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 \((\text{Pre})s(\text{Post})\)
- weakest liberal pre-conditions (Dijkstra)
  - function \(\text{WLP}(s, Q)\) returning a logic formula
  - soundness: if \(P \rightarrow \text{WLP}(s, Q)\) 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 \(\Sigma, \pi\) if it does not block on an assertion or a loop invariant
  - If \(\Sigma, \pi \models \text{WP}(e, Q)\) then \(e\) executes safely in \(\Sigma, \pi\), and if it terminates then \(Q\) valid in the final state
- Exercices

Exercise 1

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

\[
\begin{align*}
  x &\leftarrow a; \ y \leftarrow b; \\
  \text{while } y > 0 \text{ do} & \\
  \quad x &\leftarrow x + 1; \ y \leftarrow y - 1
\end{align*}
\]

(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

```plaintext
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

```plaintext
{ 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:

\[ t ::= x@L \] (labeled variable access)

Add in syntax of expressions:

\[ e ::= L:e \] (labeled expressions)

Typing:

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

Implicitly declared labels:

- \textit{Here}, available in every formula
- \textit{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:

\[
\begin{align*}
[x]_{\Sigma, \pi} &= \Sigma(x, \text{Here}) \\
[x@L]_{\Sigma, \pi} &= \Sigma(x, L)
\end{align*}
\]

The operational semantics of expressions is modified as follows

\[
\begin{align*}
\Sigma, \pi, x < - \text{val} & \implies \Sigma\{(x, \text{Here}) < - \text{val}\}, \pi, () \\
\Sigma, \pi, L : e & \implies \Sigma\{(x, L) < - \Sigma(x, \text{Here}) | x \text{ any variable}\}, \pi, e
\end{align*}
\]

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:

\[
\begin{align*}
\text{WP}(x < - t, Q) &= Q[x@\text{Here} < - t@\text{Here}] \\
\text{WP}(L : e, Q) &= \text{WP}(e, Q)[x@L < - x@\text{Here} | x \text{ any variable}]
\end{align*}
\]

Exercise:

\[
\text{WP}(L : x <- x + 42, x@\text{Here} > x@L) = ?
\]

Example: computation of the GCD

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

Euclid's algorithm:

\[
\begin{align*}
\text{requires} & \{ x >= 0 /\ y >= 0 \} \\
\text{ensures} & \{ \text{result} = \text{gcd}(x@\text{Old}, y@\text{Old}) \}
\end{align*}
\]

= L:

\[
\begin{align*}
\text{while } y > 0 \text{ do} \\
\quad \text{invariant} \{ ? \} \\
\quad \text{let } r = \text{mod } x \ y \text{ in } x <- y; y <- r \\
\quad \text{done;}
\end{align*}
\]

\[
x
\]

See file \texttt{gcd.euclid.labels.mlw}
Mutable Local Variables

We extend the syntax of expressions with

\[ e ::= \text{let ref } id = e \text{ in } e \]

(note: I use “ref” instead of “mut” because of Why3)

Example: isqrt revisited

\[
\begin{align*}
\text{val ref } x &: \text{ int} \\
\text{val ref } res &: \text{ int} \\
res &\gets 0; \\
\text{let ref } sum = 1 \text{ in} \\
\text{while } sum \leq x \text{ do} \\
&\quad \text{res }\gets \text{res }+ 1; \text{sum }\gets \text{sum }+ 2 \times \text{res }+ 1 \\
\text{done}
\end{align*}
\]

Mutable Local Variables: WP rules

Rules are exactly the same as for global variables

\[
\begin{align*}
\text{WP(}\text{let ref } x = e_1 \text{ in } e_2, Q) &= \text{WP}(e_1, \text{WP}(e_2, Q)[x \leftarrow \text{result}]) \\
\text{WP}(x \leftarrow e, Q) &= \text{WP}(e, Q[x \leftarrow \text{result}]) \\
\text{WP}(L : e, Q) &= \text{WP}(e, Q)[x @ L \leftarrow x @ \text{Here} | x \text{ any variable}]
\end{align*}
\]

Operational Semantics

\[\Sigma, \pi, e \rightarrow \Sigma', \pi', e'\]

\(\pi\) no longer contains just immutable variables

\[
\begin{align*}
\Sigma, \pi, e_1 \rightarrow \Sigma', \pi', e'_1 \\
\Sigma, \pi, \text{let ref } x = e_1 \text{ in } e_2 \rightarrow \Sigma', \pi', \text{let ref } x = e'_1 \text{ in } e_2 \\
\Sigma, \pi, \text{let ref } x = v \text{ in } e \rightarrow \Sigma, \pi\{(x, \text{Here}) \leftarrow v\}, e \\
\Sigma, \pi, x < v \rightarrow \Sigma, \pi\{(x, \text{Here}) \leftarrow v\}, e
\end{align*}
\]

Functions

Program structure:

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

Function definition:

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

```
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, ... , x_n: \tau_n): \tau
  requires Pre
  writes \( \vec{w} \)
  ensures Post
body Body
```

Well-formed definitions:

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

\[
\Gamma \vdash d : \text{wf}
\]

where \( \Gamma \) contains the global declarations

Typing: function calls

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

Well-typed function calls:

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

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 \( Body \)

\[
\pi = \{ x_i \mapsto \llbracket t_i \rrbracket_{\Sigma, \pi} \} \quad \Sigma, \pi \models Pre
\]
\[
\Sigma, \Pi, f(t_1, \ldots, t_n) \rightsquigarrow \Sigma, (\pi, Post) \cdot \Pi, (Old : Body)
\]

A call frame is a pair \((\pi, Post)\) of a local stack and a formula \(\Pi\) denotes a stack of call frames.

Blocking Semantics
Execution blocks at call if pre-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 \( Body \)

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

Modular Proof Methodology
When calling function \( f \), only the contract of \( f \) is visible, not its body

Exercise: prove that \{true\}/sqrt(42)\{result = 6\} holds

val isqrt(x:int): int
requires x >= 0
writes (nothing)
ensures result >= 0 /\
\quad \text{sqr(result)} <= x < \text{sqr(result + 1)}

Abstraction of sub-programs
- Keyword \text{val} introduces a function with a contract but without body
- \text{writes} clause is mandatory in that case

Example: isqrt(42)

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

\[
\Sigma, \pi \models Post[result \leftarrow v]
\]
\[
\Sigma, (\pi, Post) \cdot \Pi, v \rightsquigarrow \Sigma, \Pi, v
\]

Blocking Semantics
Execution blocks at return if post-condition does not hold
Example: Incrementation

```ocaml
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

```ocaml
let 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 \( \vec{w} \),
- \( \models Pre \rightarrow WP(Body, Post)[w_i @ Old \leftarrow w_i] \) holds,

then for any formula \( Q \), any expression \( e \), any configuration \( (\Sigma, \pi) \):

\[
\text{if } \Sigma, \pi \models WP(e, Q) \text{ then execution of } \Sigma, \pi, e \text{ is safe}
\]

Remark: (mutually) recursive functions are allowed.

Limitations of modular reasoning

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

```ocaml
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

```ocaml
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

- Syntax extensions
- 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:

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

New syntax construct:

\[
e ::= \text{while } e \text{ invariant } I \text{ variant } t \prec \text{ do } e
\]

Example:

\[
\{ y \geq 0 \}
\]

\[
L: \text{while } y > 0 \text{ do}
\]

\[
invariant \{ x + y = x@L + y@L \}
\]

\[
variant \{ y \}
\]

\[
x \leftarrow x + 1; \ y \leftarrow y - 1
\]

\[
\{ x = x@old + y@old \atop \ y = 0 \}
\]

Operational semantics

\[
\llbracket e \rrbracket_{\Sigma, \pi} \text{ holds}
\]

\[
\Sigma, \pi, \text{while } c \text{ invariant } I \text{ variant } t \prec \text{ do } e \prec
\]

\[
\Sigma, \pi, L : \text{if } c
\]

\[
\text{then } (e; \text{assert } t \prec t@L;
\]

\[
\text{while } c \text{ invariant } I \text{ variant } t \prec \text{ do } e)
\]

\[
\text{else }()
\]

(new parts shown in red)

Weakest Precondition

\[
\text{WP(while } c \text{ invariant } I \text{ variant } t \prec \text{ do } e, Q) =
\]

\[
I \land
\forall \vec{v}. (I \rightarrow \text{WP}(L : c, \text{if } \text{result} \text{ then } W(e, I \land t \prec t@L) \text{ else } Q))
\]

\[
[w_i \leftarrow v_i]
\]

In practice with Why3

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

Examples

Exercise: find adequate variants

\[
i \leftarrow 0;
\]

\[
\text{while } i \leq 100
\]

\[
\text{variant }?
\]

\[
do \ i \leftarrow i+1
\]

\[
done;
\]

\[
\text{while } \text{sum} \leq x
\]

\[
\text{variant }?
\]

\[
do
\]

\[
\text{res} \leftarrow \text{res} + 1; \ \text{sum} \leftarrow \text{sum} + 2 \ast \text{res} + 1
\]

\[
done;
\]

Solutions:

\[
\text{variant } 100 - i
\]

\[
\text{variant } x \geq 0
\]
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 \( \text{var}, \prec \)
writes \( \vec{w} \)
ensures Post
body Body

WP for function call:
\[
WP(f(t_1, \ldots, t_n), Q) = Pre[x_i \leftarrow t_i] \land \text{var}[x_i \leftarrow t_i] \prec \text{var}@\text{Old} \land \\
\forall \vec{y}, (Post[x_i \leftarrow t_i][w_j ← y_j][w_j@\text{Old} ← w_j] → Q[w_j ← y_j])
\]

Example of variant on a recursive function

Example of variant on a recursive function

let fib (x:int) : int
requires ?
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
\[
v_g[\vec{y} ← \vec{t}] \prec v_f@\text{Old}
\]
and symmetrically when \( g \) calls \( f \)

Home Work 1: McCarthy’s 91 Function

\[
f_{91}(n) = \text{if } n ≤ 100 \text{ then } f_{91}(f_{91}(n + 11)) \text{ else } n - 10
\]

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
\[
\text{function } f(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau = e
\]
Predicate defined as
\[
\text{predicate } p(x_1 : \tau_1, \ldots, x_n : \tau_n) = e
\]
where \(\tau, \tau_i\) are logic types (not references)
- No recursion allowed (yet)
- No side effects
- Defines total functions and predicates
Logic Symbols: Examples

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

- Record types can be defined:
  ```plaintext
type point = { x:real; y:real }
```

Fields are immutable

- We allow let with pattern, e.g.
  ```plaintext
  let (a,b) = ...
  in ...
  let { x = a; y = b } = ...
  in ...
  ```

- Dot notation for records fields, e.g.
  ```plaintext
  p.x + p.y
  ```

Axiomatic Definitions

- Function and predicate declarations of the form
  ```plaintext
  function f(τ₁,...,τₙ) : τ
  predicate p(τ₁,...,τₙ)
  ```

- together with axioms
  ```plaintext
  axiom id : formula
  ```

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

Axiomatic Definitions

Example: division

```plaintext
function div(real,real):real
axiom mul_div:
    forall x,y. y<>0 -> div(x,y)*y = x
```

Example: factorial

```plaintext
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: ∀ x, y. div(x, y) * y = x
implies 1 = div(1, 0) * 0 = 0

Outline

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

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

Higher-order logic as a built-in theory

- type of maps: τ₁ → τ₂
- lambda-expressions: fun x: τ → 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: ∀ f: α → β, x: α, v: β.
  select(store(f, x, v), x) = v
lemma select_store_neq: ∀ f: α → β, x: α, 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 \( \text{int} \rightarrow \alpha \)
- In a program, the standard assignment operation
  \[ a[i] <- e \]
  is interpreted as
  \[ a <- \text{store}(a,i,e) \]

**Simple Example**

```ocaml
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

**Example: Swap**

Permute the contents of cells \( i \) and \( j \) in an array \( a \):

```ocaml
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 $array \alpha$

```ocaml
let ref a: 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 $get(a,i)$
- $a[i] \leftarrow v$ interpreted as a call to $set(a,i,v)$

Note about Arrays in Why3

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

Example: swap

```ocaml
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

```ocaml
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

```plaintext
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:
   
   ```plaintext
   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

```plaintext
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 5: “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 int, $e$ must be of type unit

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

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

Propose a Hoare logic rule for the for loop:

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

Propose a rule for computing the WP:

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

That’s all for today, Merry Christmas!

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