Syntax Extensions
(ghost variables, labels, local mutable variables)
Functions and Function calls
Termination
Specification languages, application to arrays

Claude Marché
Cours MPRI 2-36-1 “Preuve de Programme”
17 décembre 2015

Exercise 3

Let’s assume given in the underlying logic the functions div2(x) and mod2(x) which respectively return the division of x by 2 and its remainder. The following program is supposed to compute, in variable r, the power $x^n$.

```plaintext
r := 1; p := x; e := n;
while e > 0 do
    if mod2(e) ≠ 0 then r := r * p;
    p := p * p;
    e := div2(e);
```

- Assuming that the power function exists in the logic, specify appropriate pre- and post-conditions for this program.
- Find an appropriate loop invariant, and prove the program.

Reminder of the last lecture

- Classical Hoare Logic
  - Very simple programming language
  - Deduction rules for triples $\{Pre\}s\{Post\}$
- Modern programming language, ML-like
  - more data types: int, bool, real, unit
  - logic variables: local and immutable
  - statement = expression of type unit
  - Typing rules
  - Formal operational semantics (small steps)
  - type soundness: every typed program executes without blocking.
- Blocking semantics and Weakest Preconditions:
  - $e$ executes safely in $\Sigma, \Pi$ is it does not block on an assertion or a loop invariant
  - If $\Sigma, \Pi \models WP(e, Q)$ then $e$ executes safely in $\Sigma, \Pi$, and if it terminates then $Q$ valid in the final state.

This Lecture’s Goals

- Extend the language:
  - Ghost variables and Labels
  - Local mutable variables
  - Sub-programs, modular reasoning
- Proving Termination
- (First-order) logic as modeling language
  - Automated provers capabilities
  - Towards complex data structures: Axiomatized types and predicates
- Application: Arrays
Outline

Syntax extensions
- Ghost variables and Labels
- Local Mutable Variables

Functions and Functions Calls

Termination, Variants

Advanced Modeling of Programs

Programs on Arrays

Ghost variables

Example: Euclid's algorithm, on two global variables $x, y$

Euclide:

```plaintext
requires ?
ensures ?
= while $y > 0$
do
  let $r \equiv \text{mod} \ x \ y\;;\; x := y;\; y := r$
done;
$x$
```

What should be the post-condition?

Ghost variables

additional variables, introduced for the specification

See Why3 file `euclide_ghost.mlw`

Labels: motivation

- Using ghost variables becomes quickly painful
- Label
  - simple alternative to ghost variables
  - (but not always possible)

Labels: Syntax and Typing

Add in syntax of *terms*:

```
t ::= x@L  \quad \text{(labeled variable access)}
```

Add in syntax of *expressions*:

```
e ::= L:e  \quad \text{(labeled expressions)}
```

Typing:

- only mutable variables can be accessed through a label
- labels must be declared before use

Implicit labels:

- Here, available in every formula
- Old, available in post-conditions
Toy Examples, Continued

```plaintext
{ true }
let v = r in (r := v + 42; v)
{ r = r@Old + 42 \land result = r@old }

{ true }
let tmp = x in x := y; y := tmp
{ x = y@Old \land y = x@Old }
```

SUM revisited:

```plaintext
{ y \geq 0 }
L:
while y > 0 do
  invariant { x + y = x@L + y@L }
  x := x + 1; y := y - 1
{ x = x@Old + y@Old \land y = 0 }
```

Labels: Operational Semantics

Program state
- becomes a collection of maps indexed by labels
- value of variable \( x \) at label \( L \) is denoted \( \Sigma(x, L) \)

New semantics of variables in terms:

```plaintext
\[ [x]_{\Sigma, \Pi} = \Sigma(x, \text{Here}) \]
\[ [x@L]_{\Sigma, \Pi} = \Sigma(x, L) \]
```

The operational semantics of expressions is modified as follows

\[ \Sigma, \Pi, x := \text{val} \leadsto \Sigma\{ (x, \text{Here}) \leftarrow \text{val} \}, \Pi, () \]
\[ \Sigma, \Pi, L : e \leadsto \Sigma\{ (x, \text{Here}) \leftarrow \Sigma(x, \text{Here}) \mid x \text{ any variable} \}, \Pi, e \]

Syntactic sugar: term \( t@L \)
- attach label \( L \) to any variable of \( t \) that does not have an explicit label yet.
- example: \((x + y@K + 2)@L + x\) is \( x@L + y@K + 2 + x@\text{Here} \).

New rules for WP

New rules for computing WP:

```plaintext
WP(x := t, Q) = Q[x@Here \leftarrow t]
WP(L : e, Q) = WP(e, Q)[x@L \leftarrow x@Here \mid x \text{ any variable}]
```

Exercise:

```plaintext
WP(L : x := x + 42, x@Here > x@L) =?
```

Example: Euclide's algorithm revisited

Euclide:

```plaintext
requires { x \geq 0 \land y \geq 0 }
ensures { \text{result} = \gcd(x@Old, y@Old) }
=
L:
while y > 0 do
  invariant { x \geq 0 \land y \geq 0 }
  invariant { \gcd(x, y) = \gcd(x@L, y@L) }
  let r = \text{mod} x y in x := y; y := r
done;
```

See file `euclide_labels.mlw`
Mutable Local Variables

We extend the syntax of expressions with
\[ e ::= \text{let ref } id = e \text{ in } e \]

Example: isqrt revisited

```ocaml
val x, res : ref int

isqrt:
  res := 0;
  let ref sum = 1 in
  while sum \leq x do
    res := res + 1; sum := sum + 2 * res + 1
  done
```

Mutable Local Variables: WP rules

Rules are exactly the same as for global variables

\[ \text{WP(let ref } x = e_1 \text{ in } e_2, Q) = \text{WP}(e_1, \text{WP}(e_2, Q)[x \leftarrow \text{result}]) \]

\[ \text{WP}(x := e, Q) = \text{WP}(e, Q[x \leftarrow \text{result}]) \]

\[ \text{WP}(L : e, Q) = \text{WP}(e, Q)[x@L \leftarrow x@\text{Here} | x \text{ any variable}] \]

Operational Semantics

\[ \Sigma, \Pi, e \leadsto \Sigma', \Pi', e' \]
\( \Pi \) no longer contains just immutable variables.

\[ \Sigma, \Pi, e_1 \leadsto \Sigma', \Pi', e_1' \]

\[ \Sigma, \Pi, \text{let ref } x = e_1 \text{ in } e_2 \leadsto \text{let ref } x = e_1' \text{ in } e_2 \]

\[ \Sigma, \Pi, \text{let ref } x = v \text{ in } e \leadsto \Sigma, \Pi[(x, \text{Here}) \mapsto v], e \]

\( x \text{ local variable} \)

\[ \Sigma, \Pi, x := v \leadsto \Sigma, \Pi[(x, \text{Here}) \mapsto v], e \]

And labels too.

Home Work

- Extend the post-condition of Euclidean algorithm to express the Bezout property:
  \[ \exists a, b, \text{result} = x \ast a + y \ast b \]

- Prove the program by adding appropriate ghost local variables

Use canvas file `exo_bezout.mlw`
Outline

Syntax extensions

Functions and Functions Calls

Termination, Variants

Advanced Modeling of Programs

Programs on Arrays

Functions

Program structure:

\[
\begin{align*}
\text{prog} & ::= \text{decl}^* \\
\text{decl} & ::= \text{vardecl} | \text{fundecl} \\
\text{vardecl} & ::= \text{val id : ref basetype} \\
\text{fundecl} & ::= \text{let fun id ((param,)\text{*}):basetype} \\
& \hspace{1cm} \text{contract body e} \\
\text{param} & ::= \text{id : basetype} \\
\text{contract} & ::= \text{requires } t \text{ writes (id,)\text{*} ensures } t
\end{align*}
\]

Function definition:

\begin{itemize}
  \item Contract:
    \begin{itemize}
      \item pre-condition,
      \item post-condition (label \textit{Old} available),
      \item assigned variables: clause \textit{writes}.
    \end{itemize}
  \item Body: expression.
\end{itemize}

Example: isqrt

```
let fun isqrt(x:int): int
  requires x \geq 0
  ensures result \geq 0 \wedge
  \text{sqr(result)} \leq x < \text{sqr(result + 1)}
body
  let ref res = 0 in
  let ref sum = 1 in
  while sum \leq x do
    res := res + 1;
    sum := sum + 2 * res + 1
  done;
res
```

Example using \textit{Old} label

```
val res: ref int
let fun incr(x:int)
  requires true
  writes res
  ensures res = res@Old + x
body
  res := res + x
```

Typing

Definition \(d\) of function \(f\):

\[
\text{let fun } f(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau \\
\text{ requires } Pre \\
\text{ writes } \vec{w} \\
\text{ ensures } Post \\
\text{ body } Body
\]

Well-formed definitions:

\[
\Gamma' = \{ x_i : \tau_i | 1 \leq i \leq n \} \cdot \Gamma \\
\Gamma' \vdash Pre, Post : \text{formula} \\
\Gamma' \vdash Body : \tau \\
\vec{w}_g \subseteq \vec{w} \text{ for each call } g \\
y \in \vec{w} \text{ for each assign } y
\]

where \(\Gamma\) contains the global declarations. Well-typed function calls:

\[
\Gamma \vdash t_i : \tau_i \\
\Gamma \vdash f(t_1, \ldots, t_n) : \tau
\]

Note: \(t_i\) are immutable expressions.

Operational Semantics

function \(f(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau\)

\[
\text{ requires } Pre \\
\text{ writes } \vec{w} \\
\text{ ensures } Post \\
\text{ body } Body
\]

\[
\Pi' = \{ x_i \mapsto [t_i]_{\Sigma, \Pi} \} \\
\Sigma, \Pi' \models Pre
\]

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

WP Rule of Function Call

let fun \(f(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau\)

\[
\text{ requires } Pre \\
\text{ writes } \vec{w} \\
\text{ ensures } Post \\
\text{ body } Body
\]

\[
\text{WP}(f(t_1, \ldots, t_n), Q) = Pre[x_i \leftarrow t_i] \land \\
\forall \vec{v}, (Post[x_i \leftarrow t_i, w_j \leftarrow v_j, w_j@\text{Old} \leftarrow w_j] \Rightarrow Q[w_j \leftarrow v_j])
\]

Modular proof

When calling function \(f\), only the contract of \(f\) is visible, not its body.
Example: isqrt(42)

Exercise: prove that \(\{true\} \text{isqrt}(42)\{\text{result} = 6\}\) holds.

```plaintext
val isqrt(x:int): int
requires x ≥ 0
writes (nothing)
ensures result ≥ 0 ∧
    \(\text{sqr}(\text{result}) \leq x < \text{sqr}(\text{result} + 1)\)
```

Abstraction of sub-programs

- Keyword `val` introduces a function with a contract but without body
- `writes` clause is mandatory in that case

Example: Incrementation

```plaintext
val res: ref int
val incr(x:int):unit
writes res
ensures res = res@Old + x
```

Exercise: Prove that \(\{res = 6\} \text{incr}(36)\{res = 42\}\) holds.

Soundness Theorem for a Complete Program

Assuming that for each function defined as

```plaintext
let fun f(x_1 : \tau_1, ..., x_n : \tau_n) : \tau
requires Pre
writes \vec{w}
ensures Post
body Body
```

we have

- variables assigned in `Body` belong to \(\vec{w}\),
- \(\models Pre \Rightarrow WP(\text{Body}, Post)[w_i@Old ← w_i]\) holds,

then for any formula \(Q\) and any expression \(e\), if \(\Sigma, \Pi \models WP(e, Q)\) then execution of \(\Sigma, \Pi, e\) is safe

Remark: (mutually) recursive functions are allowed

Outline

- Syntax extensions
- Functions and Functions Calls
- Termination, Variants
- Advanced Modeling of Programs
- Programs on Arrays
Termination

Goal
Prove that a program terminates (on all inputs satisfying the precondition)

Amounts to show that
- loops never execute infinitely many times
- (mutual) recursive calls cannot occur infinitely many times

Case of loops

Solution: annotate loops with loop variants
- a term that decreases at each iteration
- for some well-founded ordering $\prec$ (i.e. there is no infinite sequence $\mathtt{val}_1 \prec \mathtt{val}_2 \prec \mathtt{val}_3 \prec \cdots$
- A typical ordering on integers:
  \[
  x \prec y = x < y \land 0 \leq y
  \]

Syntax

New syntax construct:

\[
e ::= \begin{array}{l}
\text{while } e \text{ invariant } t \prec \text{ variant } t, \prec \text{ do } e
\end{array}
\]

Example:

\[
\{ \ y \geq 0 \ \}\ L:
\begin{array}{l}
\text{while } y > 0 \ do \\
\text{invariant } \{ x + y = x_{@L} + y_{@L} \} \\
\text{variant } \{ y \} \\
\ x := x + 1; \ y := y - 1 \\
\{ x = x_{@old} + y_{@old} \land y = 0 \}
\end{array}
\]

Operational semantics

\[
\begin{array}{l}
\Sigma, \Pi, \text{while } c \text{ invariant } t \prec \text{ variant } t, \prec \text{ do } e \Rightarrow \\
\Sigma, \Pi, L:\text{if } c \\
\text{ then } (e; \text{assert } t \prec t_{@L}; \\
\text{while } c \text{ invariant } l \text{ variant } t, \prec \text{ do } e) \\
\text{ else } ()
\end{array}
\]
Weakest Precondition

\[
WP(\text{while } c \text{ invariant } l \text{ variant } t, \prec \text{ do } e, Q) = \\
\] \\
\[ l \land \\
\forall \vec{v}, (l \Rightarrow WP(\text{L : c, if result then WP(e, l \land t \prec t@L) else Q})) \\
[\vec{w} \leftarrow \vec{v}]
\]

Remark: in practice with Why3:

- presence of loop variants tells if one wants to prove termination or not
- warning issued if no variant given
- keyword `diverges` in contract for non-terminating functions
- default ordering determined from type of \( t \)

Examples

Exercise: find adequate variants.

```plaintext
i := 0;
while i \leq 100
\text{invariant } ? \text{ variant } ?
\text{do } i := i+1 \text{ done;}
```

```plaintext
while sum \leq x
\text{invariant } ? \text{ variant } ?
\text{do }
\text{res := res + 1; sum := sum + 2 * res + 1}
\text{done;}
```

Recursive Functions: Termination

If a function is recursive, termination of call can be proved, provided that the function is annotated with a \textit{variant}.

```
let fun \( f(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau \) 
\text{requires } Pre \\
\text{variant } var, \prec \\
\text{writes } \vec{w} \\
\text{ensures } Post \\
\text{body } Body
```

WPs for function call:

\[
WP(f(t_1, \ldots, t_n), Q) = \text{Pre}[x_i \leftarrow t_i] \land \text{var}[x_i \leftarrow t_i] \prec \text{var}@\text{Init} \land \\
\forall \vec{y}, (\text{Post}[x_i \leftarrow t_i][w_j \leftarrow y_j][w_j@\text{Old} \leftarrow w_j] \Rightarrow Q[w_j \leftarrow y_j])
\]

with \textit{Init} a label assumed to be present at the start of \textit{Body}.

Case of mutual recursion

Assume two functions \( f(\vec{x}) \) and \( g(\vec{y}) \) that call each other.

- each should be given its own variant \( v_f \) (resp. \( v_g \)) in their contract
- with the same \textit{well-founded ordering} \( \prec \).

When \( f \) calls \( g(\vec{t}) \) the WP should include

\[
v_g[\vec{y} \leftarrow \vec{t}] \prec v_f@\text{Init}
\]

and symmetrically when \( g \) calls \( f \).
Home Work: McCarthy’s 91 Function

\[ f_{91}(n) = \begin{cases} 
    f_{91}(f_{91}(n + 11)) & \text{if } n \leq 100 \\
    n - 10 & \text{else} 
\end{cases} \]

Find adequate specifications.

```text
let fun f91(n:int): int
  requires ?
  variant ?
  writes ?
  ensures ?
body
  if n \leq 100 then f91(f91(n + 11)) else n - 10
```

See file `mccarthy.mlw`

Outline

Syntax extensions

Functions and Functions Calls

Termination, Variants

Advanced Modeling of Programs

(First-Order) Logic as a Modeling Language

Axiomatic Definitions

Axiomatic Type Definitions

Programs on Arrays

About Specification Languages

Specification languages:

- Algebraic Specifications: CASL, Larch
- Set theory: VDM, Z notation, Atelier B
- Higher-Order Logic: PVS, Isabelle/HOL, HOL4, Coq
- Object-Oriented: Eiffel, JML, OCL
- ...

Case of Why3, ACSL, Dafny: trade-off between

- expressiveness of specifications,
- support by automated provers.

Why3 Logic Language

- (mostly First-order) logic, with type polymorphism à la ML
- Built-in arithmetic (integers and reals)
- Definitions à la ML
  - logic (i.e. pure) functions, predicates
  - structured types, pattern-matching
- Axiomatizations
- Inductive predicates
- Some higher-order features: lambda-expressions are allowed with syntax \( \lambda x : \tau \. t \)

Important note

Function and predicates are always totally defined
Logic Symbols

Logic functions defined as
\[ \text{function } f(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau = e \]

Predicate defined as
\[ \text{predicate } p(x_1 : \tau_1, \ldots, x_n : \tau_n) = e \]

where \( \tau_i, \tau \) are not reference types.

- No recursion allowed
- No side effects
- Defines total functions and predicates

Logic Symbols: Examples

\[
\begin{align*}
\text{function } & \text{sqr(x:int)} = x * x \\
\text{predicate } & \text{prime(x:int)} = \\
& x \geq 2 \land \\
& \forall y, z : \text{int}. \ y \geq 0 \land z \geq 0 \land x = y*z \rightarrow \\
& y = 1 \lor z = 1
\end{align*}
\]

Axiomatic Definitions

Function and predicate declarations of the form

\[
\begin{align*}
\text{function } & f(\tau, \ldots, \tau_n) : \tau \\
\text{predicate } & p(\tau, \ldots, \tau_n)
\end{align*}
\]

together with axioms

\[
\text{axiom id : formula}
\]

specify that \( f \) (resp. \( p \)) is any symbol satisfying the axioms.

Axiomatic Definitions

Example: division

\[
\begin{align*}
\text{function } & \text{div(real,real):real} \\
\text{axiom mul_div:} & \forall x, y. \ y \neq 0 \rightarrow \text{div}(x,y) \times y = x
\end{align*}
\]

Example: factorial

\[
\begin{align*}
\text{function } & \text{fact(int):int} \\
\text{axiom fact0:} & \text{fact}(0) = 1 \\
\text{axiom factn:} & \forall n : \text{int}. \ n \geq 1 \rightarrow \text{fact}(n) = n \times \text{fact}(n-1)
\end{align*}
\]
Axiomatic Definitions

Functions/predicates are typically underspecified. ⇒ model partial functions in a logic of total functions.

About soundness: axioms may introduce inconsistencies.

Underspecified Logic Functions and Run-time Errors

Error “Division by zero” can be modeled by an abstract function

```
val div_real(x:real,y:real):real
  requires y ≠ 0.0
  ensures result = div(x,y)
```

Reminder

Execution blocks when an invalid annotation is met

Axiomatic Definitions: Example of Factorial

Exercise: Find appropriate precondition, postcondition, loop invariant, and variant, for this program:

```
let fun fact_imp (x:int): int
  requires ?
  ensures ?
body
  let ref y = 0 in
  let ref res = 1 in
  while y < x do
    y := y + 1;
    res := res * y
  done;
res
```

See file fact.mlw

Axiomatic Type Definitions

Type declarations of the form

```
type τ
```

Example: colors

```
type color
  function blue: color
  function red: color
axiom distinct: red ≠ blue
```

Polymorphic types:

```
type τ α₁⋯αₖ
```

where α₁⋯αₖ are type parameters.
### Example: Sets

```plaintext
type set α
function empty: set α
function single(α): set α
function union(set α, set α): set α
axiom union_assoc: forall x y z: set α.
  union(union(x,y),z) = union(x,union(y,z))
axiom union_comm: forall x y: set α.
  union(x,y) = union(y,x)
predicate mem(α, set α)
axiom mem_empty: forall x: α.
  ¬ mem(x, empty)
axiom mem_single: forall x y: α.
  mem(x, single(y)) ↔ x = y
axiom mem_union: forall x: α, y z: set α.
  mem(x, union(y,z)) ↔ mem(x,y) ∨ mem(x,z)
```

### Arrays as References on Pure Maps

Axiomatization of maps from int to some type α:

```plaintext
type map α
function select(map α, int): α
function store (map α, int, α): map α
axiom select_store_eq:
  forall a:map α, i:int, v:α.
  select(store(a, i, v), i) = v
axiom select_store_neq:
  forall a:map α, i j:int, v:α.
  i ≠ j → select(store(a, i, v), j) = select(a, j)
```

- Unbounded indexes.
- `select(a, i)` models the usual notation `a[i].`
- `store` denotes the functional update of a map.

### Arrays as Reference on Maps

- Array variable: variable of type `ref (map α).`
- In a program, the standard assignment operation `a[i] := e`
  is interpreted as `a := store(a, i, e)`
Simple Example

```why3
val a: ref (map int)

let fun test() writes a
    ensures select(a,0) = 13 (* a[0] = 13 *)
body
  a := store(a,0,13); (* a[0] := 13 *)
  a := store(a,1,42) (* a[1] := 42 *)

Exercise: prove this program.
```

Example: Swap

Permute the contents of cells $i$ and $j$ in an array $a$:

```why3
val a: ref (map int)

let fun swap(i:int,j:int)
  requires 0 ≤ i < length a ∧ 0 ≤ j < length a
  writes a
  ensures select(a,i) = select(a@Old,j) ∧
    select(a,j) = select(a@Old,i) ∧
    forall k:int. k ≠ i ∧ k ≠ j →
      select(a,k) = select(a@Old,k)
body
  let tmp = select(a,i) in (* tmp :=a[i]*)
  a := store(a,i,select(a,j)); (* a[i] :=a[j] *)
  a := store(a,j,tmp) (* a[j] :=tmp *)
```

Arrays as Reference on pairs (length,map)

- Goal: model "out-of-bounds" run-time errors
- Array variable: reference on a pair $(\text{length}, \text{map } \alpha)$.
- $a[i]$ interpreted as $\text{get}(a,i)$
- $a[i] := v$ interpreted as $\text{set}(a,i,v)$

```why3
val get(a:array \alpha,i:int):\alpha
  requires 0 ≤ i < fst(a)
  ensures result = select(snd(a),i)

val set(a:array \alpha,i:int,v:\alpha):unit
  requires 0 ≤ i < fst(a)
  writes a
  ensures fst(a) = fst(a@Old) ∧
    snd(a) = store(snd(a@Old),i,v)
```

In Why3: `use import array.Array syntax: a.length, a[i], a[i]<-v`

Example: Swap in Why3

```why3
val a: array int

let fun swap(i:int,j:int)
  requires { 0 ≤ i < length a ∧ 0 ≤ j < length a }
  writes { a }
  ensures { a[i] = old a[j] ∧ a[j] = old a[i] }
  ensures { forall k:int. 0 ≤ k < length a ∧ k ≠ i ∧ k ≠ j →
    a[k] = old a[k] }
body
  let tmp = a[i] in a[i] <- a[j]; a[j] <- tmp
```

Exercises on Arrays

- Prove Swap using WP.
- Prove the program

```ml
let fun test() requires
  select(a,0) = 13 ∧ select(a,1) = 42 ∧
  select(a,2) = 64
ensures
  select(a,0) = 64 ∧ select(a,1) = 42 ∧
  select(a,2) = 13
body swap(0,2)
```

- Specify, implement, and prove a program that increments by 1 all cells, between given indexes \( i \) and \( j \), of an array of reals.

Exercise: Search Algorithms

```ml
let search(a: array real, n:int, v:real): int requires 0 ≤ n ensures \{ \_ \} = \_ = ?
```

1. Formalize postcondition: if \( v \) occurs in \( a \), between 0 and \( n-1 \), then result is an index where \( v \) occurs, otherwise result is set to \(-1\).
2. Implement and prove linear search:

   ```ml
   res := -1;
   for each \( i \) from 0 to \( n-1 \): if \( a[i] = v \) then \( res := i \);
   return \( res \)
   ```

   See file `lin_search.mlw`

Home Work: Binary Search

```ml
low = 0; high = n - 1;
while low ≤ 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`

Home Work: “for” loops

Syntax: `for \( i = e_1 \) to \( e_2 \) do \( e \)`

Typing:
- \( i \) visible only in \( e \), and is immutable
- \( e_1 \) and \( e_2 \) must be of type \( \text{int} \), \( e \) must be of type \( \text{unit} \)

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

\[
\begin{align*}
\Sigma, \Pi, \text{for } i = v_1 \text{ to } v_2 \text{ do } & e \Rightarrow \Sigma, \Pi, () \\
\Sigma, \Pi, \text{let } i = v_1 \text{ in } e \tag{let } i = v_1 \text{ in } e \tag{for } i = v_1 + 1 \text{ to } v_2 \text{ do } e \\
\end{align*}
\]
Home Work: “for” loops

Propose a Hoare logic rule for the for loop:

\[
\{ ? \} e \{ ? \} \\
\{ ? \} \text{for } i = v_1 \text{ to } v_2 \text{ do } e \{ ? \}
\]

Propose a rule for computing the WP:

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

That's all for today, Merry Christmas!

(Check your mail for the project text to come!)