation
To be continued...