

Switch to a ML-style programming language  
Functions and Function calls  
Proving Termination  
More on Specification Languages and Application to  
Arrays

Claude Marché

Cours MPRI 2-36-1 "Preuve de Programme"

17 décembre 2019

# Reminder of the last lecture

- ▶ Logics and automated prover capabilities
  - ▶ propositional logic
  - ▶ first-order logic
  - ▶ theories
    - ▶ equality
    - ▶ integer arithmetic
- ▶ classical Hoare logic
  - ▶ very simple programming language
  - ▶ deduction rules for triples  $\{Pre\}s\{Post\}$
- ▶ weakest liberal pre-conditions
  - ▶ function  $WLP(s, Q)$  returning a logic formula
  - ▶ soundness: if  $P \rightarrow WLP(s, Q)$  then triple  $\{P\}s\{Q\}$  is valid

## 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_euclide.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

## Exercise 3

Let's assume given in the underlying logic the functions  $\text{div2}(x)$  and  $\text{mod2}(x)$  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);
```

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

- ▶ 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

# This Lecture's Goals

- ▶ Switch to a “modern” ML-style language
- ▶ Extend that language:
  - ▶ Labels for reasoning on the past
  - ▶ Local mutable variables
  - ▶ Sub-programs, *function calls*, *modular reasoning*
- ▶ Proving *Termination*
- ▶ (First-order) logic as a *modeling language*
  - ▶ Definitions of new types, product types
  - ▶ Definitions of functions, of predicates
  - ▶ Axiomatizations
  - ▶ *Ghost code, ghost variables, ghost functions*
  - ▶ Help provers using *lemma functions*
- ▶ Application:
  - ▶ a bit of higher-order logic
  - ▶ program on *Arrays*

# Outline

“Modern” Approach, Blocking Semantics

A ML-like Programming Language

Blocking Operational Semantics

Weakest Preconditions Revisited

Syntax extensions

Termination, Variants

Advanced Modeling of Programs

Programs on Arrays

# Beyond IMP and classical Hoare Logic

Extended language

- ▶ more data types
- ▶ *logic variables*: local and **immutable**
- ▶ *labels* in specifications

Handle termination issues:

- ▶ prove properties on non-terminating programs
- ▶ prove termination when wanted

Prepare for adding later:

- ▶ run-time errors (how to prove their absence)
- ▶ local **mutable** variables, functions
- ▶ complex data types

## Extended Syntax: Generalities

- ▶ We want a few basic data types : int, bool, real, unit
- ▶ No difference between expressions and statements anymore

Basically we consider

- ▶ A purely functional language (ML-like)
- ▶ with *global mutable variables*

very restricted notion of modification of program states



# Base Data Types, Operators, Terms

- ▶ unit type: type `unit`, only one constant `()`
- ▶ Booleans: type `bool`, constants `True`, `False`, operators `and`, `or`, `not`
- ▶ integers: type `int`, operators `+`, `-`, `×` (no division)
- ▶ reals: type `real`, operators `+`, `-`, `×` (no division)
- ▶ Comparisons of integers or reals, returning a boolean
- ▶ “if-expression”: written `if b then t1 else t2`

<code>t ::= val</code>	(values, i.e. constants)
<code>v</code>	(logic variables)
<code>x</code>	(program variables)
<code>t op t</code>	(binary operations)
<code>if t then t else t</code>	(if-expression)

## Local logic variables

We extend the syntax of terms by

$$t ::= \text{let } v = t \text{ in } t$$

Example: approximated cosine

```
let cos_x =  
  let y = x*x in  
  1.0 - 0.5 * y + 0.04166666 * y * y  
in  
...
```

## Practical Notes

- ▶ Theorem provers (inc. Alt-Ergo, CVC4, Z3) typically support such a typed logic
- ▶ may also support if-expressions and let bindings

Alternatively, Why3 manages to transform terms and formulas when needed (e.g. transformation of if-expressions and/or let-expressions into equivalent formulas)

# Syntax: Formulas

Unchanged w.r.t to previous syntax, but also addition of local binding:

$p ::= t$	(boolean term)
$p \wedge p \mid p \vee p \mid \neg p \mid p \rightarrow p$	(connectives)
$\forall v : \tau, p \mid \exists v : \tau, p$	(quantification)
$\text{let } v = t \text{ in } p$	(local binding)

# Typing

- ▶ Types:

$$\tau ::= \text{int} \mid \text{real} \mid \text{bool} \mid \text{unit}$$

- ▶ Typing judgment:

$$\Gamma \vdash t : \tau$$

where  $\Gamma$  maps identifiers to types:

- ▶ either  $v : \tau$  (logic variable, immutable)
- ▶ either  $x : \text{mut } \tau$  (program variable, mutable)

## Important

- ▶ a mutable variable is not a value (it is not a “reference” value)
- ▶ as such, there is no “reference on a reference”
- ▶ no *aliasing*

# Typing rules

Constants:

$$\overline{\Gamma \vdash n : \text{int}}$$

$$\overline{\Gamma \vdash r : \text{real}}$$

$$\overline{\Gamma \vdash \text{True} : \text{bool}}$$

$$\overline{\Gamma \vdash \text{False} : \text{bool}}$$

Variables:

$$\frac{v : \tau \in \Gamma}{\Gamma \vdash v : \tau}$$

$$\frac{x : \text{mut } \tau \in \Gamma}{\Gamma \vdash x : \tau}$$

Let binding:

$$\frac{\Gamma \vdash t_1 : \tau_1 \quad \{v : \tau_1\} \cdot \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash \text{let } v = t_1 \text{ in } t_2 : \tau_2}$$

- ▶ **All terms have a base type** (not a “reference”)
- ▶ In practice: Why3, unlike OCaml, does not require to write !x for mutable variables

# Formal Semantics: Terms and Formulas

Program states are augmented with a stack of local (immutable) variables

- ▶  $\Sigma$ : maps program variables to values (a map)
- ▶  $\Pi$ : maps logic variables to values (a stack)

$$\begin{aligned} \llbracket val \rrbracket_{\Sigma, \Pi} &= val && \text{(values)} \\ \llbracket x \rrbracket_{\Sigma, \Pi} &= \Sigma(x) && \text{if } x : \text{mut } \tau \\ \llbracket v \rrbracket_{\Sigma, \Pi} &= \Pi(v) && \text{if } v : \tau \\ \llbracket t_1 \text{ op } t_2 \rrbracket_{\Sigma, \Pi} &= \llbracket t_1 \rrbracket_{\Sigma, \Pi} \llbracket op \rrbracket \llbracket t_2 \rrbracket_{\Sigma, \Pi} \\ \llbracket \text{let } v = t_1 \text{ in } t_2 \rrbracket_{\Sigma, \Pi} &= \llbracket t_2 \rrbracket_{\Sigma, (\{v = \llbracket t_1 \rrbracket_{\Sigma, \Pi} \} \cdot \Pi)} \end{aligned}$$

## Warning

Semantics is a partial function, it is not defined on ill-typed formulas

# Type Soundness Property

Our logic language satisfies the following standard property of purely functional language

## Theorem (Type soundness)

*Every well-typed terms and well-typed formulas have a semantics*

Proof: induction on the derivation tree of well-typing



## Expressions: generalities

- ▶ Former statements of IMP are now expressions of type `unit`  
**Expressions may have Side Effects**
- ▶ Statement `skip` is identified with `()`
- ▶ The sequence is replaced by a local binding
- ▶ From now on, the condition of the `if then else` and the `while do` in programs is a Boolean expression

# Syntax

$e ::= t$	(pure term)
$e \text{ op } e$	(binary operation)
$x \leftarrow e$	(assignment)
$\text{let } v = e \text{ in } e$	(local binding)
$\text{if } e \text{ then } e \text{ else } e$	(conditional)
$\text{while } e \text{ do } e$	(loop)

- ▶ sequence  $e_1; e_2$  : syntactic sugar for

$\text{let } v = e_1 \text{ in } e_2$

when  $e_1$  has type `unit` and  $v$  not used in  $e_2$

# Toy Examples

```
z <- if x ≥ y then x else y
```

```
let v = r in (r <- v + 42; v)
```

```
while (x <- x - 1; x > 0) (* (--x > 0) in C *)  
do ()
```

```
while (let v = x in x <- x - 1; v > 0) (* (x-- > 0) in C *)  
do ()
```

# Typing Rules for Expressions

Assignment:

$$\frac{x : \text{mut } \tau \in \Gamma \quad \Gamma \vdash e : \tau}{\Gamma \vdash x \leftarrow e : \text{unit}}$$

Let binding:

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \{v : \tau_1\} \cdot \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \text{let } v = e_1 \text{ in } e_2 : \tau_2}$$

Conditional:

$$\frac{\Gamma \vdash c : \text{bool} \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \text{if } c \text{ then } e_1 \text{ else } e_2 : \tau}$$

Loop:

$$\frac{\Gamma \vdash c : \text{bool} \quad \Gamma \vdash e : \text{unit}}{\Gamma \vdash \text{while } c \text{ do } e : \text{unit}}$$

# Operational Semantics

## Novelty w.r.t. IMP

Need to precise the order of evaluation: left to right  
(e.g.  $x \leftarrow 0; ((x \leftarrow 1); 2) + x = 2$  or  $3$  ?)

- ▶ one-step execution has the form

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

$\Pi$  is the *stack of local variables*

- ▶ values do not reduce

# Operational Semantics

## ► Assignment

$$\frac{\Sigma, \Pi, e \rightsquigarrow \Sigma', \Pi', e'}{\Sigma, \Pi, x \leftarrow e \rightsquigarrow \Sigma', \Pi', x \leftarrow e'}$$

$$\overline{\Sigma, \Pi, x \leftarrow val \rightsquigarrow \Sigma[x \leftarrow val], \Pi, ()}$$

## ► Let binding

$$\frac{\Sigma, \Pi, e_1 \rightsquigarrow \Sigma', \Pi', e'_1}{\Sigma, \Pi, \text{let } v = e_1 \text{ in } e_2 \rightsquigarrow \Sigma', \Pi', \text{let } v = e'_1 \text{ in } e_2}$$

$$\overline{\Sigma, \Pi, \text{let } v = val \text{ in } e \rightsquigarrow \Sigma, \{v = val\} \cdot \Pi, e}$$

# Operational Semantics, Continued

► Binary operations

$$\frac{\Sigma, \Pi, e_1 \rightsquigarrow \Sigma', \Pi', e'_1}{\Sigma, \Pi, e_1 + e_2 \rightsquigarrow \Sigma', \Pi', e'_1 + e_2}$$

$$\frac{\Sigma, \Pi, e_2 \rightsquigarrow \Sigma', \Pi', e'_2}{\Sigma, \Pi, val_1 + e_2 \rightsquigarrow \Sigma', \Pi', val_1 + e'_2}$$

$$\frac{val = val_1 + val_2}{\Sigma, \Pi, val_1 + val_2 \rightsquigarrow \Sigma, \Pi, val}$$

# Operational Semantics, Continued

## ► Conditional

$$\frac{\Sigma, \Pi, c \rightsquigarrow \Sigma', \Pi', c'}{\Sigma, \Pi, \text{if } c \text{ then } e_1 \text{ else } e_2 \rightsquigarrow \Sigma', \Pi', \text{if } c' \text{ then } e_1 \text{ else } e_2}$$

$$\frac{}{\Sigma, \Pi, \text{if } \textit{True} \text{ then } e_1 \text{ else } e_2 \rightsquigarrow \Sigma, \Pi, e_1}$$

$$\frac{}{\Sigma, \Pi, \text{if } \textit{False} \text{ then } e_1 \text{ else } e_2 \rightsquigarrow \Sigma, \Pi, e_2}$$

## ► Loop

$$\frac{}{\Sigma, \Pi, \text{while } c \text{ do } e \rightsquigarrow \Sigma, \Pi, \text{if } c \text{ then } (e; \text{while } c \text{ do } e) \text{ else } ()}$$



## Context Rules versus Let Binding

Remark: most of the context rules can be avoided

- ▶ An equivalent operational semantics can be defined using `let v = ... in ...` instead, e.g.:

$$\frac{v_1, v_2 \text{ fresh}}{\Sigma, \Pi, e_1 + e_2 \rightsquigarrow \Sigma, \Pi, \text{let } v_1 = e_1 \text{ in let } v_2 = e_2 \text{ in } v_1 + v_2}$$

- ▶ Thus, only the context rule for `let` is needed

# Type Soundness

## Theorem

*Every well-typed expression evaluate to a value or execute infinitely*

Classical proof:

- ▶ type is preserved by reduction
- ▶ execution of well-typed expressions that are not values can progress

# Blocking Semantics: General Ideas

- ▶ add *assertions* in expressions
- ▶ failed assertions = “run-time errors”

First step: modify expression syntax with

- ▶ new expression: assertion
- ▶ adding loop invariant in loops

```

$$e ::= \text{assert } p \quad (\text{assertion})$$

$$| \text{while } e \text{ invariant } I \text{ do } e \quad (\text{annotated loop})$$

```

## Toy Examples

```
z <- if x ≥ y then x else y ;  
assert (z ≥ x ∧ z ≥ y)
```

```
while (x <- x - 1; x > 0)  (* (--x > 0) in C *)  
  invariant x ≥ 0 do ();  
assert (x = 0)
```

```
while (let v = x in x <- x - 1; v > 0)  (* (x-- > 0) in C *)  
  invariant x ≥ -1 do ();  
assert (x < 0)
```

## Result value in post-conditions

New addition in the specification language:

- ▶ keyword **result** in post-conditions
- ▶ denotes the value of the expression executed

Example:

```
{ true }  
if  $x \geq y$  then  $x$  else  $y$   
{ result  $\geq x \wedge$  result  $\geq y$  }
```

## Blocking Semantics: Modified Rules

$$\frac{\llbracket P \rrbracket_{\Sigma, \Pi} \text{ holds}}{\Sigma, \Pi, \text{assert } P \rightsquigarrow \Sigma, \Pi, ()}$$

$$\frac{\llbracket I \rrbracket_{\Sigma, \Pi} \text{ holds}}{\Sigma, \Pi, \text{while } c \text{ invariant } I \text{ do } e \rightsquigarrow \Sigma, \Pi, \text{if } c \text{ then } (e; \text{while } c \text{ invariant } I \text{ do } e) \text{ else } ()}$$

### Important

Execution blocks as soon as an invalid annotation is met

# Soundness of a program

## Definition

Execution of an expression in a given state is *safe* if it does not block: either terminates on a value or runs infinitely.

## Definition

A triple  $\{P\}e\{Q\}$  is valid if for any state  $\Sigma, \Pi$  satisfying  $P$ ,  $e$  *executes safely* in  $\Sigma, \Pi$ , and if it terminates, the final state satisfies  $Q$

# Weakest Preconditions Revisited

Goal:

- ▶ construct a new calculus  $WP(e, Q)$

Expected property: in any state satisfying  $WP(e, Q)$ ,

- ▶  $e$  is guaranteed to execute safely
- ▶ if it terminates,  $Q$  holds in the final state



# New Weakest Precondition Calculus

Pure expressions (i.e. without side-effects, a.k.a. “terms”)

$$WP(t, Q) = Q[result \leftarrow t]$$

‘let’ binding

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

Reminder: sequence is a particular case of ‘let’

$$WP((e_1; e_2), Q) = WP(e_1, WP(e_2, Q))$$

## Weakest Preconditions, continued

- ▶ Assignment:

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

- ▶ Alternative:

$$\begin{aligned}\text{WP}(x \leftarrow e, Q) &= \text{WP}(\textit{let } v = e \textit{ in } x \leftarrow v, Q) \\ \text{WP}(x \leftarrow t, Q) &= Q[\textit{result} \leftarrow (); x \leftarrow t]\end{aligned}$$

## WP: Exercise

$WP(\text{let } v = x \text{ in } (x \leftarrow x + 1; v), x > \textit{result}) = ?$

## WP: Exercise

$WP(\text{let } v = x \text{ in } (x \leftarrow x + 1; v), x > \textit{result}) = ?$

$WP(\underline{\text{let } v = x \text{ in } (x \leftarrow x + 1; v)}, x > \textit{result})$

## WP: Exercise

$WP(\text{let } v = x \text{ in } (x \leftarrow x + 1; v), x > \textit{result}) = ?$

$$\begin{aligned} & WP(\text{let } v = x \text{ in } (x \leftarrow x + 1; v), x > \textit{result}) \\ = & \text{WP}(x, (\text{WP}(\underline{(x \leftarrow x + 1; v)}, x > \textit{result})[v \leftarrow \textit{result}]))) \end{aligned}$$

## WP: Exercise

$WP(\text{let } v = x \text{ in } (x \leftarrow x + 1; v), x > \text{result}) = ?$

$$\begin{aligned} & WP(\text{let } v = x \text{ in } (x \leftarrow x + 1; v), x > \text{result}) \\ = & WP(x, (WP(\underline{(x \leftarrow x + 1; v)}, x > \text{result})[v \leftarrow \text{result}])) \\ = & WP(x, (WP(x \leftarrow x + 1, WP(\underline{v}, x > \text{result}))))[v \leftarrow \text{result}]) \end{aligned}$$

## WP: Exercise

$WP(\text{let } v = x \text{ in } (x \leftarrow x + 1; v), x > \text{result}) = ?$

$$\begin{aligned} & WP(\text{let } v = x \text{ in } (x \leftarrow x + 1; v), x > \text{result}) \\ = & WP(x, (WP(\underline{(x \leftarrow x + 1; v)}, x > \text{result})[v \leftarrow \text{result}])) \\ = & WP(x, (WP(x \leftarrow x + 1, WP(\underline{v}, x > \text{result}))))[v \leftarrow \text{result}]) \\ = & WP(x, (WP(\underline{x \leftarrow x + 1}, x > v)))[v \leftarrow \text{result}]) \end{aligned}$$

## WP: Exercise

$WP(\text{let } v = x \text{ in } (x \leftarrow x + 1; v), x > \text{result}) = ?$

$$\begin{aligned} & WP(\text{let } v = x \text{ in } (x \leftarrow x + 1; v), x > \text{result}) \\ = & WP(x, (WP(\underline{(x \leftarrow x + 1; v)}, x > \text{result})[v \leftarrow \text{result}])) \\ = & WP(x, (WP(x \leftarrow x + 1, WP(\underline{v}, x > \text{result}))) [v \leftarrow \text{result}])) \\ = & WP(x, (WP(\underline{x \leftarrow x + 1}, x > v)) [v \leftarrow \text{result}])) \\ = & WP(x, \underline{(x + 1 > v)} [v \leftarrow \text{result}])) \end{aligned}$$



## WP: Exercise

$WP(\text{let } v = x \text{ in } (x \leftarrow x + 1; v), x > \text{result}) = ?$

$$\begin{aligned} & WP(\text{let } v = x \text{ in } (x \leftarrow x + 1; v), x > \text{result}) \\ = & WP(x, (WP(\underline{(x \leftarrow x + 1; v)}, x > \text{result})[v \leftarrow \text{result}]))) \\ = & WP(x, (WP(x \leftarrow x + 1, WP(\underline{v}, x > \text{result}))) [v \leftarrow \text{result}])) \\ = & WP(x, (WP(\underline{x \leftarrow x + 1}, x > v)) [v \leftarrow \text{result}])) \\ = & WP(x, \underline{(x + 1 > v)} [v \leftarrow \text{result}])) \\ = & \underline{WP(x, (x + 1 > \text{result}))} \end{aligned}$$

## WP: Exercise

$WP(\text{let } v = x \text{ in } (x \leftarrow x + 1; v), x > \text{result}) = ?$

$$\begin{aligned} & WP(\text{let } v = x \text{ in } (x \leftarrow x + 1; v), x > \text{result}) \\ = & WP(x, (WP(\underline{(x \leftarrow x + 1; v)}, x > \text{result})[v \leftarrow \text{result}])) \\ = & WP(x, (WP(x \leftarrow x + 1, WP(\underline{v}, x > \text{result}))) [v \leftarrow \text{result}])) \\ = & WP(x, (WP(\underline{x \leftarrow x + 1}, x > v)) [v \leftarrow \text{result}])) \\ = & WP(x, \underline{(x + 1 > v)} [v \leftarrow \text{result}])) \\ = & \underline{WP(x, (x + 1 > \text{result}))} \\ = & x + 1 > x \end{aligned}$$

## Weakest Preconditions, continued

- ▶ Conditional

$$\text{WP}(\text{if } e_1 \text{ then } e_2 \text{ else } e_3, Q) = \\ \text{WP}(e_1, \text{if } \textit{result} \text{ then } \text{WP}(e_2, Q) \text{ else } \text{WP}(e_3, Q))$$

- ▶ Alternative with let: (exercise!)

## Weakest Preconditions, continued

- ▶ Assertion

$$\begin{aligned}\text{WP}(\text{assert } P, Q) &= P \wedge Q \\ &= P \wedge (P \rightarrow Q)\end{aligned}$$

(second version useful in practice)

- ▶ While loop

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

where  $w_1, \dots, w_k$  is the set of assigned variables in expressions  $c$  and  $e$  and  $v_1, \dots, v_k$  are fresh logic variables

## Soundness of WP

### Lemma (Preservation by Reduction)

*If  $\Sigma, \Pi \models \text{WP}(e, Q)$  and  $\Sigma, \Pi, e \rightsquigarrow \Sigma', \Pi', e'$  then  $\Sigma', \Pi' \models \text{WP}(e', Q)$*

Proof: predicate induction of  $\rightsquigarrow$ .

### Lemma (Progress)

*If  $\Sigma, \Pi \models \text{WP}(e, Q)$  and  $e$  is not a value then there exists  $\Sigma', \Pi, e'$  such that  $\Sigma, \Pi, e \rightsquigarrow \Sigma', \Pi', e'$*

Proof: structural induction of  $e$ .

### Corollary (Soundness)

*If  $\Sigma, \Pi \models \text{WP}(e, Q)$  then*

- ▶  *$e$  executes safely in  $\Sigma, \Pi$ .*
- ▶ *if execution terminates,  $Q$  holds in the final state*

# Outline

“Modern” Approach, Blocking Semantics

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:

```
{  $y \geq 0$  }  
L:  
while y > 0 do  
  invariant {  $x + y = x@L + y@L$  }  
  x <- x + 1; y <- y - 1  
{  $x = x@old + y@old \wedge y = 0$  }
```

# 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 in post-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

Euclide's algorithm:

```
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 [gcd\\_euclid\\_labels.mlw](#)

# Mutable Local Variables

We extend the syntax of expressions with

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

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

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

$\Pi$  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

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

$\Pi$  no longer contains just immutable variables

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

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

$x$  local variable

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

And labels too

## 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* | *fundecl*

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



# Functions

Program structure:

*prog* ::= *decl*\*

*decl* ::= *vardecl* | *fundecl*

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

*fundecl* ::= **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  $\geq$  0
  ensures result  $\geq$  0  $\wedge$ 
           sqr(result)  $\leq$  x < sqr(result + 1)
body
  let ref res = 0 in
  let ref sum = 1 in
  while sum  $\leq$  x do
    res <- res + 1;
    sum <- sum + 2 * res + 1
  done;
  res
```

## Example using *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  $\Gamma$  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:

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

Note: the  $t_j$  are immutable expressions

# Operational Semantics

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

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

## Blocking Semantics

Execution blocks at call if pre-condition does not hold

# Operational Semantics of Function Call

frame is a dummy expression that keeps track of the *local variables* of the callee:

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

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

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

## 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  $Body$

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

## 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 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  $f(x_1 : \tau_1, \dots, x_n : \tau_n) : \tau$   
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  $Q$  and any expression  $e$ ,  
if  $\Sigma, \Pi \models WP(e, Q)$  then execution of  $\Sigma, \Pi, e$  is *safe*

Remark: (mutually) recursive functions are allowed

# Outline

“Modern” Approach, Blocking Semantics

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:

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

# 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

$\llbracket I \rrbracket_{\Sigma, \Pi}$  holds

---

$\Sigma, \Pi, \text{while } c \text{ invariant } / \text{ variant } t, \prec \text{ do } e \rightsquigarrow$   
     $\Sigma, \Pi, L : \text{if } c$   
        then ( $e; \text{assert } t \prec t@L;$   
            while  $c$  invariant / variant  $t, \prec$  do  $e$ )  
        else ()

# Weakest Precondition

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

## 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  
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*

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:

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

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

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

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

“Modern” Approach, Blocking Semantics

Syntax extensions

Termination, Variants

**Advanced Modeling of Programs**

(First-Order) Logic as a Modeling Language

Ghost code

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
- ▶ *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  $\tau_i, \tau$  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 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
```

# Introducing Ghost Code

Example: Euclidean division / just compute the remainder:

```
q <- 0; r <- x;  
while r ≥ y do  
  invariant { x = q * y + r }  
  r <- r - y; q <- q + 1
```

# Introducing Ghost Code

Example: Euclidean division / just compute the remainder:

```
    r <- x;  
while r ≥ y do  
  invariant { exists q. x = q * y + r }  
  r <- r - y;
```

# Introducing Ghost Code

Example: Euclidean division / just compute the remainder:

```
q <- 0 ; r <- x;  
while r ≥ y do  
  invariant { x = q * y + r }  
  r <- r - y; q <- q + 1
```



# Introducing Ghost Code

Example: Euclidean division / just compute the remainder:

```
q <- 0 ; r <- x;  
while r ≥ y do  
  invariant { x = q * y + r }  
  r <- r - y; q <- q + 1
```

## Ghost code, ghost variables

- ▶ Cannot interfere with regular code (checked by typing)
- ▶ Visible only in annotations

(See Why3 file [euclid\\_rem.mlw](#))

## Home Work 2

- ▶ Extend the post-condition of Euclid's algorithm for GCD to express the Bézout property:

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

- ▶ Prove the program by adding appropriate ghost local variables

Use canvas file [exo\\_bezout.mlw](#)

# 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*

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.  $y \neq 0 \rightarrow \text{div}(x,y) * y = x$ 
```

# 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.  $n \geq 1 \rightarrow \text{fact}(n) = n * \text{fact}(n-1)$ 
```

# 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



# More Ghosts: Programs turned into Logic Functions

If the program  $f$  is

- ▶ *Proved terminating*
- ▶ *Has no side effects*

let  $f(x_1 : \tau_1, \dots, x_n : \tau_n) : \tau$   
requires  $Pre$   
variant  $var, \prec$   
ensures  $Post$   
body  $Body$

then there exists a logic function:

function  $f : \tau_1 \dots \tau_n : \tau$

lemma  $f_{spec} : \forall x_1, \dots, x_n. Pre \rightarrow Post[result \leftarrow f(x_1, \dots, x_n)]$

and if  $Body$  is a pure term then

lemma  $f_{body} : \forall x_1, \dots, x_n. Pre \rightarrow f(x_1, \dots, x_n) = Body$

Offers an important alternative to axiomatic definitions

In Why3: done using keywords `let function`

## Example: axiom-free specification of factorial

```
let function fact (n:int) : int
  requires { n ≥ 0 }
  variant { n }
= if n=0 then 1 else n * fact(n-1)
```

generates the logic context

```
function fact int : int

axiom f_body: forall n. n ≥ 0 →
  fact n = if n=0 then 1 else n * fact(n-1)
```

## Axiomatic Definitions: Example of Factorial

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

```
let 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](#)

## More Ghosts: Lemma functions

- ▶ if a program function is *without side effects* and *terminating*:

let  $f(x_1 : \tau_1, \dots, x_n : \tau_n) : \text{unit}$

requires  $Pre$

variant  $var, \prec$

ensures  $Post$

body  $Body$

then it is a proof of

$$\forall x_1, \dots, x_n. Pre \rightarrow Post$$

- ▶ If  $f$  is recursive, it simulates a proof by induction

## Example: sum of odds

```
function sum_of_odd_numbers int : int
(** 'sum_of_odd_numbers n' denote the sum of
    odd numbers from '1' to '2n-1' **)

axiom sum_of_odd_numbers_base : sum_of_odd_numbers 0 = 0

axiom sum_of_odd_numbers_rec : forall n. n ≥ 1 →
    sum_of_odd_numbers n = sum_of_odd_numbers (n-1) + 2*n-1

goal sum_of_odd_numbers_any:
    forall n. n ≥ 0 → sum_of_odd_numbers n = n * n
```

See file [arith\\_lemma\\_function.mlw](#)

## Example: sum of odds as lemma function

```
let rec lemma sum_of_odd_numbers_any (n:int)
  requires { n ≥ 0 }
  variant { n }
  ensures { sum_of_odd_numbers n = n * n }
= if n > 0 then sum_of_odd_numbers_any (n-1)
```

## Home work 3

Prove the helper lemmas stated for the fast exponentiation algorithm

## Home Work 4

Prove Fermat's little theorem for case  $p = 3$ :

$$\forall x, \exists y. x^3 - x = 3y$$

using a lemma function



# Outline

“Modern” Approach, Blocking Semantics

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 :  $\tau$  -> 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 :  $\alpha$ )  $\rightarrow$  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`  $\rightarrow \alpha$
- ▶ 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

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

## 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

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

## Simple Example

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

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  $\alpha$  = { length : int; elts : int  $\rightarrow$   $\alpha$ }
```

```
val get (ref a:array  $\alpha$ ) (i:int) :  $\alpha$   
  requires  $0 \leq i < a.length$   
  ensures result = select(a.elts,i)
```

```
val set (ref a:array  $\alpha$ ) (i:int) (v: $\alpha$ ) : unit  
  requires  $0 \leq i < a.length$   
  writes a  
  ensures a.length = a@old.length  $\wedge$   
           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  $0 \leq i < a.length \wedge 0 \leq j < a.length$ 
  writes a
  ensures select(a.elts,i) = select(a@old.elts,j)  $\wedge$ 
           select(a.elts,j) = select(a@old.elts,i)  $\wedge$ 
           forall k:int.  $0 \leq k < a.length \wedge k \neq i \wedge k \neq j \rightarrow$ 
             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 using WP
- ▶ Prove the program

```
let test()  
  requires  
    select(a,0) = 13  $\wedge$  select(a,1) = 42  $\wedge$   
    select(a,2) = 64  
  ensures  
    select(a,0) = 64  $\wedge$  select(a,1) = 42  $\wedge$   
    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

```
var a: array real

let search(n:int, v:real): int
  requires  $0 \leq n$ 
  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:

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

That's all for today, Merry Christmas !



- ▶ Several home work exercises to do
- ▶ Project text on the web page soon, and announced by e-mail