Exercise 2

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

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

- 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 \(\text{div}2(x)\) and \(\text{mod}2(x)\) which respectively return the division of \(x\) by 2 and its remainder. The following program is supposed to compute, in variable \(r\), the power \(x^n\).

\[
r := 1; p := x; e := n; \\
\textbf{while } e > 0 \textbf{ do} \\
\quad \textbf{if } \text{mod}2(e) \neq 0 \textbf{ then } r := r \ast p; \\
\quad p := p \ast p; \\
\quad e := \text{div}2(e);
\]

- 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

- Classical Hoare Logic
  - Very simple programming language
  - Deduction rules for triples \(\{\text{Pre}\}s\{\text{Post}\}\)
  - WLP: if \(\text{Pre} \Rightarrow \text{WLP}(s, \text{Post})\) then \(\{\text{Pre}\}s\{\text{Post}\}\) valid
  - Use of Why3
- Modern programming language, ML-like
  - more data types: int, bool, real, unit
  - \textit{logic variables}: local and immutable
  - statement = expression of type unit
  - Typing rules
  - Formal operational semantics (small steps)
  - \textit{type soundness}: every typed program executes without blocking.
- \textit{Blocking semantics}:
  - Program safety defined by \textit{Blocking Semantics}
This Lecture’s Goals

▶ Blocking semantics continued:
  ▶ New WP calculus
  ▶ **WP soundness**: validity of the WP implies safety of execution
▶ 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
  ▶ Application: **Arrays**

Outline

Blocking Semantics continued, WP revisited

Syntax extensions

Termination, Variants

Advanced Modeling of Programs

Reminder: Syntax

\[ e ::= \begin{array}{l}
  t \quad \text{(pure term)} \\
  e \ op \ e \quad \text{(binary operation)} \\
  x := e \quad \text{(assignment)} \\
  \text{let } v = e \text{ in } e \quad \text{(local binding)} \\
  \text{if } e \text{ then } e \text{ else } e \quad \text{(conditional)} \\
  \text{assert } p \quad \text{(assertion)} \\
  \text{while } e \text{ invariant } I \text{ do } e \quad \text{(annotated loop)}
\end{array} \]

▶ sequence \( e_1 ; e_2 \) : syntactic sugar for

\[ \text{let } v = e_1 \text{ in } e_2 \]

when \( e_1 \) has type \( \text{unit} \) and \( v \) not used in \( e_2 \)
▶ Addition in the logic language: keyword **result** in post-conditions, denotes the value of the expression executed

Reminder: Operational Semantics

▶ one-step execution has the form

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

▶ values (i.e. constants) do not reduce
▶ failed assertions = “run-time errors”

Novelties

▶ Need for **context rules**
▶ Precise the order of evaluation: left to right
Blocking Semantics: Modified Rules

\[ [P]_{\Sigma, \Pi} \text{ holds} \]
\[ \Sigma, \Pi, \text{assert } P \iff \Sigma, \Pi, () \]

\[ [\_]_{\Sigma, \Pi} \text{ holds} \]
\[ \Sigma, \Pi, \text{while } c \text{ invariant } I \text{ do } e \rightarrow \]
\[ \Sigma, \Pi, \text{if } c \text{ then } (e; \text{while } c \text{ invariant } I \text{ do } e) \text{ else } () \]

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

Remark:
- Stating this for \( Q = true \) is enough to ensure safety
- But need to state this for any \( Q \) to prove soundness (by induction)

New Weakest Precondition Calculus

- Pure terms:
  \[ WP(t, Q) = Q[\text{result} \leftarrow t] \]

- Let binding:
  \[ WP(\text{let } x = e_1 \text{ in } e_2, Q) = WP(e_1, WP(e_2, Q)[x \leftarrow \text{result}]) \]
Weakest Preconditions, continued

▶ Assignment:

\[
WP(x := e, Q) = WP(e, Q[result \leftarrow (); x \leftarrow result])
\]

▶ Alternative:

\[
WP(x := e, Q) = WP(let v = e in x := v, Q)
\]

\[
WP(x := t, Q) = Q[result \leftarrow (); x \leftarrow t]]
\]

Weakest Preconditions, continued

▶ Conditional

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

▶ Alternative with let: (exercise!)

Weakest Preconditions, continued

▶ Assertion

\[
WP(\text{assert } P, Q) = P \land Q = P \land (P \Rightarrow Q)
\]

(second version useful in practice)

▶ While loop

\[
WP(\text{while } c \text{ invariant } I \text{ do } e, Q) =
I \land ^
\forall \vec{v}, (I \Rightarrow WP(c, if result then WP(e, I) else Q))[w_1 \leftarrow v_1]
\]

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

WP: Exercise

\[
WP(\text{let } v = x \text{ in } (x := x + 1; v), x > result) = ?
\]
Soundness of WP

**Lemma (Preservation by Reduction)**

If $\Sigma, \Pi \models WP(e, Q)$ and $\Sigma, \Pi, e \leadsto \Sigma', \Pi', e'$ then $\Sigma', \Pi' \models WP(e', Q)$

Proof: predicate induction of $\leadsto$.

**Lemma (Progress)**

If $\Sigma, \Pi \models WP(e, Q)$ and $e$ is not a value then there exists $\Sigma', \Pi, e'$ such that $\Sigma, \Pi, e \leadsto \Sigma', \Pi', e'$

Proof: structural induction of $e$.

**Corollary (Soundness)**

If $\Sigma, \Pi \models WP(e, Q)$ then $e$ executes safely in $\Sigma, \Pi$.

Outline

- Blocking Semantics continued, WP revisited
- Syntax extensions
  - Ghost variables and Labels
  - Local Mutable Variables
  - Functions
- Termination, Variants
- Advanced Modeling of Programs

Ghost variables

Example: Euclidean's algorithm, on two global variables $x, y$

Euclidean:

- **requires** ?
- **ensures** ?
- $= \text{while } y > 0 \text{ do}$
  - let $r = \text{mod } x \text{ y in } x := y; \ y := r$
  - done;
- $x$

What should be the post-condition?

**Labels: motivation**

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

Ghost variables

- additional variables, introduced for the specification

See Why3 file euclide_ghost.mlw
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

Toy Examples, Continued

{ true }
let \( v = r \) in \( r := v + 42; v \)
{ \( r \) = \( r@old + 42 \) \&\& \( \text{result} = r@old \) }

{ 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 \{ \( x + y = x@L + y@L \) \}
  \( x := x + 1; y := y - 1 \)
{ \( x = x@old + y@old \) \&\& \( y = 0 \) }

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:
\[ Jx \rightarrow K \Sigma, \Pi = \Sigma(x, \text{Here}) \]
\[ Jx@L \rightarrow K \Sigma, \Pi = \Sigma(x, L) \]

Syntactic sugar: term \( t@L \)
- attach label \( L \) to any variable of \( t \) that does not have an explicit label yet.
- example: \((x + y@K + 2)@L + x\) is \( x@L + y@K + 2 + x@\text{Here} \).

New rules for WP

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

Exercise:
\[ \text{WP}(L :: x := x + 42, x@\text{Here} > x@L) = ? \]
Example: Euclidean’s algorithm revisited

Euclidean:
  \( \text{requires } \{ x \geq 0 \land y \geq 0 \} \)
  \( \text{ensures } \{ \text{result} = \gcd(x@Old, y@Old) \} \)

= L:
  \( \text{while } y > 0 \text{ do} \)
  \( \text{invariant } \{ x \geq 0 \land y \geq 0 \} \)
  \( \text{invariant } \{ \gcd(x, y) = \gcd(x@L, y@L) \} \)
  \( \text{let } r = \text{mod } x \text{ y in} \ x := y; y := r \)
  \( \text{done}; \)
  \( x \)

See file euclidean_labels.ml

Mutable Local Variables

We extend the syntax of expressions with

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

Example: isqrt revisited

val x, res : ref int

isqrt:
  \( \text{res := 0; \let ref sum = 1 in} \)
  \( \text{while } \text{sum} \leq x \text{ do} \)
  \( \text{res := res + 1; sum := sum + 2 * res + 1} \)
  \( \text{done} \)

Operational Semantics

\[ \Sigma, \Pi, e \rightsquigarrow \Sigma', \Pi', e' \]
\( \Pi \) no longer contains just immutable variables.

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

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

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

And labels too.

Mutable Local Variables: WP rules

Rules are exactly the same as for global variables

\[ \text{WP(let ref } x = e_1 \text{ in } e_2, Q) = \text{WP(e}_1, \text{WP(e}_2, Q)[x \leftarrow \text{result}]) \]

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

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

- Extend the post-condition of Euclidean algorithm to express the Bezout property:
  \[ \exists a, b, \text{result} = x \cdot a + y \cdot b \]

- Prove the program by adding appropriate ghost local variables

Use canvas file `exo_bezout.mlw`

Example: `isqrt`

```ml
let fun isqrt(x:int): int
    requires x \geq 0
    ensures \text{result} \geq 0 \land \text{sqr(result)} \leq x < \text{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

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

Well-formed definitions:

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

$\Gamma \vdash d : \text{wf}$

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

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

WP Rule of Function Call

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

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

Modular proof

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.

```plaintext
val isqrt(x:int): int
    requires x ≥ 0
    writes (nothing)
    ensures result ≥ 0 ∧
        sqrt(result) ≤ x < sqrt(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

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

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

Soundness of WP

Assuming that for each function defined as

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

we have

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

then for any formulas \(P\) and \(Q\) and any expression \(e\), if \(\models P \Rightarrow WP(e, Q)\), then \(e\) executes safely in any state satisfying \(P\).

Outline

- Blocking Semantics continued, WP revisited
- Syntax extensions
- Termination, Variants
- Advanced Modeling of Programs
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 \(<\) (i.e. there is no infinite sequence \(val_1 > val_2 > val_3 > \cdots\))
▶ A typical ordering on integers:

\[
x < y \Leftrightarrow x < y \land 0 \leq y
\]

Syntax

New syntax construct:

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

Example:

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

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

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

\[
\text{variant } \{ y \}
\]

\[
x := x + 1; y := y - 1
\]

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

Operational semantics
Weakest Precondition

No distinction liberal/strict:
- presence of loop variants tells if one wants to prove termination or not

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

\[
[w_i \leftarrow v_i]
\]

Examples

Exercise: find adequate variants.

```
1 := 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;
```

Recursive Functions: Termination

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

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

WP for function call:

\[
\text{WP}(f(t_1, \ldots, t_n), Q) = Pre[x_i \leftarrow t_i] \land \text{var}[x_i \leftarrow t_i] \prec \text{var}@\text{Init} \land \\
\forall \vec{y}. (\text{Post}[x_i \leftarrow t_i][w_j \leftarrow y_j][w_j@\text{Old} \leftarrow w_j] \Rightarrow Q[w_j \leftarrow y_j])
\]

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 \(f\)
Example: McCarthy's 91 Function

\[ f_{91}(n) = \begin{cases} f_{91}(f_{91}(n + 11)) & \text{if } n \leq 100 \\ n - 10 & \text{else} \end{cases} \]

Exercise: find adequate specifications.

```
let fun f91(n:int): int
  requires ?
  variant ?
  writes ?
  ensures ?
body
  if n \leq 100 then f91(f91(n + 11)) else n - 10
```

Outline

- Blocking Semantics continued, WP revisited
- Syntax extensions
- Termination, Variants

Advanced Modeling of Programs
- (First-Order) Logic as a Modeling Language
- Axiomatic Definitions
- About Automated Provers Capabilities

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, 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
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 \(\tau_i, \tau\) are not reference types.

- No recursion allowed
- No side effects
- Defines total functions and predicates

Logic Symbols: Examples

\[
\begin{align*}
\text{function } & \text{sqr}(x \text{: int}) = x * x \\
\text{predicate } & \text{prime}(x \text{: int}) = x \geq 2 \land \\
& \forall y, z \text{: int}. \ y \geq 0 \land z \geq 0 \land x = y * z \rightarrow \\
& \ y = 1 \lor z = 1
\end{align*}
\]

Axiomatic Definitions

Function and predicate declarations of the form
\[
\begin{align*}
\text{function } & f(\tau, \ldots, \tau_n) : \tau \\
\text{predicate } & p(\tau, \ldots, \tau_n)
\end{align*}
\]
together with axioms
\[
\begin{align*}
\text{axiom } & \text{id} : \text{formula} \\
\text{specify that } & f \text{ (resp. } p) \text{ is any symbol satisfying the axioms.}
\end{align*}
\]

Axiomatic Definitions

Example: division
\[
\begin{align*}
\text{function } & \text{div}(\text{real}, \text{real}) : \text{real} \\
\text{axiom mul_div: } & \forall x, y. \ y \neq 0 \rightarrow \text{div}(x, y) * y = x
\end{align*}
\]

Example: factorial
\[
\begin{align*}
\text{function } & \text{fact}(\text{int}) : \text{int} \\
\text{axiom fact0: } & \text{fact}(0) = 1 \\
\text{axiom factn: } & \forall n \text{: int}. \ n \geq 1 \rightarrow \text{fact}(n) = n * \text{fact}(n-1)
\end{align*}
\]
Axiomatic Definitions

- Functions/predicates are typically **underspecified**. 
  - ⇒ model partial functions in a logic of total functions.
- About soundness: axioms may introduce **inconsistencies**.

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 annotations is met

Axiomatic Definitions: Example of Factorial

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

```plaintext
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
```

Automated Provers Capabilities

SMT solvers like Alt-Ergo, CVC, 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*:
  
  ```ml
  let fun f(x_1 : τ_1, ..., x_n : τ_n) : τ
    requires Pre
    variant var, <
    ensures Post
  body Body
  then it is a (constructive) proof of
  
  \forall x_1, ..., x_n. \exists result. Pre \Rightarrow Post
  ```

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

**Example: power function**

```
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*

**Exercise**

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

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

Using a lemma function