**Home Work last lecture**

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: “for” loops**

**Syntax:** `for i = e1 to e2 do e`

**Typing:**
- $i$ visible only in $e$, and is immutable
- $e_1$ and $e_2$ must be of type `int`, $e$ must be of type `unit`

**Operational semantics:**
(assuming $e_1$ and $e_2$ are values $v_1$ and $v_2$)

\[
\begin{align*}
\text{if } v_1 > v_2 & \text{ then } \\
\Sigma, \Pi, \text{for } i = v_1 \text{ to } v_2 \text{ do } e & \rightsquigarrow \Sigma, \Pi, () \text{ otherwise }
\end{align*}
\]

\[
\begin{align*}
\text{if } v_1 \leq v_2 & \text{ then } \\
\Sigma, \Pi, \text{for } i = v_1 \text{ to } v_2 \text{ do } e & \rightsquigarrow \Sigma, \Pi, (\text{let } i = v_1 \text{ in } e); \text{(for } i = v_1 + 1 \text{ to } v_2 \text{ do } e)
\end{align*}
\]

**Additional exercise:** use a `for` loop in the linear search example `lin_search_for.mlw`
Propose a Hoare logic rule for the for loop:

\[
\{I \land v_1 \leq i \leq v_2\} \text{e}\{i \leftarrow i + 1\} = \{I[i \leftarrow v_1 \land v_1 \leq v_2] \text{for } i = v_1 \text{ to } v_2 \text{ do } \text{e}\{i \leftarrow v_2 + 1\}\}
\]

Propose a rule for computing the WP:

\[\text{WP}(\text{for } i = v_1 \text{ to } v_2 \text{ do } \text{e} , Q) = ?\]

Additional exercise: use a for loop in the linear search example

lin_search_for.mlw

Home Work: “for” loops

Home Work: Binary Search

\[
\text{low} = 0; \text{high} = a.\text{length} - 1;
\]

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 file bin_search.mlw

Binary Search with overflow checking

See bin_search_int32.mlw

Home Work

 Implement and prove binary search using also a immediate exit:

\[
\text{low} = 0; \text{high} = a.\text{length} - 1;
\]

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)
Reminder of the last lecture

- 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*
- Computer Arithmetic: *bounded integers, floating-point numbers*

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*

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
- Pointer Programs
Need for call by reference

Example: stacks of integers

```ml
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)
```

If we need two stacks in the same program:
- We don’t want to write the functions twice!
- We want to write

```ml
type stack = list int

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

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

Call by Reference: example

```ml
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

```ml
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)
```

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

- Declaration of functions: (references first for simplicity)
  ```
  let f(ref y_1 : τ_1, ..., ref y_k : τ_k, x_1 : τ'_1, ..., x_n : τ'_n):
  ...
  ```

- Call:
  ```
  f(z_1, ..., z_k, e_1, ..., e_n)
  ```
  where each $z_i$ must be a (mutable) variable

Operational Semantics

Intuitive semantics, by substitution:

```
Π' = \{ x_i ← [t_i] \}_{i=1}^n \quad Σ, Π' \models Pre
Σ, Π, f(z_1, ..., z_k, t_1, ..., t_n) \leadsto Σ, Π, (Old : frame(Π', 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:

```
Σ' = Σ[y_j ← Σ(z_j)] \quad \Pi' = \{ x_i ← [t_i] \}_{i=1}^n \quad Σ, Π' \models Pre
Σ, Π, f(z_1, ..., z_k, t_1, ..., t_n) \leadsto Σ', Π, (Old : frame(Π', Body, Post))
```

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)

```ocaml
let f(ref x: int, ref y: int):
  writes x, y
  ensures x = 1 \land y = 2
  body x <- 1; y <- 2
val ref g : int
let test():
  body
  f(g,g);
  assert g = 1 \land g = 2 (* ??? *)
```

Aliasing of reference parameters

Aliasing Issues (2)

```ocaml
val ref g1 : int
val ref g2 : int
let p(ref x: int):
  writes g1, x
  ensures g1 = 1 \land x = 2
  body g1 <- 1; x <- 2
let 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)

```
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)

```
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)
```

New need in specifications

Need to specify read references in contracts

See file stack2.mlw
Typing: Alias-Freedom Conditions

For a function of the form

\[ f(\text{ref } y_1 : \tau_1, ..., \text{ref } y_k : \tau_k, ...) : \tau : \]

writes \( \vec{w} \)
reads \( \vec{r} \)

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

\[ \ldots \forall i, j \neq j \rightarrow z_i \neq z_j \quad \forall i, j, z_i \neq w_j \quad \forall i, j, z_i \neq r_j \]
\[ \ldots \vdash f(z_1, \ldots, z_k, ...) : \tau \]

- 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

```plaintext
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

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; 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

For simplicity, we assume a language with pointers to records
Access to record field: e.f
Update of a record field: e.f <- e’

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]

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

```plaintext
val acc(ref field: loc -> α, l:loc) : α
  requires l ≠ null
  reads field
  ensures result = select(field,l)

val upd(ref field: 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
  ```plaintext
  struct List { int data; linked_list next; }
  linked_list;
  while (p <> NULL) { p->data++; p = p->next }
  ```
- Encoded as
  ```plaintext
  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:

```plaintext
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

```plaintext
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 the function

Predicate \( \text{list\_seg}(p, \text{next}, p_M, q) : \)

- \( p \) points to a list of nodes \( p_M \) that ends at \( q \)
  \[ p = p_0 \mapsto p_1 \mapsto \cdots \mapsto p_k \mapsto q \]
  \[ p_M = \text{Cons}(p_0, \text{Cons}(p_1, \ldots \text{Cons}(p_k, \text{Nil}) \ldots)) \]

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

\[
\text{match } p_M \text{ with
  \n  | Nil → p = q
  \n  | Cons h t →
  \n  p \neq \text{null} ∧ h=p ∧ \text{list\_seg}((next p),\text{next},t,q)
\]

Annotated In-place Reversal

```ml
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 \neq \text{null} do
    let n = acc(next,p) in
    upd(next,p,r);
    r <- p;
    p <- n
  done;
  r
```

See file `linked\_list\_rev.mlw`

Specification

- **pre:** input \( l \) well-formed:
  \[ \exists l_M.\text{list\_seg}(l, next, l_M, \text{null}) \]
- **post:** output well-formed:
  \[ \exists r_M.\text{list\_seg}(\text{result}, next, r_M, \text{null}) \]
  and
  \[ r_M = \text{rev}(l_M) \]

Issue: quantification on \( l_M \) is global to the function

- Use **ghost** variables

In-place Reversal: loop invariant

```ml
  while (p \neq \text{null}) do
    let n = acc(next,p) in
    upd(next,p,r);
    r <- p;
    p <- n
  done;
  r
```

Local ghost variables \( p_M, r_M \)

- \( \text{list\_seg}(p, next, p_M, \text{null}) \)
- \( \text{list\_seg}(r, next, r_M, \text{null}) \)
- \( \text{append}(\text{rev}(p_M), r_M) = \text{rev}(l_M) \)
Needed lemmas

To prove invariant \( \text{list} \_\text{seg}(p, next, pM, null) \), we need to show that \( \text{list} \_\text{seg} \) remains true when \( next \) is updated:

\[
\text{lemma list} \_\text{seg} \_\text{frame: } \forall next1 \text{ next2} : \text{map loc loc}, \quad p q r v : \text{loc}, \quad pM : \text{list loc}.
\text{list} \_\text{seg}(p, next1, pM, q) \land
\text{next2} = \text{store}(next1, r, v) \land
\lnot \text{mem}(r, pM) \rightarrow \text{list} \_\text{seg}(p, next2, pM, q)
\]

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.

\[
\begin{align*}
\text{Frame property} \\
P \text{ is invariant under mutations outside } fr(P) \\
H \vdash P \\
H \cap fr(P) = H' \cap fr(P) \\
\hline \\
H' \vdash P
\end{align*}
\]

See also [Kassios, 2006]

Needed lemmas

- To prove invariant \( \text{list} \_\text{seg}(p, next, pM, null) \), we need to show that \( \text{list} \_\text{seg} \) remains true when \( next \) is updated:

\[
\text{lemma list} \_\text{seg} \_\text{no} \_\text{repet: } \forall next : \text{map loc loc}, \quad p : \text{loc}, \quad pM : \text{list loc}.
\text{list} \_\text{seg}(p, next, pM, null) \rightarrow \text{no} \_\text{repet}(pM)
\]

- But to apply the frame lemma, we need to show that a path going to \( null \) cannot contain repeated elements

- 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:

```pseudocode
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


Component-as-array modeling


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

- January 21th: lecture 5 by Jean-Marie Madiot
- January 28th: lecture 6 by Jean-Marie Madiot
- February 4th: lab session, help for solving the project, also this room
- February 11th: lecture 7 by Jean-Marie Madiot
- February 14th: deadline for sending your project solution
- February 18th: lecture 8 by Jean-Marie Madiot
- Written exam: March 3rd, 08:45, also this room