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.

```
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 Σ, Π is it does not block on an assertion or a loop invariant
  - If $\Sigma, \Pi \models \mathrm{WP}(e, Q)$ then e executes safely in Σ, Π, 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: Euclide's algorithm, on two global variables `x,y`

```
Euclide:
    requires  ?
    ensures   ?
  = while y > 0 do
      let r = mod x y in 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*:

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

Add in syntax of *expressions*:

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

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

```
{ true }
let v = r in (r := v + 42; v)
{ r = r@Old + 42 ∧ result = r@Old }
```

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

SUM revisited:

```
{ y ≥ 0 }
L:
while y > 0 do
  invariant { x + y = x@L + y@L }
  x := x + 1; y := y - 1
{ x = x@Old + y@Old ∧ 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:

$$\begin{array}{rcl} [\![x]\!]_{\Sigma,\Pi} & = & \Sigma(x, Here) \\ [\![x@L]\!]_{\Sigma,\Pi} & = & \Sigma(x, L) \end{array}$$

The operational semantics of expressions is modified as follows

$$\begin{array}{rcl} \Sigma, \Pi, x := val & \rightsquigarrow & \Sigma\{(x, Here) \leftarrow val\}, \Pi, () \\ \Sigma, \Pi, L : e & \rightsquigarrow & \Sigma\{(x, L) \leftarrow \Sigma(x, Here) \mid x \text{ any variable}\}, \Pi, e \end{array}$$

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 $x@L + y@K + 2 + x@Here$.

New rules for WP

New rules for computing WP:

$$\begin{array}{rcl} \mathrm{WP}(x := t, Q) & = & Q[x@Here \leftarrow t] \\ \mathrm{WP}(L : e, Q) & = & \mathrm{WP}(e, Q)[x@L \leftarrow x@Here \mid x \text{ any variable}] \end{array}$$

Exercise:

$$\mathrm{WP}(L : x := x + 42, x@Here > x@L) = ?$$

Example: Euclide's algorithm revisited

```
Euclide:
    requires { x ≥ 0 ∧ y ≥ 0 }
    ensures  { result = gcd(x@Old,y@Old) }
  = L:
    while y > 0 do
      invariant { x ≥ 0 ∧ y ≥ 0 }
      invariant { gcd(x,y) = gcd(x@L,y@L) }
      let r = mod x y in x := y; y := r
    done;
    x
```

See file `euclide_labels.mlw`

Mutable Local Variables

We extend the syntax of expressions with

$$e ::= \texttt{let ref}\ id = e\ \texttt{in}\ e$$

Example: isqrt revisited

```
val x, res : ref int

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

Operational Semantics

$$\Sigma, \Pi, e \rightsquigarrow \Sigma', \Pi', e'$$

Π no longer contains just immutable variables.

$$\frac{\Sigma, \Pi, e_1 \rightsquigarrow \Sigma', \Pi', e_1'}{\Sigma, \Pi, \texttt{let ref}\ x = e_1\ \texttt{in}\ e_2 \rightsquigarrow \texttt{let ref}\ x = e_1'\ \texttt{in}\ e_2}$$

$$\frac{}{\Sigma, \Pi, \texttt{let ref}\ x = v\ \texttt{in}\ e \rightsquigarrow \Sigma, \Pi\{(x, Here) \mapsto v\}, e}$$

$$\frac{x \text{ local variable}}{\Sigma, \Pi, x := v \rightsquigarrow \Sigma, \Pi\{(x, Here) \mapsto v\}, e}$$

And labels too.

Mutable Local Variables: WP rules

Rules are exactly the same as for global variables

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

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

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

Home Work

- Extend the post-condition of Euclide algorithm to express the Bezout property:

$$\exists a, b, result = x * a + y * b$$

- 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:

$$
\begin{array}{rcl}
\mathit{prog} & ::= & \mathit{decl}^* \\
\mathit{decl} & ::= & \mathit{vardecl} \mid \mathit{fundecl} \\
\mathit{vardecl} & ::= & \text{val } \mathit{id} : \text{ref } \mathit{basetype} \\
\mathit{fundecl} & ::= & \text{let fun } \mathit{id}(\ (\mathit{param},)^*\ ) : \mathit{basetype} \\
 & & \quad \mathit{contract}\ \text{body } e \\
\mathit{param} & ::= & \mathit{id} : \mathit{basetype} \\
\mathit{contract} & ::= & \text{requires } t \text{ writes } (\mathit{id},)^* \text{ ensures } t
\end{array}
$$

Function definition:

- Contract:
  - pre-condition,
  - post-condition (label *Old* available),
  - assigned variables: clause writes .
- Body: expression.

Example: isqrt

```
let fun isqrt(x:int): int
  requires x ≥ 0
  ensures result ≥ 0 ∧
          sqr(result) ≤ x < sqr(result + 1)
body
  let ref res = 0 in
  let ref sum = 1 in
  while sum ≤ x do
    res := res + 1;
    sum := sum + 2 * res + 1
  done;
  res
```

Example using *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:

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

Well-formed definitions:

$$\frac{\begin{array}{ll} \Gamma' = \{x_i : \tau_i \mid 1 \leq i \leq n\} \cdot \Gamma & \vec{w} \subseteq \Gamma \\ \Gamma' \vdash Pre, Post : formula & \Gamma' \vdash Body : \tau \\ \vec{w}_g \subseteq \vec{w} \text{ for each call } g & y \in \vec{w} \text{ for each assign } y \end{array}}{\Gamma \vdash d : wf}$$

where Γ contains the global declarations. Well-typed function calls:

$$\frac{\Gamma \vdash t_i : \tau_i}{\Gamma \vdash f(t_1, \ldots, t_n) : \tau}$$

Note: t_i are immutable expressions.

Operational Semantics

function $f(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau$
  requires Pre
  writes $\vec{w}$
  ensures $Post$
  body $Body$

$$\frac{\Pi' = \{x_i \mapsto [\![t_i]\!]_{\Sigma,\Pi}\} \qquad \Sigma, \Pi' \models Pre}{\Sigma, \Pi, f(t_1, \ldots, t_n) \rightsquigarrow \Sigma, \Pi, (Old : \texttt{frame}(\Pi', Body, Post))}$$

Operational Semantics of Function Call

`frame` is a dummy operation that keeps track of the local variables of the callee:

$$\frac{\Sigma, \Pi, e \rightsquigarrow \Sigma', \Pi', e'}{\Sigma, \Pi'', (\texttt{frame}(\Pi, e, P)) \rightsquigarrow \Sigma', \Pi'', (\texttt{frame}(\Pi', e', P))}$$

It also checks that the post-condition holds at the end:

$$\frac{\Sigma, \Pi' \models P[\text{result} \leftarrow v]}{\Sigma, \Pi, (\texttt{frame}(\Pi', v, P)) \rightsquigarrow \Sigma, \Pi, v}$$

WP Rule of Function Call

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

$$\begin{array}{l} \mathrm{WP}(f(t_1, \ldots, t_n), Q) = Pre[x_i \leftarrow t_i] \wedge \\ \quad \forall \vec{v},\ (Post[x_i \leftarrow t_i, w_j \leftarrow v_j, w_j@Old \leftarrow w_j] \Rightarrow Q[w_j \leftarrow v_j]) \end{array}$$

Modular proof

When calling function f, only the contract of f is visible, not its body

Example: isqrt(42)

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

```
val isqrt(x:int): int
  requires x ≥ 0
  writes (nothing)
  ensures result ≥ 0 ∧
          sqr(result) ≤ x < sqr(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

```
val res: ref int

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

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

Soundness Theorem for a Complete Program

Assuming that for each function defined as

```
let fun f(x1 : τ1, ..., xn : τn) : τ
  requires Pre
  writes w⃗
  ensures Post
  body Body
```

we have

- variables assigned in *Body* belong to $\vec{w}$,
- $\models Pre \Rightarrow \mathrm{WP}(Body, Post)[w_i@Old \leftarrow w_i]$ holds,

then for any formula Q and any expression e,
if $\Sigma, \Pi \models \mathrm{WP}(e, Q)$ then execution of Σ, Π, 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 $val_1 \succ val_2 \succ val_3 \succ \cdots$
- A typical ordering on integers:

$$x \prec y \quad = \quad x < y \wedge 0 \leq y$$

Syntax

New syntax construct:

$$e \quad ::= \quad \texttt{while}\ e\ \texttt{invariant}\ I\ \texttt{variant}\ t, \prec\ \texttt{do}\ e$$

Example:

```
{ y ≥ 0 }
L:
while y > 0 do
  invariant { x + y = x@L + y@L }
  variant { y }
  x := x + 1; y := y - 1
{ x = x@Old + y@Old ∧ y = 0 }
```

Operational semantics

$$\frac{[\![I]\!]_{\Sigma,\Pi} \text{ holds}}{\begin{array}{l}\Sigma, \Pi, \texttt{while}\ c\ \texttt{invariant}\ I\ \texttt{variant}\ t, \prec\ \texttt{do}\ e \rightsquigarrow \\ \quad \Sigma, \Pi, L : \texttt{if}\ c \\ \qquad \texttt{then}\ (e; \texttt{assert}\ t \prec t@L; \\ \qquad\quad \texttt{while}\ c\ \texttt{invariant}\ I\ \texttt{variant}\ t, \prec\ \texttt{do}\ e) \\ \qquad \texttt{else}\ ()\end{array}}$$

Weakest Precondition

$$\begin{array}{l}\mathrm{WP}(\texttt{while } c \texttt{ invariant } I \texttt{ variant } t, \prec \texttt{ do } e, Q) = \\ \quad I \wedge \\ \quad \forall \vec{v}, (I \Rightarrow \mathrm{WP}(L : c, \texttt{if } \mathit{result} \texttt{ then } \mathrm{WP}(e, I \wedge t \prec t@L) \texttt{ else } Q)) \\ \qquad [w_i \leftarrow v_i]\end{array}$$

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.

```
i := 0;
while i ≤ 100
invariant ? variant ?
do i := i+1 done;
```

```
while sum ≤ x
invariant ? variant ?
do
  res := res + 1; sum := sum + 2 * res + 1
done;
```

Recursive Functions: Termination

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

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

WP for function call:

$$\begin{array}{l}\mathrm{WP}(f(t_1, \ldots, t_n), Q) = \mathit{Pre}[x_i \leftarrow t_i] \wedge \mathit{var}[x_i \leftarrow t_i] \prec \mathit{var}@\mathit{Init} \wedge \\ \quad \forall \vec{y},\ (\mathit{Post}[x_i \leftarrow t_i][w_j \leftarrow y_j][w_j@\mathit{Old} \leftarrow w_j] \Rightarrow Q[w_j \leftarrow y_j])\end{array}$$

with *Init* a label assumed to be present at the start of *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* well-founded ordering $\prec$.

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

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

and symmetrically when g calls f

Home Work: McCarthy's 91 Function

$$f91(n) = \texttt{if } n \leq 100 \texttt{ then } f91(f91(n+11)) \texttt{ else } n - 10$$

Find adequate specifications.

```
let fun f91(n:int): int
  requires ?
  variant ?
  writes ?
  ensures ?
body
  if n ≤ 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 `\x:τ. t`

Important note

Function and predicates are always totally defined

Logic Symbols

Logic functions defined as

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

Predicate defined as

predicate $p(x_1 : \tau_1, \ldots, x_n : \tau_n) = e$

where τ_i, τ are not reference types.

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

Logic Symbols: Examples

```
function sqr(x:int) = x * x

predicate prime(x:int) =
  x ≥ 2 ∧
  forall y z:int. y ≥ 0 ∧ z ≥ 0 ∧ x = y*z →
    y=1 ∨ z=1
```

Axiomatic Definitions

Function and predicate declarations of the form

function $f(\tau, \ldots, \tau_n) : \tau$
predicate $p(\tau, \ldots, \tau_n)$

together with axioms

axiom *id* : *formula*

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

Axiomatic Definitions

Example: division

```
function div(real,real):real
axiom mul_div:
  forall x,y. y≠0 → div(x,y)*y = x
```

Example: factorial

```
function fact(int):int
axiom fact0:
  fact(0) = 1
axiom factn:
  forall n:int. n ≥ 1 → fact(n) = n * fact(n-1)
```

Axiomatic Definitions

- Functions/predicates are typically underspecified.
  $\Rightarrow$ 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 $\tau\ \alpha_1 \cdots \alpha_k$

where $\alpha_1 \cdots \alpha_k$ are type parameters.

Example: Sets

```
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)
```

Outline

Syntax extensions

Functions and Functions Calls

Termination, Variants

Advanced Modeling of Programs

Programs on Arrays

Arrays as References on Pure Maps

Axiomatization of maps from int to some type α:

```
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

```
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*:

```
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 `(length,map α)`.
- `a[i]` interpreted as `get(a,i)`
- `a[i] := v` interpreted as `set(a,i,v)`

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

val set(a:array α,i:int,v:α):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

```
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] }
=
  let tmp = a[i] in a[i] <- a[j]; a[j] <- tmp
```

Exercises on Arrays

- Prove Swap using WP.
- Prove the program

```
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

```
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:
   $res := -1$;
   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

$low = 0$; $high = n - 1$;
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`

Home Work: "for" loops

Syntax: $\texttt{for}\ i = e_1\ \texttt{to}\ e_2\ \texttt{do}\ e$
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)

$$\frac{v_1 > v_2}{\Sigma, \Pi, \texttt{for}\ i = v_1\ \texttt{to}\ v_2\ \texttt{do}\ e \rightsquigarrow \Sigma, \Pi, ()}$$

$$\frac{v_1 \leq v_2}{\Sigma, \Pi, \texttt{for}\ i = v_1\ \texttt{to}\ v_2\ \texttt{do}\ e \rightsquigarrow \Sigma, \Pi, \begin{array}{l}(\texttt{let}\ i = v_1\ \texttt{in}\ e);\\ (\texttt{for}\ i = v_1 + 1\ \texttt{to}\ v_2\ \texttt{do}\ e)\end{array}}$$

Home Work: “for” loops

Propose a Hoare logic rule for the `for` loop:

$$\frac{\{?\}e\{?\}}{\{?\}\texttt{for}\ i = v_1\ \texttt{to}\ v_2\ \texttt{do}\ e\{?\}}$$

Propose a rule for computing the WP:

$$\mathrm{WP}(\texttt{for}\ i = v_1\ \texttt{to}\ v_2\ \texttt{invariant}\ I\ \texttt{do}\ e, Q) = ?$$

That’s all for today, Merry Christmas !

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