Exercise 1

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

```plaintext
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

```plaintext
q <- 0; r <- x;
while r ≥ y do
  r <- r - y; q <- q + 1
```

(Why3 file to fill in: imp_euclid.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 invariant and prove the correctness of the program

Exercise 3

Let's assume given in the underlying logic the functions div2(x) and mod2(x) which respectively return the division of $x$ by 2 and its remainder. The following program is supposed to compute, in variable $r$, the power $x^n$.

```plaintext
r <- 1; p <- x; e <- n;
while e > 0 do
  if mod2(e) ≠ 0 then r <- r * p;
  p <- p * p;
  e <- div2(e);
```

- Assuming that the power function exists in the logic, specify appropriate pre- and post-conditions for this program
- Find an appropriate loop invariant, and prove the program
Reminder of the last lecture

▶ First-order logic and automated prover capabilities
▶ Classical Hoare Logic
  ▶ Very simple programming language
  ▶ Deduction rules for triples \( \{Pre\} P \{Post\} \)
▶ Modern programming language, ML-like
  ▶ more data types: int, bool, real, unit
  ▶ logic variables: local and immutable
  ▶ statement = expression of type unit
  ▶ Typing rules
  ▶ Formal operational semantics (small steps)
  ▶ type soundness: every typed program executes without blocking.
▶ Blocking semantics and Weakest Preconditions:
  ▶ \( e \) executes safely in \( \Sigma, \Pi \) if it does not block on an assertion or a loop invariant
  ▶ If \( \Sigma, \Pi \vDash WP(e, Q) \) then \( e \) executes safely in \( \Sigma, \Pi \), and if it terminates then \( Q \) valid in the final state.

This Lecture’s Goals

▶ Extend the language:
  ▶ Labels
  ▶ Local mutable variables
  ▶ Sub-programs, 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

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

\[
\{ \text{true} \} \\
\text{let } v = r \text{ in } (r \leftarrow v + 42; \; v) \\
\{ r = r@old + 42 \wedge \text{result} = r@old \}
\]

\[
\{ \text{true} \} \\
\text{let } \text{tmp} = x \text{ in } x \leftarrow y; \; y \leftarrow \text{tmp} \\
\{ x = y@old \wedge y = x@old \}
\]

SUM revisited:

\[
\begin{align*}
\{ y \geq 0 \} \\
\text{L:} \\
\text{while } y > 0 \text{ do} \\
\quad \text{invariant } \{ x + y = x@L + y@L \} \\
\quad x \leftarrow x + 1; \; y \leftarrow y - 1 \\
\quad \{ x = x@old + y@old \wedge y = 0 \}
\end{align*}
\]
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

Implicit labels:

- *Here*, available in every formula
- *Old*, available in post-conditions

New rules for WP

New rules for computing WP:

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

Exercise:

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

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

\[
\begin{align*}
\Sigma, \Pi, x < - \text{val} \rightarrow \Sigma((x, \text{Here}) \leftarrow \text{val}), \Pi, () \\
\Sigma, \Pi, L : e \rightarrow \Sigma((x, L) \leftarrow \Sigma(x, \text{Here}) \mid x \text{ any variable}), \Pi, e
\end{align*}
\]

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 \( x@L + y@K + 2 + x@\text{Here} \)

Example: computation of the GCD

Euclid’s algorithm:

\[
\begin{align*}
\text{requires} \{ x \geq 0 \land y \geq 0 \} \\
\text{ensures} \{ \text{result} = \gcd(x@Old, y@Old) \} \\
= L: \\
\text{while} \ y > 0 \ \text{do} \\
\quad \text{invariant} \{ x \geq 0 \land y \geq 0 \} \\
\quad \text{invariant} \{ \gcd(x, y) = \gcd(x@L, y@L) \} \\
\quad \text{let} \ r = \text{mod} \ x \ y \ \text{in} \ x \leftarrow y; y \leftarrow r \\
\quad \text{done}
\end{align*}
\]

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

We extend the syntax of expressions with

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

Example: isqrt revisited

```plaintext
val ref x : int
val ref res : int
res <- 0;
let ref sum = 1 in
while sum \leq x do
res <- res + 1; sum <- sum + 2 * res + 1
done
```

Mutable Local Variables: WP rules

Rules are exactly the same as for global variables

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

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

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

Operational Semantics

\[ \Sigma, \Pi, e \rightsquigarrow \Sigma', \Pi', e' \]

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

\[ \Sigma, \Pi, e_1 \rightsquigarrow \Sigma', \Pi', e_1' \]

\[ \Sigma, \Pi, \text{let ref } x = e_1 \text{ in } e_2 \rightsquigarrow \Sigma', \Pi', \text{let ref } x = e_1' \text{ in } e_2 \]

\[ \Sigma, \Pi, \text{let ref } x = v \text{ in } e \rightsquigarrow \Sigma', \Pi'\{(x, \text{Here}) \leftarrow v\}, e \]

\( x \) local variable

\[ \Sigma, \Pi, x < v \rightsquigarrow \Sigma, \Pi\{(x, \text{Here}) \leftarrow v\}, e \]

And labels too

Functions

Program structure:

\[ \text{prog ::= decl}^* \]

\[ \text{decl ::= vardecl | fundecl} \]

\[ \text{vardecl ::= val ref } id : \text{basetype} \]

\[ \text{fundecl ::= let } id((\text{param},)^*) : \text{basetype} \text{ contract body } e \]

\[ \text{param ::= id : \text{basetype} \text{ contract ::= requires } t \text{ writes } (id)^* \text{ ensures } t} \]

Function definition:

- Contract:
  - pre-condition
  - post-condition (label \textit{Old} available)
  - assigned variables: clause \textit{writes}

- Body: expression
Example: `isqrt`

```plaintext
let isqrt(x:int): int
  requires x ≥ 0
  ensures result ≥ 0 ∧
  \text{sqr(result)} ≤ x < \text{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

```plaintext
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 : τ_1, ..., x_n : τ_n) : τ
  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 Pre, Post : \text{formula}
\Gamma' \vdash Body : \tau
\vec{w}_g \subseteq \vec{w} \text{ for each call } g \quad 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 : τ_1, ..., x_n : τ_n) : τ
  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: the \( t_i \) are immutable expressions
Operational Semantics

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

\[
\Pi' = \{ x_i \mapsto \vec{[t]_i} \}_{\Sigma, \Pi} \quad \Sigma, \Pi' \models \text{Pre}
\]

\[
\Sigma, \Pi, f(t_1, \ldots, t_n) \rightsquigarrow \Sigma, \Pi, (\text{Old}: \text{frame}(\Pi', \text{Body}, \text{Post}))
\]

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 \( \text{Pre} \)
writes \( \vec{w} \)
ensures \( \text{Post} \)
body \( \text{Body} \)

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

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

Operational Semantics of Function Call

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

\[
\Sigma, \Pi, e \rightsquigarrow \Sigma', \Pi', e'
\]

\[
\Sigma, \Pi''(\text{frame}(\Pi, e, P)) \rightsquigarrow \Sigma', \Pi', (\text{frame}(\Pi', e', P))
\]

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

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

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

Example: isqrt(42)

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

\[
\text{val isqrt}(:\text{int}) : \text{int} \\
\quad \text{requires } x \geq 0 \\
\quad \text{writes } (\text{nothing}) \\
\quad \text{ensures } \text{result} \geq 0 \land \\
\quad \text{sqr} \text{(result)} \leq x < \text{sqr} \text{(result + 1)}
\]

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: Incrementation

```plaintext
val res: ref int
val incr(x:int):unit
  writes res
  ensures res = res@old + x
```

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

Soundness Theorem for a Complete Program

Assuming that for each function defined as

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

we have

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

then for any formula \( Q \) and any expression \( e \),

\[ \Sigma, \Pi \models WP(e, Q) \]

then execution of \( \Sigma, \Pi, e \) is safe

Remark: (mutually) recursive functions are allowed

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 $\text{val}_1 \succ \text{val}_2 \succ \text{val}_3 \succ \cdots$
- A typical ordering on integers:
  \[ x \prec y \iff x < y \land 0 \leq y \]

Syntax

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

Example:
\[
\{ y \geq 0 \} \\
L:: \text{while } y > 0 \text{ do} \\
\quad \text{invariant } \{ x + y = x@L + y@L \} \\
\quad \text{variant } \{ y \} \\
\quad x \gets x + 1; y \gets y - 1 \\
\{ x = x@Old + y@Old \land y = 0 \}
\]

Operational semantics

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

Remark: in practice with Why3:
- presence of loop variants tells if one wants to prove termination or not
- warning issued if no variant given
- keyword *diverges* in contract for non-terminating functions
- default ordering determined from type of $t$
Examples

Exercise: find adequate variants

```plaintext
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

```plaintext
let f(X1 : τ1, ..., Xn : τn) : τ
  requires Pre
  variant var, <
  writes w
  ensures Post
  body Body

WP for function call:

\[
WP(f(t_1, ..., t_n), Q) = Pre[x_i ← t_i] \land var[x_i ← t_i] < var@Init \land \\
\forall \vec{y}. (Post[x_i ← t_i][w_j ← y_j][w_j@Old ← w_j] → Q[w_j ← y_j])
\]

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 <

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

\[ v_g[\vec{y} ← \vec{t}] < v_f@Init \]

and symmetrically when \( g \) calls \( f \)

Home Work 1: McCarthy’s 91 Function

Find adequate specifications

```plaintext
f91(n) = 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
  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
- ...

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 \( \tau, \tau \) 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 ∧ 
  ∀ 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

Example: Euclidean division / just compute the remainder:

r <- r - y;
Introducing Ghost Code

Example: Euclidean division / just compute the remainder:

\[
\begin{aligned}
q &\gets 0; \\
r &\gets x; \\
\text{while } r \geq y \text{ do} \\
&\quad \text{invariant } \{ x = q \times y + r \} \\
&\quad r \gets r - y; \\
&\quad q \gets q + 1
\end{aligned}
\]

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:

\[
\exists a, b, \text{result} = x \times a + y \times b
\]

▶ Prove the program by adding appropriate ghost local variables

Use canvas file `exo_bezout.mlw`

Axiomatic Definitions

Function and predicate declarations of the form

\[
\begin{aligned}
\text{function } f(\tau, \ldots, \tau_n) : \tau \\
\text{predicate } p(\tau, \ldots, \tau_n)
\end{aligned}
\]

together with axioms

\[
\text{axiom id : formula}
\]

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)
```

Underspecified Logic Functions and Run-time Errors

Error “Division by zero” can be modeled by an abstract function

```plaintext
val div_real(x: real, y: real): real
  requires y ≠ 0.0
  ensures result = div(x, y)
```

Reminder

Execution blocks when an invalid annotation is met

Axiomatic Definitions

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

Warning about soundness

Axioms may introduce inconsistencies

```plaintext
function div(real, real): real
axiom mul_div:
  forall x, y. div(x, y) * y = x
implies 1 = div(1, 0) * 0 = 0
```

More Ghosts: Programs turned into Logic Functions

If the program \( f \) is

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

then there exists a logic function:

```plaintext
let function f (x_1 : \tau_1, ..., x_n : \tau_n) : \tau
  requires Pre
  variant \text{var}, <
  ensures Post
  body \text{Body}

\text{Pre} \rightarrow \text{Body}
```

and if \text{Body} is a pure term then

```plaintext
lemma f_body : \forall x_1, ..., x_n. Pre \rightarrow f(x_1, ..., x_n) = \text{Body}
```

Offers an important alternative to axiomatic definitions

In Why3: done using keywords \textit{let function}
Example: axiom-free specification of factorial

```plaintext
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

```plaintext
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:

```plaintext
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:
  ```plaintext
  let f(x_1: \tau_1, ..., x_n: \tau_n): unit
  requires Pre
  variant var, \prec
  ensures Post
  body Body
  then it is a proof of
  \forall x_1, ..., x_n. Pre → Post
  ```

- If \( f \) is recursive, it simulates a proof by induction

Example: sum of odds

```plaintext
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

```ocaml
let rec lemma sum_of_odd_numbers_any (n:int) avoids { 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

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: $\text{fun } x : \tau \rightarrow t$

Definition of selection function:

\[
\text{function select } (f : \alpha \rightarrow \beta) \ (x : \alpha) : \beta = f \ x
\]

Definition of function update:

\[
\text{function store } (f : \alpha \rightarrow \beta) \ (x : \alpha) \ (v : \beta) : \alpha \rightarrow \beta = \\
\text{fun } (y : \alpha) \rightarrow \text{if } x = y \text{ then } v \text{ else } f \ y
\]

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

- Lemma select_store_eq: $\forall f : \alpha \rightarrow \beta, x : \alpha, v : \beta$. $\text{select}(\text{store}(f, x, v), x) = v$
- Lemma select_store_neq: $\forall f : \alpha \rightarrow \beta, x, y : \alpha, v : \beta$. $x \neq y \rightarrow \text{select}(\text{store}(f, x, v), y) = \text{select}(f, j)$

Arrays as Mutable References on Maps

- Array variable: mutable variable of type $\text{int} \rightarrow \alpha$
- In a program, the standard assignment operation $a[i] \leftarrow e$
  is interpreted as
  $a \leftarrow \text{store}(a, i, e)$

Simple Example

```haskell
val ref a: int \rightarrow 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 $a$:

```haskell
val ref a: int \rightarrow 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 \neq i ∧ k \neq j \rightarrow
          select(a,k) = select(a@Old,k)
body
  let tmp = select(a,i) in (* tmp <-a[i]*)
  a <- store(a,i,select(a,j)); (* a[i]<-a[j]*)
  a <- store(a,j,tmp) (* a[j]<-tmp *)
```
Arrays as Reference on pairs (length,map)

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

```
type array α = { length : int; elts : int → α }
val get (ref a:array α) (i:int) : α
  requires 0 ≤ i < a.length
  ensures result = select(a.elts,i)
val set (ref a:array α) (i:int) (v:α) : 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] }
body
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 ∧ select(a,1) = 42 ∧ select(a,2) = 64
ensures
  select(a,0) = 64 ∧ select(a,1) = 42 ∧ select(a,2) = 13
body
  swap(0,2)
```

- **Specify, implement, and prove a program that increments by 1 all cells, between given indexes `i` and `j`, of an array of reals**
Exercise: Search Algorithms

var a: array real

let

search(n:int, v:real): int

requires 0 ≤ n
ensures { ? }

1. Formalize postcondition: if v occurs in a, between 0 and n − 1, then result is an index where v occurs, otherwise result is set to −1

2. Implement and prove linear search:
   res ← −1;
   for each i from 0 to n − 1: if a[i] = v then res ← i;
   return res

See file lin_search.mlw

Home Work 4: Binary Search

low = 0; high = n − 1;
while low ≤ 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 = e₁ to e₂ do e

Typing:
► i visible only in e, and is immutable
► e₁ and e₂ must be of type int, e must be of type unit

Operational semantics:
(assuming e₁ and e₂ are values v₁ and v₂)

\[
\begin{align*}
\Sigma, \Pi, \text{for } i = v₁ \text{ to } v₂ \text{ do } e & \rightarrow \Sigma, \Pi, () \\
\end{align*}
\]

\[
\begin{align*}
\Sigma, \Pi, \text{for } i = v₁ \text{ to } v₂ \text{ do } e & \rightarrow \Sigma, \Pi, (\text{let } i = v₁ \text{ in } e); \\
\Sigma, \Pi, \text{for } i = v₁ + 1 \text{ to } v₂ \text{ do } e & \rightarrow \Sigma, \Pi, (\text{for } i = v₁ + 1 \text{ to } v₂ \text{ do } e)
\end{align*}
\]