Simple Syntax Extensions
(labels, local mutable variables)

Functions and Function calls
Proving Termination

More on Specification Languages and Application to Arrays

Claude Marché

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

December 13th, 2022

Reminder of the last lecture

- Logics and automated prover capabilities
  - propositional logic
  - first-order logic
  - theories: equality, integer arithmetic
- classical Floyd-Hoare logic
  - very simple “IMP” programming language
  - deduction rules for triples $\{Pre\}s\{Post\}$
- weakest liberal pre-conditions (Dijkstra)
  - function $\mathrm{WLP}(s, Q)$ returning a logic formula
  - soundness: if $P \rightarrow \mathrm{WLP}(s, Q)$ then triple $\{P\}s\{Q\}$ is valid
- main “creative” activity: *discovering loop invariants*

Reminder of the last lecture (continued)

- 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 if 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
- Exercices

Exercise 1

Consider the following (inefficient) program for computing the sum $a + b$

```
x <- a; y <- b;
while y > 0 do
  x <- x + 1; y <- y - 1
```

(Why3 file to fill in: `imp_sum.mlw`)

- Propose a post-condition stating that the final value of x is the sum of the values of a and b
- Find an appropriate loop invariant
- Prove the program

Exercise 2

The following program is one of the original examples of Floyd

```
q <- 0; r <- x;
while r >= y do
  r <- r - y; q <- q + 1
```

(Why3 file to fill in: `imp_euclidean_div.mlw`)

- Propose a formal precondition to express that x is assumed non-negative, y is assumed positive, and a formal post-condition expressing that q and r are respectively the quotient and the remainder of the Euclidean division of x by y
- Find appropriate loop invariants and prove the correctness of the program

This Lecture's Goals

- Extend that language:
  - Labels for reasoning on the past, local mutable variables
  - Sub-programs, *function calls*, *modular reasoning*
  - Limitations of modular reasoning: subcontract weaknesses, non-inductive invariants
- Analyzing *Termination*
  - prove termination when wanted
- (First-order) logic as a *modeling language*
  - Definitions of new types, product types
  - Definitions of functions, of predicates
  - Axiomatizations
- Application:
  - a bit of higher-order logic
  - program on *Arrays*

Outline

Syntax extensions
- Labels
- Local Mutable Variables
- Functions and Functions Calls

Termination, Variants

Advanced Modeling of Programs

Programs on Arrays

Labels: motivation

Ability to refer to past values of variables

```
{ 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: 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

Implicitly declared labels:

- *Here*, available in every formula
- *Old*, available everywhere except pre-conditions

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

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

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:

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

Exercise:

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

Example: computation of the GCD

(assuming notion of greatest common divisor exists in the logic)

Euclid's algorithm:

```
    requires { x >= 0 /\ y >= 0 }
    ensures  { result = gcd(x@Old,y@Old) }
  = L:
    while y > 0 do
      invariant { ? }
      let r = mod x y in x <- y; y <- r
    done;
    x
```

See file `gcd_euclid_labels.mlw`

Mutable Local Variables

We extend the syntax of expressions with

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

(note: I use "`ref`" instead of "`mut`" because of Why3)

Example: isqrt revisited

```
val ref x : int
val ref res : int

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

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

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

$$\frac{x \text{ local variable}}{\Sigma, \pi, x \texttt{ <- } v \rightsquigarrow \Sigma, \pi\{(x, Here) \leftarrow v\}, e}$$

Mutable Local Variables: WP rules

Rules are exactly the same as for global variables

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

$$\mathrm{WP}(x \texttt{ <- } e, Q) = \mathrm{WP}(e, Q[x \leftarrow \mathit{result}])$$

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

Functions

Program structure:

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

Function definition:

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

Example: isqrt

```
let 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 ref res: int

let incr(x:int)
  requires true
  writes res
  ensures res = res@Old + x
body
  res <- res + x
```

Typing

Definition d of function f:

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

Well-formed definitions:

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

where Γ contains the global declarations

Typing: function calls

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

Well-typed function calls:

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

Note: for simplicity the expressions t_i are assumed without side-effect (introduce extra let-expression if needed)

Operational Semantics of a Function Call

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

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

A *call frame* is a pair $(\pi, Post)$ of a local stack and a formula
Π denotes a *stack of call frames*

Blocking Semantics

Execution blocks at call if pre-condition does not hold

Operational Semantics of returning from Function Call

We check that the *post-condition* holds at the end:

$$\frac{\Sigma, \pi \models Post[\mathrm{result} \leftarrow v]}{\Sigma, (\pi, Post) \cdot \Pi, v \rightsquigarrow \Sigma, \Pi, v}$$

Blocking Semantics

Execution blocks at return if post-condition does not hold

WP Rule of Function Call

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

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

Modular Proof Methodology

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

Example: isqrt(42)

Exercise: prove that $\{true\}isqrt(42)\{\mathrm{result} = 6\}$ 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 ref res: 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 $f(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau$
  requires Pre
  writes $\vec{w}$
  ensures $Post$
  body $Body$

we have

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

then for any formula Q, any expression e, any configuration (Σ, π):

if $\Sigma, \pi \models \mathrm{WP}(e, Q)$ then execution of Σ, π, e is *safe*

Remark: (mutually) recursive functions are allowed

Limitations of modular reasoning

```
let f (x:int) : int
  ensures { result > x }
  = x+1

let g () =
  let a = f(0) in
  assert { a = 1 }
```

Subcontract weakness

A program can be *safe* (never blocks on annotations) and yet not being provable

Non-inductive loop invariants

```
let ref i = 0 in
while i < 2 do
  invariant { i <> 1 }
  i <- i+2;
done
```

Weakness of loop invariants

An invariant might be valid (the program is safe) and yet not be provably preserved by an arbitrary loop iteration

Inductive invariants

A loop invariant is called *inductive* when it can be proved initially valid and preserved by loop iterations

In other words: a loop invariant may be valid (in the sense of safety) and yet not being inductive

Limitations of modular reasoning (case of loops)

```
let ref i = 5 in
while i < 10 do
  invariant { i >= 0 }
  i <- i+2;
done;
assert { i = 11 }
```

Subcontract weakness (for loop)

A program can be *safe* (never blocks on annotations) and yet not being provable

Outline

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:

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

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

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

(new parts shown in red)

Weakest Precondition

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

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
  variant ?
do i <- i+1
done;
```

```
while sum <= x
  variant ?
do
  res <- res + 1; sum <- sum + 2 * res + 1
done;
```

Solutions:

```
variant 100 - i
```

```
invariant res >= 0
variant x - sum
```

Recursive Functions: Termination

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

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

WP for function call:

$$\mathrm{WP}(f(t_1, \ldots, t_n), Q) = Pre[x_i \leftarrow t_i] \wedge var[x_i \leftarrow t_i] \prec var@Old \wedge$$
$$\forall \vec{y},\ (Post[x_i \leftarrow t_i][w_j \leftarrow y_j][w_j@Old \leftarrow w_j] \rightarrow Q[w_j \leftarrow y_j])$$

Example of variant on a recursive function

```
let fib (x:int) : int
  variant ?
  body
    if x <= 1 then 1 else fib (x-1) + fib (x-2)
```

Solution:

```
  variant x
```

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

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

and symmetrically when g calls f

Home Work 1: 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 f91(n:int): int
  requires ?
  variant ?
  writes ?
  ensures ?
body
  if n <= 100 then f91(f91(n + 11)) else n - 10
```

Use canvas file `mccarthy.mlw`

Outline

Syntax extensions

Termination, Variants

Advanced Modeling of Programs
- (First-Order) Logic as a Modeling Language
- Axiomatic 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

- (First-order) logic, built-in arithmetic (integers and reals)
- *Definitions* à la ML
  - logic (i.e. pure) *functions, predicates*
  - structured types, pattern-matching (next lecture)
- *type polymorphism* à la ML
- *higher-order logic as a built-in theory of functions*
- Axiomatizations
- Inductive predicates (next lecture)

Important note

Logic functions and predicates are *always totally defined*

Definition of new Logic Symbols

Logic functions defined as

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

Predicate defined as

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

where τ_i, τ are logic types (not references)

- *No recursion allowed* (yet)
- *No side effects*
- Defines *total* functions and predicates

Logic Symbols: Examples

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

predicate divides(x:int,y:int) =
  exists z:int. y = x * z

predicate is_prime(x:int) =
  x >= 2 /\
  forall y z:int. y >= 0 /\ z >= 0 /\ x = y*z ->
    y=1 \/ z=1
```

Definition of new logic types: Product Types

- Tuples types are built-in:

```
type pair = (int, int)
```

- Record types can be defined:

```
type point = { x:real; y:real }
```

Fields are immutable

- We allow let with pattern, e.g.

```
let (a,b) = ... in ...
let { x = a; y = b } = ... in ...
```

- Dot notation for records fields, e.g.

```
p.x + p.y
```

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)
```

Exercise: axiomatize the GCD

Axiomatic Definitions

- Functions/predicates are typically underspecified
  ⇒ we can model partial functions in a logic of total functions

Warning about soundness

Axioms may introduce *inconsistencies*

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

implies `1 = div(1,0)*0 = 0`

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

Outline

Higher-order logic as a built-in theory

- type of *maps* : $\tau_1 \rightarrow \tau_2$
- lambda-expressions: `fun` $x : \tau$ `->` t

Definition of selection function:

```
function select (f : α → β) (x : α) : β = f x
```

Definition of function update:

```
function store (f : α → β) (x : α) (v : β) : α → β =
  fun (y : α) -> if x = y then v else f y
```

SMT (first-order) theory of “functional arrays”

```
lemma select_store_eq: forall f:α->β, x:α, v:β.
  select(store(f,x,v),x) = v
lemma select_store_neq: forall f:α->β, x y:α, v:β.
  x <> y -> select(store(f,x,v),y) = select(f,y)
```

Arrays as Mutable Variables of type "Map"

- Array variable: mutable variable of type `int ->` α
- In a program, the standard assignment operation

  `a[i] <- e`

  is interpreted as

  `a <- store(a,i,e)`

Simple Example

```
val ref a: int -> int

let 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

Simple Example

$$
\begin{array}{rl}
 & WP((a \leftarrow store(a,0,13); \\
 & \quad a \leftarrow store(a,1,42)), select(a,0) = 13)) \\
= & WP(a \leftarrow store(a,0,13), \\
 & \quad WP(a \leftarrow store(a,1,42), select(a,0) = 13))) \\
= & WP(a \leftarrow store(a,0,13); select(store(a,1,42),0) = 13) \\
= & select(store(store(a,0,13),1,42),0) = 13 \\
= & select(store(a,0,13),0) = 13 \\
= & 13 = 13 \\
= & true
\end{array}
$$

Note how we use both lemmas *select_store_eq* and *select_store_neq*

Example: Swap

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

```
val ref a: int -> int

let swap(i:int,j:int)
  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 Variables of Type “length × map”

- Goal: model “out-of-bounds” run-time errors
- Array variable: mutable variable of type `array` α

```
type array 'a = { length : int; elts : int -> 'a}

val get (ref a:array 'a) (i:int) : 'a
  requires 0 <= i < a.length
  ensures  result = select(a.elts,i)

val set (ref a:array 'a) (i:int) (v:'a) : unit
  requires 0 <= i < a.length
  writes   a
  ensures  a.length = a@Old.length /\
           a.elts = store(a@Old.elts,i,v)
```

- `a[i]` interpreted as a call to `get(a,i)`
- `a[i] <- v` interpreted as a call to `set(a,i,v)`

Example: Swap again

```
val ref a: array int

let swap(i:int,j:int)
  requires 0 <= i < a.length /\ 0 <= j < a.length
  writes a
  ensures select(a.elts,i) = select(a@Old.elts,j) /\
          select(a.elts,j) = select(a@Old.elts,i) /\
          forall k:int. 0 <= k < a.length /\ k <> i /\ k <> j ->
            select(a.elts,k) = select(a@Old.elts,k)
body
  let tmp = get(a,i) in   (* tmp <-a[i]*)
  set(a,i,get(a,j));      (* a[i]<-a[j]*)
  set(a,j,tmp)            (* a[j]<-tmp *)
```

Note about Arrays in Why3

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

Example: swap

```
val a: array int

let swap (i:int) (j:int)
  requires { 0 <= i < a.length /\ 0 <= j < a.length }
  writes   { a }
  ensures  { a[i] = old a[j] /\ a[j] = old a[i]}
  ensures  { forall k:int.
               0 <= k < a.length /\ 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 by computing the WP
- Using WP, prove the program

```
let 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)
```

Exercise on Arrays: incrementation

- Specify, implement, and prove a program that increments by 1 all cells, between given indices i and j, of an array of reals

See file `array_incr.mlw`

Exercise: Search Algorithms

```
var a: array real

let search(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 4: 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 5: "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 !

- Next lecture on January 3th
- Several home work exercises to do