

Simple Syntax Extensions

(labels, local mutable variables)

Functions and Function calls

Proving Termination

More on Specification Languages and Application to Arrays

Claude Marché (*François Bobot*)

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

December 15th, 2023

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

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 $WLP(s, Q)$ returning a logic formula
 - ▶ soundness: if $P \rightarrow WLP(s, Q)$ 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 $\Sigma, \pi \models \text{WP}(e, Q)$ then e executes safely in Σ, π , and if it terminates then Q valid in the final state
- ▶ Exercises

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*:

$t ::= x@L$ (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:

- ▶ *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:

$$\begin{aligned}\llbracket x \rrbracket_{\Sigma, \pi} &= \Sigma(x, \text{Here}) \\ \llbracket x@L \rrbracket_{\Sigma, \pi} &= \Sigma(x, L)\end{aligned}$$

The operational semantics of expressions is modified as follows

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

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:

$$\text{WP}(x \leftarrow t, Q) = Q[x@Here \leftarrow t@Here]$$

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

Exercise:

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

Example: computation of the GCD

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

Euclid's algorithm:

```
requires {  $x \geq 0 \wedge y \geq 0$  }  
ensures { result = gcd(x@old,y@old) }  
= L:  
  while  $y > 0$  do  
    invariant { ? }  
    let  $r = \text{mod } x \ y$  in  $x \leftarrow y; y \leftarrow r$   
  done;  
  x
```

See file [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

```
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

Judgements:

$$\Sigma, \pi, e \rightsquigarrow \Sigma', \pi', e'$$

π no longer contains just immutable variables

$$\frac{\Sigma, \pi, e_1 \rightsquigarrow \Sigma', \pi', e'_1}{\Sigma, \pi, \text{let ref } x = e_1 \text{ in } e_2 \rightsquigarrow \Sigma', \pi', \text{let ref } x = e'_1 \text{ in } e_2}$$

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

Operational Semantics

Judgements:

$$\Sigma, \pi, e \rightsquigarrow \Sigma', \pi', e'$$

π no longer contains just immutable variables

$$\frac{\Sigma, \pi, e_1 \rightsquigarrow \Sigma', \pi', e'_1}{\Sigma, \pi, \text{let ref } x = e_1 \text{ in } e_2 \rightsquigarrow \Sigma', \pi', \text{let ref } x = e'_1 \text{ in } e_2}$$

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

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

Mutable Local Variables: WP rules

Rules are exactly the same as for global variables

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

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

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

Functions

Program structure:

prog ::= *decl**

decl ::= *vardecl* | *fundekl*

vardecl ::= **val ref** *id* : *basetype*

Functions

Program structure:

prog ::= *decl**

decl ::= *vardecl* | *fundekl*

vardecl ::= **val ref** *id* : *basetype*

fundekl ::= **let** *id*((*param*,)*):*basetype*
 contract **body** *e*

param ::= *id* : *basetype*

contract ::= **requires** *t* **writes** (*id*,)* **ensures** *t*

Functions

Program structure:

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

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:

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

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

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

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:

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

Blocking Semantics

Execution blocks at return if post-condition does not hold

WP Rule of Function Call

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

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

Modular Proof Methodology

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

Example: isqrt(42)

Exercise: prove that $\{true\}isqrt(42)\{result = 6\}$ 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 $f(x_1 : \tau_1, \dots, x_n : \tau_n) : \tau$
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 (Σ, π) :

if $\Sigma, \pi \models WP(e, Q)$ 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

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 $val_1 \succ val_2 \succ val_3 \succ \dots$)
- ▶ A typical ordering on integers:

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

Syntax

New syntax construct:

$e ::= \text{while } e \text{ invariant } / \text{variant } t, \prec \text{ do } e$

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

$$\frac{\llbracket I \rrbracket_{\Sigma, \pi} \text{ holds}}{\Sigma, \pi, \text{while } C \text{ invariant / variant } t, \prec \text{ do } e \rightsquigarrow \Sigma, \pi, \textcolor{red}{L} : \text{if } C \text{ then } (e; \textcolor{red}{\text{assert } t \prec t@L; \text{while } C \text{ invariant / variant } t, \prec \text{ do } e}) \text{ else } ()}$$

(new parts shown in red)

Weakest Precondition

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

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;
```

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

WP for function call:

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

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)
```

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

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

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
- ▶ ...

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 (but ad-hoc Polymorphism)
- ▶ *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 $f(\tau, \dots, \tau_n) : \tau$
predicate $p(\tau, \dots, \tau_n)$

together with *axioms*

axiom id : formula

Semantics

these declarations specify that f (resp. p) is **any logic function** (resp. predicate) satisfying the axioms

Axiomatic Definitions

Example: division

```
function div(real,real):real  
axiom mul_div:  
  forall x,y.  $y \neq 0 \rightarrow \text{div}(x,y) * y = x$ 
```

Axiomatic Definitions

Example: division

```
function div(real,real):real
axiom mul_div:
  forall x,y.  $y \neq 0 \rightarrow \text{div}(x,y) * y = x$ 
```

Example: factorial

```
function fact(int):int
axiom fact0:
  fact(0) = 1
axiom factn:
  forall n:int.  $n \geq 1 \rightarrow \text{fact}(n) = n * \text{fact}(n-1)$ 
```

Exercise: axiomatize the GCD

Axiomatic Definitions

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

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

Syntax extensions

Termination, Variants

Advanced Modeling of Programs

Programs on Arrays

Higher-order logic as a built-in theory

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

Definition of selection function:

```
function select ( $f : \alpha \rightarrow \beta$ ) ( $x : \alpha$ ) :  $\beta = f\ x$ 
```

Definition of function update:

```
function store ( $f : \alpha \rightarrow \beta$ ) ( $x : \alpha$ ) ( $v : \beta$ ) :  $\alpha \rightarrow \beta =$   
  fun ( $y : \alpha$ ) -> if  $x = y$  then  $v$  else  $f\ y$ 
```

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

```
lemma select_store_eq: forall  $f : \alpha \rightarrow \beta$ ,  $x : \alpha$ ,  $v : \beta$ .  
  select(store( $f, x, v$ ),  $x$ ) =  $v$ 
```

```
lemma select_store_neq: forall  $f : \alpha \rightarrow \beta$ ,  $x\ y : \alpha$ ,  $v : \beta$ .  
   $x \neq y \rightarrow$  select(store( $f, x, v$ ),  $y$ ) = select( $f, y$ )
```

Arrays as Mutable Variables of type “Map”

- ▶ Array variable: mutable variable of type `int ->  $\alpha$`
- ▶ 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

$WP((a \leftarrow \text{store}(a, 0, 13);$
 $a \leftarrow \text{store}(a, 1, 42)), \text{select}(a, 0) = 13))$

Simple Example

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

Simple Example

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

Simple Example

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

Simple Example

$WP((a \leftarrow \text{store}(a, 0, 13);$
 $a \leftarrow \text{store}(a, 1, 42)), \text{select}(a, 0) = 13))$
 $= WP(a \leftarrow \text{store}(a, 0, 13),$
 $WP(a \leftarrow \text{store}(a, 1, 42), \text{select}(a, 0) = 13)))$
 $= WP(a \leftarrow \text{store}(a, 0, 13); \text{select}(\text{store}(a, 1, 42), 0) = 13)$
 $= \text{select}(\text{store}(\text{store}(a, 0, 13), 1, 42), 0) = 13$
 $= \text{select}(\text{store}(a, 0, 13), 0) = 13$

Simple Example

$WP((a \leftarrow \text{store}(a, 0, 13);$
 $a \leftarrow \text{store}(a, 1, 42)), \text{select}(a, 0) = 13))$
= $WP(a \leftarrow \text{store}(a, 0, 13),$
 $WP(a \leftarrow \text{store}(a, 1, 42), \text{select}(a, 0) = 13)))$
= $WP(a \leftarrow \text{store}(a, 0, 13); \text{select}(\text{store}(a, 1, 42), 0) = 13)$
= $\text{select}(\text{store}(\text{store}(a, 0, 13), 1, 42), 0) = 13$
= $\text{select}(\text{store}(a, 0, 13), 0) = 13$
= $13 = 13$

Simple Example

$WP((a \leftarrow \text{store}(a, 0, 13);$
 $a \leftarrow \text{store}(a, 1, 42)), \text{select}(a, 0) = 13))$
= $WP(a \leftarrow \text{store}(a, 0, 13),$
 $WP(a \leftarrow \text{store}(a, 1, 42), \text{select}(a, 0) = 13)))$
= $WP(a \leftarrow \text{store}(a, 0, 13); \text{select}(\text{store}(a, 1, 42), 0) = 13)$
= $\text{select}(\text{store}(\text{store}(a, 0, 13), 1, 42), 0) = 13$
= $\text{select}(\text{store}(a, 0, 13), 0) = 13$
= $13 = 13$
= true

Simple Example

```
WP((a <- store(a, 0, 13);  
    a <- store(a, 1, 42)), select(a, 0) = 13))  
= WP(a <- store(a, 0, 13),  
    WP(a <- store(a, 1, 42), select(a, 0) = 13)))  
= WP(a <- 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
```

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 $\times$ map”

- ▶ Goal: model “out-of-bounds” run-time errors
- ▶ Array variable: mutable variable of type `array  $\alpha$`

```
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*:
 $res \leftarrow -1$;
 for each i from 0 to $n - 1$: if $a[i] = v$ then $res \leftarrow i$;
 return res

See file [lin_search.mlw](#)

Home Work 4: Binary Search

```
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)

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

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

Home Work: “for” loops

Propose a Hoare logic rule for the for loop:

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

Propose a rule for computing the WP:

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