Reminder of the last lecture

- Additional feature of the programming language
  - Exceptions
  - Function contracts extended with exceptional post-conditions
- Additional features of the specification language
  - Product Types: records and such
  - Sum Types, e.g. lists
  - Abstract Types: e.g. sets, maps
- Programs on Arrays
- 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, sets)

Important points:
- pure types, no internal “in-place” assignment
- Mutable variables = references to pure types

Exercise from Previous Lecture

Implement and prove binary search using a immediate exit:

\[
\begin{align*}
  low &= 0; \quad high = n - 1; \\
  \text{while } low \leq high: \\
  &\text{ let } m \text{ be the middle of } low \text{ and } high \\
  &\text{ if } a[m] = v \text{ then return } m \\
  &\text{ if } a[m] < v \text{ then continue search between } m \text{ and } high \\
  &\text{ if } a[m] > v \text{ then continue search between } low \text{ and } m
\end{align*}
\]

See `bin_search.mlw`
Aliasing

Aliasing = two different “names” for the same mutable data

Two sub-topics:
▶ Call by reference
▶ Pointer programs

Outline

Call by Reference

Inductive Predicates

Pointer Programs

Need for call by reference

Example: stacks of integers

```ml
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

```ml
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) = 42 ∧ head(s4) = 13 }
  body push(s3,42); push(s4,13)

let fun wrong(s5 : ref stack) : int
  writes s5
  ensures { head(s5) = 42 ∧ head(s5) = 13 }
  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(y₁ : ref τ₁, ..., yₖ : ref τₖ, x₁ : τ'₁, ..., xₙ : τ'ₙ):
  ...
```

▶ Call:

```
f(z₁, ..., zₖ, e₁, ..., eₙ)
where each zᵢ must be a reference
```

Operational Semantics

Intuitive semantics, by substitution:

```
Π' = Π[tiₗ] yᵢ
Σ, Π' ⊢ Pre  Body' = Body[yᵢ ← zᵢ]
Σ, Π, f(z₁, ..., zₖ, t₁, ..., tₙ) ⊢ 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:

\[
\Sigma' = \Sigma[y_j \leftarrow \Sigma(z_j)] \quad \Pi' = \{x_j \leftarrow [t_j]\}_{\Sigma, \Pi} \quad \Sigma, \Pi' \models \text{Pre}
\]
\[
\Sigma, \Pi, f(z_1, \ldots, z_k, t_1, \ldots, t_n) \Rightarrow \Sigma', \Pi, (\text{Old} : \text{frame}(\Pi', \text{Body}, \text{Post}))
\]

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

Warning: not the same semantics!

Difference in the semantics

```plaintext
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)

```plaintext
let fun f(x:ref int, y:ref int):
  writes x, y
  ensures x = 1 \land y = 2
  body x := 1; y := 2

val g : ref int

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

- Aliasing of reference parameters

Aliasing Issues (2)

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

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

let fun 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)

```ml
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

New need in specifications

Need to *specify read references in contracts*

```ml
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

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

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

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

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

- 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
New references

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

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

- Typing should require that a returned reference is always fresh

Example: Sorting Algorithms

```plaintext
let fun sort (a:array int)
  writes a
  ensures ?
```

How to formalize postconditions:

- array in increasing order between 0 and \textit{a.length} – 1
- array at exit is a permutation of the array at entrance

Outline

- Call by Reference
- Inductive Predicates
- Pointer Programs

Inductive Predicates

- Definition à la Prolog, also in Coq, PVS, etc.
- An \textit{inductive definition} of a predicate has the form

  \[
  \text{inductive } p(x_1, \ldots, x_n):
  \]

  \[
  \mid \text{id}_1: \text{clause}_1
  \]

  ... 

  \[
  \mid \text{id}_k: \text{clause}_k
  \]

  where clauses have the form

  \[
  \forall x. \text{hyp} \Rightarrow p(e_1, \ldots, e_n)
  \]

  and \( p \) occurs only positively in \( \text{hyp} \) (Horn clause).

- Always one smallest fix-point: predicate satisfying the clauses that is true the less often.
Inductive Predicates: Example

Classical example: transitive closure.

**predicate** \( r(x:t, y:t) = \ldots \)

**inductive** \( r\_star(t, t) = \)
  - **refl:** for all \( x \in t \), \( r\_star(x, x) \)
  - **single:** for all \( x, y \in t \), \( r(x, y) \implies r\_star(x, y) \)
  - **trans:** for all \( x, y, z \in t \),
    \( r\_star(x, y) \land r\_star(y, z) \implies r\_star(x, z) \)

Exercise: Selection Sort

```ml
let fun swap (a : array int) (i: int) (j: int) : unit
  writes a ensures ?
let fun index_min (a : array int) (i: int) : int ...
let fun sel_sort (a: array int) ...
```

1. Formalize postconditions:
   - array in increasing order between 0 and \( a.length - 1 \),
   - array at exit is a permutation of the array at entrance.

2. Implement and prove selection sort algorithm:
   - for each \( i \) from 0 to \( a.length - 1 \):
     - find index \( idx \) of the min element between \( i \) and \( a.length - 1 \)
     - swap elements at indexes \( i \) and \( idx \)

See `sel_sort.mlw`

Outline

- Call by Reference
- Inductive Predicates
- 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
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: $\text{loc}$ = the type of pointers
- A special value $\text{null}$ of type $\text{loc}$ is given
- A program state is now a pair of
  - a 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 $\tau$ 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

```
type loc
constant null : loc

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

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

Encoding:
- Access to record field: $e \rightarrow f$ becomes $\text{acc}(f,e)$
- Update of a record field:
  $e \rightarrow f := e' \text{ becomes } \text{upd}(f,e,e')$
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:

```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

```why3
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
    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`.

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

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

`p_M` is the model list of `p`

```why3
inductive list_seg(loc,map loc loc,list loc,list loc) =
| list_seg_nil: |
| list_seg_cons: forall p:loc, next:map loc loc. list_seg(p,next,Nil,p) |
| list_seg_cons: forall p q:loc, next:map loc loc, p_M:list loc.
  p <> null \land \text{list_seg(select(next,p),next,p_M,q)} \rightarrow
  list_seg(p,next,Cons(p,p_M),q)
```
Specification

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

Issue: quantification on `l_M` is global to the function
- Use *ghost* variables

Annotated In-place Reversal

```ml
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 \neq 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 \neq null) do
  let n = acc(next,p) in
  store(next,p,r);
  r := p;
  p := n

Local ghost variables `p_M`, `r_M`

\[ \text{list}_\text{seg}(p, next, p_M, \text{null}) \]
\[ \text{list}_\text{seg}(r, next, r_M, \text{null}) \]
\[ \text{append}(\text{rev}(p_M), r_M) = \text{rev}(l_M) \]

See file `linked_list_rev.mlw`

Needed lemmas

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

  ```ml
  lemma list_seg_frame: forall next1 next2:map loc loc, p q v: loc, p_M:list loc.
  \text{list}_\text{seg}(p,next1,p_M,\text{null}) \land
  next2 = store(next1,q,v) \land
  \neg \text{mem}(q,p_M) \rightarrow \text{list}_\text{seg}(p,next2,p_M,\text{null})
  ```

- For that, we need to show that a path going to `null` cannot contain repeated elements

  ```ml
  lemma list_seg_no_repet: forall next:map loc loc, p: loc, p_M:list loc.
  \text{list}_\text{seg}(p,next,p_M,\text{null}) \implies \\neg \text{mem}(q,p_M)
  ```

- To prove invariant `\text{list}_\text{seg}(r, next, r_M, \text{null})`, we need the frame lemma, again, and for that, we need to show that `p_M`, `r_M` remain *disjoint*: it is an additional invariant
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;
  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`

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:

- Lectures on January 29th, February 5th
- February 12th: Lab session, help for solving the project
- Lectures on February 19th, February 26th
- Written exam on March 12th