Simple Syntax Extensions
(ghost variables, labels, local mutable variables)
Functions and Function calls
Proving Termination
Specification languages
Application to arrays

Claude Marché
Cours MPRI 2-36-1 “Preuve de Programme”
14 décembre 2016

Exercise 2

The following program is one of the original examples of Floyd.

\[ q := 0; \ r := x; \]
\[ \text{while } r \geq y \ do \]
\[ r := r - y; \ q := q + 1 \]

(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 invariant 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 \( x^n \).

\[ r := 1; \ p := x; \ e := n; \]
\[ \text{while } e > 0 \ do \]
\[ \quad \text{if } \text{mod2}(e) \neq 0 \ then \ r := r \times p; \]
\[ \quad p := p \times p; \]
\[ \quad e := \text{div2}(e); \]

▶ 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.

Exercise 4

The Fibonacci sequence is defined recursively by \( \text{fib}(0) = 0, \text{fib}(1) = 1 \) and \( \text{fib}(n + 2) = \text{fib}(n + 1) + \text{fib}(n) \). The following program is supposed to compute \( \text{fib} \) in linear time, the result being stored in \( y \).

\[ y := 0; \ x := 1; \ i := 0; \]
\[ \text{while } i < n \ do \]
\[ \quad \text{aux} := y; \ y := x; \ x := x + \text{aux}; \ i := i + 1 \]

▶ Assuming \( \text{fib} \) exists in the logic, specify appropriate pre- and post-conditions.

▶ Prove the program.
Exercise (Exam 2011-2012)

In this exercise, we consider the simple language of the first lecture of this course, where expressions do not have any side effect.

1. Prove that the triple

\[ \{ P \} x := e \{ \exists v, e[x \leftarrow v] = x \land P[x \leftarrow v] \} \]

is valid with respect to the operational semantics.

2. Show that the triple above can be proved using the rules of Hoare logic.

Let us assume that we replace the standard Hoare rule for assignment by the rule

\[ \{ P \} x := e \{ \exists v, e[x \leftarrow v] = x \land P[x \leftarrow v] \} \]

3. Show that the triple \( \{ P[x \leftarrow e] \} x := e \{ P \} \) can be proved with the new set of rules.

Reminder of the last lecture

- Classical Hoare Logic
  - Very simple programming language
  - Deduction rules for triples \{ Pre \} s \{ 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 \( \Sigma, \Pi \models \text{WP}(e, Q) \) 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:
  - Ghost variables and Labels
  - Local mutable variables
  - Sub-programs, modular reasoning
- Proving Termination
- (First-order) logic as modeling language
  - Automated provers capabilities
  - Towards complex data structures: Axiomatized types and predicates
  - Help provers via lemma functions
- Application: program on Arrays

Outline

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

Termination, Variants

Advanced Modeling of Programs

Programs on Arrays
Ghost variables

Example: Euclid’s algorithm, on two global variables x, y

Euclid:

\[
\begin{align*}
\text{requires} & \quad ? \\
\text{ensures} & \quad ? \\
& = \text{while } y > 0 \text{ do} \\
& \quad \text{let } r = \text{mod } x \text{ y} \text{ in } x := y; y := r \\
& \quad \text{done} \\
& \quad x
\end{align*}
\]

What should be the post-condition?

Ghost variables
additional variables, introduced for the specification

See Why3 file euclid_ghost.mlw

Labels: motivation

- Using ghost variables becomes quickly painful
- Label
  - simple alternative to ghost variables
  - (but not always possible)

Labels: Syntax and Typing

Add in syntax of terms:

\[
t ::= x@L \quad \text{(labeled variable access)}
\]

Add in syntax of expressions:

\[
e ::= L : e \quad \text{(labeled expressions)}
\]

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

Toy Examples, Continued

\[
\begin{align*}
\{ \text{true} \} \\
\text{let } v = r \text{ in } (r := v + 42; v) \\
\{ r = r@Old + 42 \land \text{result} = r@Old \}
\end{align*}
\]

\[
\begin{align*}
\{ \text{true} \} \\
\text{let } \text{tmp} = x \text{ in } x := y; y := \text{tmp} \\
\{ x = y@Old \land y = x@Old \}
\end{align*}
\]

SUM revisited:

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

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

The operational semantics of expressions is modified as follows

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

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: Euclid's algorithm revisited

Euclid:

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

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

See file euclid_labels.mlw

New rules for WP

New rules for computing WP:

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

Exercise:

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

Mutable Local Variables

We extend the syntax of expressions with

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

Example: isqrt revisited

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

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

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

\[ \Sigma, \Pi, e_1 \leadsto \Sigma', \Pi', e'_1 \]

\[ \Sigma, \Pi, \text{let ref } x = e_2 \leadsto \Sigma, \Pi\{(x, \text{Here}) \mapsto v\}, e \]

\( x \) local variable

\[ \Sigma, \Pi, x := v \leadsto \Sigma, \Pi\{(x, \text{Here}) \mapsto v\}, e \]

And labels too.

Mutable Local Variables: WP rules

Rules are exactly the same as for global variables

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

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

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

Functions

Program structure:

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

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

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

\[ \text{fundecl} ::= \text{let fun id(\ (param,)^\ ) :basetype} \]

\[ \text{contract body e} \]

\[ \text{param} ::= \text{id : basetype} \]

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

Function definition:

\[ \text{Contract:} \]

\[ \text{pre-condition,} \]

\[ \text{post-condition (label \textit{Old} available),} \]

\[ \text{assigned variables: clause \textit{writes}.} \]

\[ \text{Body: expression.} \]

Home Work 1

- Extend the post-condition of Euclid’s algorithm to express the Bézout property:
  \[ \exists a, b, \text{result} = x \ast a + y \ast b \]

- Prove the program by adding appropriate ghost local variables

Use canvas file \texttt{exo_bezout.mlw}
Example: $\text{isqrt}$

```plaintext
let fun isqrt(x:int): int
  requires x ≥ 0
  ensures result ≥ 0 ∧
  \(\text{sqr}(\text{result}) ≤ x < \text{sqr}(\text{result} + 1)\)

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 $\text{Old}$ label

```plaintext
val res: ref int

let fun incr(x:int)
  requires true
  writes res
  ensures res = res@Old + x

body
  res := res + x
```

Typing

Definition $d$ of function $f$:

```
let fun f(x_1 : τ_1,...,x_n : τ_n) : τ
  requires Pre
  writes \(\vec{w}\)
  ensures Post
  body Body
```

Well-formed definitions:

\[
\begin{align*}
\Gamma' &= \{x_i : \tau_i | 1 \leq i \leq n\} \cdot \Gamma \\
\vec{w} &\subseteq \Gamma \\
\Gamma' \vdash \text{Pre, } \text{Post : formula} \\
\Gamma' \vdash \text{Body : } \tau \\
\end{align*}
\]

\[
\begin{align*}
\vec{w}_g \subseteq \vec{w} \quad \text{for each call } g \\
y \in \vec{w} \quad \text{for each assign } y
\end{align*}
\]

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

where $\Gamma$ contains the global declarations. Well-typed function calls:

\[
\Gamma \vdash t_i : \tau_i
\]

\[
\Gamma \vdash f(t_1,\ldots,t_n) : \tau
\]

Note: $t_i$ are immutable expressions.

Operational Semantics

```
function f(x_1 : τ_1,...,x_n : τ_n) : τ
  requires Pre
  writes \(\vec{w}\)
  ensures Post
  body Body
```

```
\begin{align*}
\Pi' &= \{x_i \mapsto [t_i]_{\Sigma,\Pi}\} \\
\Sigma,\Pi' &\vdash \text{Pre} \\
\Sigma,\Pi, f(t_1,\ldots,t_n) &\Rightarrow \Sigma,\Pi, (\text{Old : frame}(\Pi', \text{Body}, \text{Post}))
\end{align*}
```

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:

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

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

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

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

WP Rule of Function Call

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

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

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

Example: isqrt(42)

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

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

Example: Incrementation

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

```
val res: ref int
val incr(x:int):unit
  writes res
  ensures res = res@Old + x
```
Soundness Theorem for a Complete Program

Assuming that for each function defined as

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

we have

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

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

if \( \Sigma, \Pi \models \text{WP} (e, Q) \) 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 \( \text{val}_1 \succ \text{val}_2 \succ \text{val}_3 \cdots \))
- A typical ordering on integers:

\[
x \prec y \equiv x < y \land 0 \leq y
\]
Syntax

New syntax construct:

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

Example:

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

Operational semantics

\[ \Sigma, \Pi, \text{while } c \text{ invariant } l \text{ variant } t, \prec \text{ do } e \mapsto \Sigma, \Pi, L : \text{if } c \]
\[ \text{then } (e; \text{assert } t \prec t@L) \]
\[ \text{else } () \]

Weakest Precondition

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

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 \textit{diverges} in contract for non-terminating functions
- default ordering determined from type of \( t \)

Examples

Exercise: find adequate variants.

\begin{verbatim}
 i := 0;
 while i \leq 100
 invariant ? variant ?
do i := i+1 done;

 while sum \leq x
 invariant ? variant ?
do
 res := res + 1; sum := sum + 2 * res + 1
done;
\end{verbatim}
Recursive Functions: Termination

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

```
let fun f(x_1:τ_1,...,x_n:τ_n):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]∧ var[x_i ← t_i]≺ var@Init ∧
∀⃗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.

Home Work 2: McCarthy’s 91 Function

```
f91(n) = if n ≤ 100 then f91(f91(n + 11)) else n - 10
```

Find adequate specifications.

```
let fun 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
  Axiomatic Type Definitions
  Automated Provers Capabilities, Lemma Functions

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

▶ (mostly First-order) logic, with type polymorphism à la ML
▶ Built-in arithmetic (integers and reals)
▶ Definitions à la ML
▶ logic (i.e. pure) functions, predicates
▶ structured types, pattern-matching
▶ Axiomatizations
▶ Inductive predicates
▶ Some higher-order features: lambda-expressions are allowed with syntax \x: \tau. t

Important note
Function and predicates are always totally defined

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_1, \ldots, \tau_n are not reference types.
▶ No recursion allowed
▶ No side effects
▶ Defines total functions and predicates

Logic Symbols: Examples

\text{function} \quad \text{sqr}(x:\text{int}) = x * x

\text{predicate} \quad \text{prime}(x:\text{int}) =
\quad x \geq 2 \land
\quad \forall y z : \text{int}. \ y \geq 0 \land z \geq 0 \land x = y * z \rightarrow
\quad y = 1 \lor z = 1
Axiomatic Definitions

Function and predicate declarations of the form

\[
\text{function } f(\tau_1, \ldots, \tau_n) : \tau \\
\text{predicate } p(\tau_1, \ldots, \tau_n)
\]

together with axioms

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

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

Example: division

\[
\text{function } \text{div}(\text{real}, \text{real}) : \text{real} \\
\text{axiom } \text{mul_div} : \\
\quad \forall x, y. y \neq 0 \implies \text{div}(x, y) \cdot y = x
\]

Example: factorial

\[
\text{function } \text{fact}(\text{int}) : \text{int} \\
\text{axiom } \text{fact0} : \\
\quad \text{fact}(0) = 1 \\
\text{axiom } \text{factn} : \\
\quad \forall n : \text{int}. n \geq 1 \implies \text{fact}(n) = n \cdot \text{fact}(n-1)
\]

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.

\[
\text{function } \text{div}(\text{real}, \text{real}) : \text{real} \\
\text{axiom } \text{mul_div} : \forall x, y. \text{div}(x, y) \cdot y = x \\
\implies 1 = \text{div}(1,0) \cdot 0 = 0
\]

Underspecified Logic Functions and Run-time Errors

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

\[
\text{val } \text{div}_\text{real}(x : \text{real}, y : \text{real}) : \text{real} \\
\text{requires } y \neq 0.0 \\
\text{ensures } \text{result} = \text{div}(x, y)
\]

Reminder

Execution blocks when an invalid annotation is met
Axiomatic Definitions: Example of Factorial

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

```ocaml
let fun 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

Axiomatic Type Definitions

Abstract type declarations, of the form

```
type τ
```

associated with axiomatized functions and predicates

Example: colors

```
type color
function blue: color
function red: color
axiom distinct: red ≠ blue
```

Polymorphic types:

```
type τ α₁ ⋯ αₖ
```

where \( α₁ ⋯ αₖ \) are type parameters.

Example: Sets

```
type set α
function empty: set α
function single(α): set α
function union(set α, set α): set α
axiom union_assoc: forall x y z: set α.
  union(union(x,y),z) = union(x,union(y,z))
axiom union_comm: forall x y: set α.
  union(x,y) = union(y,x)
predicate mem(α, set α)
axiom mem_empty: forall x: α. ¬ mem(x, empty)
axiom mem_single: forall x y: α.
  mem(x,single(y)) ↔ x=y
axiom mem_union: forall x: α, y z: set α.
  mem(x, union(y,z)) ↔ mem(x,y) ∨ mem(x,z)
```

Automated Provers Capabilities

SMT solvers like Alt-Ergo, CVC4, Z3 are the best ones for deductive verification because:

- they understand (typed) first-order logic
- they have built-in support for the equality predicate
- they support integer and real arithmetic
- they allow user definitions and axiomatizations

Weaknesses:

- incompleteness (this logic is too powerful to be decidable)
- weak support for quantifiers (sometimes FO provers like Vampire, Spass, E can be better)
- existential goals are typically hard: provers cannot guess the “witness”
- no support for advanced reasoning like induction
Some hints to help provers

- Simplify the goal: inline definitions, compute what can be computed
- Split the goal into subgoals (hint: try to inline definition of the head symbol of the goal)
- help the provers by
  - introduce extra assertions in the code ("local lemmas")
  - introduce extra lemmas before the code
  - prove extra lemmas using lemma functions

Lemma functions

- Basic idea: if a program function is without side effects and terminating:
  \[
  \text{let fun } f(x_1:\tau_1,\ldots,x_n:\tau_n):\tau \\
  \text{ requires } Pre \\
  \text{ variant } var, < \\
  \text{ ensures } Post \\
  \text{ body } Body \\
  \text{ then it is a (constructive) proof of }
  \forall x_1,\ldots,x_n. \exists result. Pre \Rightarrow Post
  \]
- If $f$ is recursive, it simulates a proof by induction

Example: power function

```ml
function power int int : int
axiom power_0 : forall x:int. power x 0 = 1
axiom power_s : forall x n:int. n >= 0 ->
    power x (n+1) = x * power x n
lemma power_1 : forall x:int. power x 1 = x
lemma sqrt4_256 : exists x:int. power x 4 = 256
lemma power_sum : forall x n m: int. 0 <= n & 0 <= m ->
    power x (n+m) = power x n * power x m
```

See file lemma_functions.mlw

Home Work 3

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

\[
\forall x, \exists y. x^3 - x = 3y
\]

using a lemma function
Arrays as References on Pure Maps

Axiomatization of $\text{maps}$ from $\text{int}$ to some type $\alpha$:

- **type** $\text{map} \ \alpha$
- **function** $\text{select} (\text{map} \ \alpha, \ \text{int}) : \ \alpha$
- **function** $\text{store} (\text{map} \ \alpha, \ \text{int}, \ \alpha) : \ \text{map} \ \alpha$
- **axiom** $\text{select\_store\_eq}$:
  
  $\forall a : \text{map} \ \alpha, \ i : \text{int}, \ v : \alpha.
  \text{select}(\text{store}(a, i, v), i) = v$

- **axiom** $\text{select\_store\_neq}$:
  
  $\forall a : \text{map} \ \alpha, \ i, j : \text{int}, \ v : \alpha.
  \ i \neq j \rightarrow \text{select}(\text{store}(a, i, v), j) = \text{select}(a, j)$

- Unbounded indexes.
- $\text{select}(a, i)$ models the usual notation $a[i]$.
- $\text{store}$ denotes the *functional update* of a map.

---

Arrays as Reference on Maps

- Array variable: variable of type $\text{ref} (\text{map} \ \alpha)$.
- In a program, the standard assignment operation $a[i] := e$ is interpreted as $a := \text{store}(a, i, e)$

Simple Example

```plaintext
val a: ref (map int)

let fun 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 \(a\):

```why3
val a: ref (map int)

let fun swap(i:int,j:int)
  requires 0 ≤ i < length a ∧ 0 ≤ j < length a
  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 *)
```

Example: Swap again

```why3
val a: ref (int,map int)

let fun swap(i:int,j:int)
  requires 0 ≤ i < fst a ∧ 0 ≤ j < fst a
  writes a
  ensures select(snd a,i) = select(snd a@Old,j) ∧
  select(snd a,j) = select(snd a@Old,i) ∧
  forall k:int. k ≠ i ∧ k ≠ j →
  select(snd a,k) = select(snd a@Old,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 *)
```

Arrays as Reference on pairs (length,map)

- Goal: model "out-of-bounds" run-time errors
- Array variable: mutable variable of type \((\text{int},\text{map }\alpha)\).

```why3
val get(a:array \alpha,i:int):\alpha
  requires 0 ≤ i < fst(a)
  ensures result = select(snd(a),i)

val set(a:array \alpha,i:int,\alpha):unit
  requires 0 ≤ i < fst(a)
  writes a
  ensures fst(a) = fst(a@Old) ∧
  snd(a) = store(snd(a@Old),i,\alpha)

▷ a[i] interpreted as a call to get(a,i)
▷ a[i] := \alpha interpreted as a call to set(a,i,\alpha)
```

Note about Arrays in Why3

```why3
use import array.Array
syntax: a.length,a[i],a[i]<-\alpha

Example: swap

```why3
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
```

Example: Swap again

```why3
val a: array int

let fun 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 (* tmp :=a[i]*)
  set(a,i,get(a,j)); (* a[i]:=a[j]*)
  set(a,j,tmp) (* a[j]:=tmp *)
```
**Exercises on Arrays**

- Prove Swap using WP.
- Prove the program

```ml
let fun 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.

---

**Home Work 4: Binary Search**

\[
\begin{align*}
low &= 0; \quad high = n - 1; \\
\text{while } low \leq high: \\
\text{let } m \text{ be the middle of } low \text{ and } high \\
\text{if } a[m] = v \text{ then return } m \\
\text{if } a[m] < v \text{ then continue search between } m \text{ and } high \\
\text{if } a[m] > v \text{ then continue search between } low \text{ and } m
\end{align*}
\]

See file `bin_search.mlw`

---

**Exercise: Search Algorithms**

```ml
var a: array real
let fun 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:
   ```ml
   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 5: “for” loops**

Syntax: \( \text{for } i = e_1 \text{ to } e_2 \text{ do } e \)  

Typing:
- \( i \) visible only in \( e \), and is immutable  
- \( e_1 \) and \( e_2 \) must be of type \( \text{int} \), \( e \) must be of type \( \text{unit} \)

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

\[
\begin{align*}
\Sigma, \Pi, \text{for } i = v_1 \text{ to } v_2 \text{ do } e & \Rightarrow \Sigma, \Pi, ()
\end{align*}
\]

\[
\begin{align*}
\Sigma, \Pi, \text{for } i = v_1 \text{ to } v_2 \text{ do } e & \Rightarrow \Sigma, \Pi, \left( \text{let } i = v_1 \text{ in } e \right)
\end{align*}
\]

\[
\begin{align*}
\Sigma, \Pi, \text{for } i = v_1 \text{ to } v_2 \text{ do } e & \Rightarrow \Sigma, \Pi, \left( \text{for } i = v_1 + 1 \text{ to } v_2 \text{ do } e \right)
\end{align*}
\]
Propose a Hoare logic rule for the for loop:

\[
\begin{align*}
\{ ? \} & e \{ ? \} \\
\{ ? \} & \text{for } i = v_1 \text{ to } v_2 \text{ do } e \{ ? \}
\end{align*}
\]

Propose a rule for computing the WP:

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

That's all for today, Merry Christmas!