Blocking Semantics, Weakest Preconditions

First Extensions
(ghost variables, labels, function calls, termination, specification languages)

Claude Marché

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

18 décembre 2014

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
```

- 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.

```
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

- Classical Hoare Logic
  - Very simple programming language
  - Deduction rules for triples $\{Pre\}s\{Post\}$
  - WLP: if $Pre \Rightarrow \mathrm{WLP}(s, Post)$ then $\{Pre\}s\{Post\}$ valid
  - Use of Why3
- 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*:
  - Program safety defined by *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	$::=$	t	(pure term)
	$\mid$	$e\ op\ e$	(binary operation)
	$\mid$	$x := e$	(assignment)
	$\mid$	$\texttt{let}\ v = e\ \texttt{in}\ e$	(local binding)
	$\mid$	$\texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e$	(conditional)
	$\mid$	$\texttt{assert}\ p$	(*assertion*)
	$\mid$	$\texttt{while}\ e\ \texttt{invariant}\ I\ \texttt{do}\ e$	(*annotated* loop)

- sequence $e_1; e_2$: syntactic sugar for

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

  when e_1 has type `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

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

- 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

$$\frac{[\![P]\!]_{\Sigma,\Pi} \text{ holds}}{\Sigma, \Pi, \texttt{assert}\ P \rightsquigarrow \Sigma, \Pi, ()}$$

$$\frac{[\![I]\!]_{\Sigma,\Pi} \text{ holds}}{\begin{array}{l}\Sigma, \Pi, \texttt{while}\ c\ \texttt{invariant}\ I\ \texttt{do}\ e \rightsquigarrow \\ \quad \Sigma, \Pi, \texttt{if}\ c\ \texttt{then}\ (e; \texttt{while}\ c\ \texttt{invariant}\ I\ \texttt{do}\ e)\ \texttt{else}\ ()\end{array}}$$

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 Σ, Π satisfying P, e *executes safely* in Σ, Π, and if it terminates, the final state satisfies Q

Weakest Preconditions Revisited

Goal:

- construct a new calculus $\mathrm{WP}(e, Q)$
- expected property: in any state satisfying $\mathrm{WP}(e, Q)$, e is guaranteed to execute safely

Remark:

- Stating this for $Q = \mathit{true}$ is enough to ensure safety
- But need to state this for any Q to prove soundness (by induction)

New Weakest Precondition Calculus

- Pure terms:

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

- Let binding:

$$\begin{array}{l}\mathrm{WP}(\texttt{let}\ x = e_1\ \texttt{in}\ e_2, Q) = \\ \quad \mathrm{WP}(e_1, \mathrm{WP}(e_2, Q)[x \leftarrow \mathit{result}])\end{array}$$

Weakest Preconditions, continued

- Assignment:

$$\mathrm{WP}(x := e, Q) = \mathrm{WP}(e, Q[\mathit{result} \leftarrow (); x \leftarrow \mathit{result}])$$

- Alternative:

$$\begin{array}{rcl} \mathrm{WP}(x := e, Q) & = & \mathrm{WP}(\texttt{let}\ v = e\ \texttt{in}\ x := v, Q) \\ \mathrm{WP}(x := t, Q) & = & Q[\mathit{result} \leftarrow (); x \leftarrow t]) \end{array}$$

WP: Exercise

$$\mathrm{WP}(\texttt{let}\ v = x\ \texttt{in}\ (x := x + 1; v), x > \mathit{result}) = ?$$

Weakest Preconditions, continued

- Conditional

$$\begin{array}{l} \mathrm{WP}(\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3, Q) = \\ \quad \mathrm{WP}(e_1, \texttt{if}\ \mathit{result}\ \texttt{then}\ \mathrm{WP}(e_2, Q)\ \texttt{else}\ \mathrm{WP}(e_3, Q)) \end{array}$$

- Alternative with let: (exercise!)

Weakest Preconditions, continued

- Assertion

$$\begin{array}{rcl} \mathrm{WP}(\texttt{assert}\ P, Q) & = & P \wedge Q \\ & = & P \wedge (P \Rightarrow Q) \end{array}$$

(second version useful in practice)

- While loop

$$\begin{array}{l} \mathrm{WP}(\texttt{while}\ c\ \texttt{invariant}\ I\ \texttt{do}\ e, Q) = \\ \quad I \wedge \\ \quad \forall \vec{v}, (I \Rightarrow \mathrm{WP}(c, \texttt{if}\ \mathit{result}\ \texttt{then}\ \mathrm{WP}(e, I)\ \texttt{else}\ Q))[w_i \leftarrow v_i] \end{array}$$

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

Soundness of WP

Lemma (Preservation by Reduction)

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

Proof: predicate induction of $\rightsquigarrow$.

Lemma (Progress)

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

Proof: structural induction of e.

Corollary (Soundness)

If $\Sigma, \Pi \models \mathrm{WP}(e, Q)$ *then* e *executes safely in* Σ, Π.

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: Euclide's algorithm, on two global variables `x,y`

```
Euclide:
    requires  ?
    ensures   ?
  = while y > 0 do
      let r = mod x y in x := y; y := r
    done;
    x
```

What should be the post-condition?

Ghost variables

additional variables, introduced for the specification

See Why3 file `euclide_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*:

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

Add in syntax of *expressions*:

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

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

```
{ true }
let v = r in (r := v + 42; v)
{ r = r@Old + 42 ∧ result = r@Old }
```

```
{ true }
let tmp = x in x := y; y := tmp
{ x = y@Old ∧ y = x@Old }
```

SUM revisited:

```
{ y ≥ 0 }
L:
while y > 0 do
  invariant { x + y = x@L + y@L }
  x := x + 1; y := y - 1
{ x = x@Old + y@Old ∧ y = 0 }
```

Labels: 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:

$$\begin{array}{rcl} [\![x]\!]_{\Sigma,\Pi} & = & \Sigma(x, Here) \\ [\![x@L]\!]_{\Sigma,\Pi} & = & \Sigma(x, L) \end{array}$$

The operational semantics of expressions is modified as follows

$$\begin{array}{rcl} \Sigma, \Pi, x := val & \rightsquigarrow & \Sigma\{(x, Here) \leftarrow val\}, \Pi, () \\ \Sigma, \Pi, L : e & \rightsquigarrow & \Sigma\{(x, L) \leftarrow \Sigma(x, Here) \mid x \text{ any variable}\}, \Pi, e \end{array}$$

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:

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

Exercise:

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

Example: Euclide's algorithm revisited

```
Euclide:
    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 euclide_labels.mlw

Mutable Local Variables

We extend the syntax of expressions with

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

Example: isqrt revisited

```
val x, res : ref int

isqrt:
  res := 0;
  let ref sum = 1 in
  while sum ≤ x do
    res := res + 1; sum := sum + 2 * res + 1
  done
```

Operational Semantics

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

Π no longer contains just immutable variables.

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

$$\frac{}{\Sigma, \Pi, \texttt{let ref}\ x = v\ \texttt{in}\ e \rightsquigarrow \Sigma, \Pi\{(x, \mathit{Here}) \mapsto v\}, e}$$

$$\frac{x \text{ local variable}}{\Sigma, \Pi, x := v \rightsquigarrow \Sigma, \Pi\{(x, \mathit{Here}) \mapsto v\}, e}$$

And labels too.

Mutable Local Variables: WP rules

Rules are exactly the same as for global variables

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

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

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

Exercise

- Extend the post-condition of Euclide algorithm to express the Bezout property:

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

- Prove the program by adding appropriate ghost local variables

Use canvas file `exo_bezout.mlw`

Functions

Program structure:

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

Function definition:

- Contract:
  - pre-condition,
  - post-condition (label *Old* available),
  - assigned variables: clause writes .
- Body: expression.

Example: isqrt

```
let fun isqrt(x:int): int
  requires x ≥ 0
  ensures result ≥ 0 ∧
          sqr(result) ≤ x < sqr(result + 1)
body
  let ref res = 0 in
  let ref sum = 1 in
  while sum ≤ x do
    res := res + 1;
    sum := sum + 2 * res + 1
  done;
  res
```

Example using *Old* label

```
val 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:

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

where Γ contains the global declarations. Well-typed function calls:

$$\frac{\Gamma \vdash t_i : \tau_i}{\Gamma \vdash f(t_1, \ldots, t_n) : \tau}$$

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 $Body$

$$\frac{\Pi' = \{x_i \mapsto [\![t_i]\!]_{\Sigma,\Pi}\} \qquad \Sigma, \Pi' \models Pre}{\Sigma, \Pi, f(t_1, \ldots, t_n) \rightsquigarrow \Sigma, \Pi, (Old : \texttt{frame}(\Pi', Body, Post))}$$

Operational Semantics of Function Call

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

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

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

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

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 $Body$

$$\begin{array}{l} \mathrm{WP}(f(t_1, \ldots, t_n), Q) = Pre[x_i \leftarrow t_i] \wedge \\ \quad \forall \vec{v},\ (Post[x_i \leftarrow t_i, w_j \leftarrow v_j, w_j@Old \leftarrow w_j] \Rightarrow Q[w_j \leftarrow v_j]) \end{array}$$

Modular proof

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

Example: isqrt(42)

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

Assuming that for each function defined as

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

we have

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

then for any formulas P and Q and any expression e,
if $\models P \Rightarrow \mathrm{WP}(e, Q)$, 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* $\prec$ (i.e. there is no infinite sequence $val_1 \succ val_2 \succ val_3 \succ \cdots$
- A typical ordering on integers:

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

Syntax

New syntax construct:

$$e \quad ::= \quad \texttt{while}\ e\ \texttt{invariant}\ I\ \texttt{variant}\ t, \prec\ \texttt{do}\ e$$

Example:

```
{ y ≥ 0 }
L:
while y > 0 do
  invariant { x + y = x@L + y@L }
  variant { y }
  x := x + 1; y := y - 1
{ x = x@Old + y@Old ∧ y = 0 }
```

Operational semantics

$$\frac{[\![I]\!]_{\Sigma,\Pi} \text{ holds}}{\begin{array}{l}\Sigma, \Pi, \texttt{while}\ c\ \texttt{invariant}\ I\ \texttt{variant}\ t, \prec\ \texttt{do}\ e \rightsquigarrow \\ \quad \Sigma, \Pi, L : \texttt{if}\ c \\ \qquad \texttt{then}\ (e; \texttt{assert}\ t \prec t@L; \\ \qquad\quad \texttt{while}\ c\ \texttt{invariant}\ I\ \texttt{variant}\ t, \prec\ \texttt{do}\ e) \\ \qquad \texttt{else}\ ()\end{array}}$$

Weakest Precondition

No distinction liberal/strict:

- presence of loop variants tells if one wants to prove termination or not

$$\mathrm{WP}(\texttt{while } c \texttt{ invariant } I \texttt{ variant } t, \prec \texttt{ do } e, Q) =$$
$$I \wedge$$
$$\forall \vec{v}, (I \Rightarrow \mathrm{WP}(L : c, \texttt{if } \mathit{result} \texttt{ then } \mathrm{WP}(e, I \wedge t \prec t@L) \texttt{ else } Q))$$
$$[w_i \leftarrow v_i]$$

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.

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

WP for function call:

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

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

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

and symmetrically when g calls f

Example: McCarthy's 91 Function

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

Exercise: 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
```

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

function $f(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau = e$

Predicate defined as

predicate $p(x_1 : \tau_1, \ldots, x_n : \tau_n) = e$

where τ_i, τ are not reference types.

- No recursion allowed
- 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
```

Axiomatic Definitions

Function and predicate declarations of the form

function $f(\tau, \ldots, \tau_n) : \tau$
predicate $p(\tau, \ldots, \tau_n)$

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≠0 → div(x,y)*y = x
```

Example: factorial

```
function fact(int):int
axiom fact0:
  fact(0) = 1
axiom factn:
  forall n:int. n ≥ 1 → fact(n) = n * fact(n-1)
```

Axiomatic Definitions

- Functions/predicates are typically underspecified.
  $\Rightarrow$ 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

```
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:

```
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* :

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

  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

```
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`

Exercice

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

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

using a lemma function