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

Claude Marché
Cours MPRI 2-36-1 “Preuve de Programme”
17 décembre 2019

Reminder of the last lecture

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

Exercise 2

The following program is one of the original examples of Floyd

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

(Why3 file to fill in: imp_euclide.mlw)

▶ Propose a formal precondition to express that \(x\) is assumed non-negative, \(y\) is assumed positive, and a formal post-condition expressing that \(q\) and \(r\) are respectively the quotient and the remainder of the Euclidean division of \(x\) by \(y\)
▶ Find appropriate loop invariants and prove the correctness of the program

Exercise 3

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

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

(Why3 file to fill in: power_int.mlw)

▶ Assuming that the power function exists in the logic, specify appropriate pre- and post-conditions for this program
▶ Find an appropriate loop invariant, and prove the program
This Lecture’s Goals

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

Outline

“Modern” Approach, Blocking Semantics
A ML-like Programming Language
Blocking Operational Semantics
Weakest Preconditions Revisited

Syntax extensions
Termination, Variants
Advanced Modeling of Programs
Programs on Arrays

Beyond IMP and classical Hoare Logic

Extended language
- more data types
- logic variables: local and immutable
- labels in specifications

Handle termination issues:
- prove properties on non-terminating programs
- prove termination when wanted

Prepare for adding later:
- run-time errors (how to prove their absence)
- local mutable variables, functions
- complex data types

Extended Syntax: Generalities

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

Basically we consider
- A purely functional language (ML-like)
- with global mutable variables
  very restricted notion of modification of program states
Base Data Types, Operators, Terms

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

\[
\begin{align*}
t & ::= \text{val} & \text{(values, i.e. constants)} \\
& | \ v & \text{(logic variables)} \\
& | \ x & \text{(program variables)} \\
& | \ t \ op \ t & \text{(binary operations)} \\
& | \ \text{if} \ t \ \text{then} \ t \ \text{else} \ t & \text{(if-expression)}
\end{align*}
\]

Local logic variables

We extend the syntax of terms by

\[
t ::= \text{let} \ v = t \ \text{in} \ t
\]

Example: approximated cosine

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

Practical Notes

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

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

Syntax: Formulas

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

\[
p ::= \ t \quad \text{(boolean term)}
\ |
  p \land p | p \lor p | \neg p | \rightarrow p \quad \text{(connectives)}
\ |
  \forall v : \tau, p \mid \exists v : \tau, p \quad \text{(quantification)}
\ |
  \text{let} \ v = t \ \text{in} \ p \quad \text{(local binding)}
\]
Typing

- Types:
  \( \tau ::= \text{int} \mid \text{real} \mid \text{bool} \mid \text{unit} \)

- Typing judgment:
  \( \Gamma \vdash t : \tau \)
  where \( \Gamma \) maps identifiers to types:
  - either \( v : \tau \) (logic variable, immutable)
  - either \( x : \text{mut } \tau \) (program variable, mutable)

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

Formal Semantics: Terms and Formulas

Program states are augmented with a stack of local (immutable) variables
- \( \Sigma \): maps program variables to values (a map)
- \( \Pi \): maps logic variables to values (a stack)

\[
\begin{align*}
\llbracket \text{val} \rrbracket_{\Sigma, n} & = \text{val} \\
\llbracket x \rrbracket_{\Sigma, n} & = \Sigma(x) \\
\llbracket v \rrbracket_{\Sigma, n} & = \Pi(v) \\
\llbracket t_1 \ op \ t_2 \rrbracket_{\Sigma, n} & = \llbracket t_1 \rrbracket_{\Sigma, n} \ op \ \llbracket t_2 \rrbracket_{\Sigma, n} \\
\llbracket \text{let } v = t_1 \ in \ t_2 \rrbracket_{\Sigma, n} & = \llbracket t_2 \rrbracket_{\Sigma, \{v \mapsto \llbracket t_1 \rrbracket_{\Sigma, n} \}}
\end{align*}
\]

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

Type Soundness Property

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

**Theorem (Type soundness)**
Every well-typed terms and well-typed formulas have a semantics

**Proof:** induction on the derivation tree of well-typing
Expressions: generalities

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

Syntax

\[
\begin{align*}
e & ::= t \quad \text{(pure term)} \\
& | e \ op \ e \quad \text{(binary operation)} \\
& | x \ <- \ e \quad \text{(assignment)} \\
& | \text{let } v = e \ \text{in} \ e \quad \text{(local binding)} \\
& | \text{if } e \ \text{then} \ e \ \text{else} \ e \quad \text{(conditional)} \\
& | \text{while } e \ \text{do} \ e \quad \text{(loop)} \\
\end{align*}
\]

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

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

when \( e_1 \) has type `unit` and \( v \) not used in \( e_2 \)

Toy Examples

\[
z \gets \ \text{if } x \geq y \ \text{then } x \ \text{else} \ y
\]

\[
\text{let } v = r \ \text{in} \ (r \gets v + 42; v)
\]

\[
\text{while } (x \gets x - 1; x > 0) \ (\ast \ (\ast x > 0) \ \text{in } C \ \ast) \ \text{do } ()
\]

\[
\text{while } (\text{let } v = x \ \text{in} \ x \gets x - 1; v > 0) \ (\ast \ (\ast x > 0) \ \text{in } C \ \ast) \ \text{do } ()
\]

Typing Rules for Expressions

Assignment:

\[
\frac{x : \text{mut } \tau \in \Gamma}{\Gamma \vdash x \gets e : \tau}
\]

Let binding:

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

Conditional:

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

Loop:

\[
\frac{\Gamma \vdash c : \text{bool} \quad \Gamma \vdash e : \text{unit}}{\Gamma \vdash \text{while } c \ \text{do} \ e : \text{unit}}
\]
Operational Semantics

Novelty w.r.t. IMP
Need to precise the order of evaluation: left to right
(e.g. $x < 0; ((x < 1) ; 2 + x) = 2$ or 3 ?)

▶ one-step execution has the form
\[\Sigma, \Pi, e \rightsquigarrow \Sigma', \Pi', e'\]
\[\text{\Pi is the stack of local variables}\]
▶ values do not reduce

Operational Semantics, Continued

▶ Binary operations
\[\Sigma, \Pi, e_1 \rightsquigarrow \Sigma', \Pi', e'_1\]
\[\Sigma, \Pi, e_1 + e_2 \rightsquigarrow \Sigma', \Pi', e'_1 + e_2\]
\[\Sigma, \Pi, e_2 \rightsquigarrow \Sigma', \Pi', e'_2\]
\[\Sigma, \Pi, val_1 + e_2 \rightsquigarrow \Sigma', \Pi', val_1 + e'_2\]
\[\text{\text{val} = val_1 + val_2}\]
\[\Sigma, \Pi, val_1 + val_2 \rightsquigarrow \Sigma, \Pi, val\]

Operational Semantics

▶ Assignment
\[\Sigma, \Pi, x \leftarrow e \rightsquigarrow \Sigma', \Pi', x \leftarrow e'\]
\[\Sigma, \Pi, x \leftarrow \text{val} \rightsquigarrow \Sigma[x \leftarrow \text{val}], \Pi(), \text{\(\)}\]

▶ Let binding
\[\Sigma, \Pi, e_1 \rightsquigarrow \Sigma', \Pi', e'_1\]
\[\Sigma, \Pi, \text{let } v = e_1 \text{ in } e_2 \rightsquigarrow \Sigma', \Pi', \text{let } v = e'_1 \text{ in } e_2\]
\[\Sigma, \Pi, \text{let } v = \text{val} \text{ in } e \rightsquigarrow \Sigma, \{v = \text{val}\} \cdot \Pi, e\]

Operational Semantics, Continued

▶ Conditional
\[\Sigma, \Pi, c \rightsquigarrow \Sigma', \Pi', c'\]
\[\Sigma, \Pi, \text{if } c \text{ then } e_1 \text{ else } e_2 \rightsquigarrow \Sigma', \Pi', \text{if } c' \text{ then } e_1 \text{ else } e_2\]
\[\Sigma, \Pi, \text{if } \text{True} \text{ then } e_1 \text{ else } e_2 \rightsquigarrow \Sigma, \Pi, e_1\]
\[\Sigma, \Pi, \text{if } \text{False} \text{ then } e_1 \text{ else } e_2 \rightsquigarrow \Sigma, \Pi, e_2\]

▶ Loop
\[\Sigma, \Pi, \text{while } c \text{ do } e \rightsquigarrow\]
\[\Sigma, \Pi, \text{if } c \text{ then } (e; \text{while } c \text{ do } e) \text{ else } ()\]
Context Rules versus Let Binding

Remark: most of the context rules can be avoided

- An equivalent operational semantics can be defined using
  \[ \text{let } v = \ldots \text{ in } \ldots \] instead, e.g.:

  \[
  \sigma, \Pi, e_1 + e_2 \rightarrow \sigma, \Pi, \text{let } v_1 = e_1 \text{ in let } v_2 = e_2 \text{ in } v_1 + v_2
  \]

- Thus, only the context rule for let is needed

Type Soundness

Theorem

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

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

Blocking Semantics: General Ideas

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

First step: modify expression syntax with
- new expression: assertion
- adding loop invariant in loops

Toy Examples

\[
z \leftarrow \text{if } x \geq y \text{ then } x \text{ else } y; \\
\text{assert } (z \geq x \land z \geq y)
\]

\[
\text{while } (x \leftarrow x - 1; x > 0) \quad (* (\neg x > 0) \text{ in } C *) \\
\text{invariant } x \geq 0 \text{ do } (); \\
\text{assert } (x = 0)
\]

\[
\text{while } (\text{let } v = x \text{ in } x \leftarrow x - 1; v > 0) \quad (* (x-- > 0) \text{ in } C *) \\
\text{invariant } x \geq -1 \text{ do } (); \\
\text{assert } (x < 0)
\]
Result value in post-conditions

New addition in the specification language:
- keyword `result` in post-conditions
- denotes the value of the expression executed

Example:
```
{ true }
if x ≥ y then x else y
{ result ≥ x ∧ result ≥ y }
```

Blocking Semantics: Modified Rules

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>[[P]]_{\Sigma, \Pi}</code> holds</td>
<td>$\Sigma, \Pi, \text{assert } P \rightarrow \Sigma, \Pi, ()$</td>
</tr>
<tr>
<td><code>[[I]]_{\Sigma, \Pi}</code> holds</td>
<td>$\Sigma, \Pi, \text{while } c \text{ invariant } I \text{ do } e \rightarrow \Sigma, \Pi, \text{if } c \text{ then } (e; \text{while } c \text{ invariant } I \text{ do } e) \text{ else } ()$</td>
</tr>
</tbody>
</table>

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 $\{P \mid e \mid Q\}$ 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 $\text{WP}(e, Q)$

Expected property: in any state satisfying $\text{WP}(e, Q)$,
- $e$ is guaranteed to execute safely
- if it terminates, $Q$ holds in the final state
New Weakest Precondition Calculus

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

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

’let’ binding

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

Reminder: sequence is a particular case of ‘let’

\[ WP((e_1, e_2), Q) = WP(e_1, WP(e_2, Q)) \]

WP: Exercise

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

\[
WP(\text{let } v = x \text{ in } (x < x + 1; v), x > \text{result}) \\
= WP(x, (WP((x < x + 1; v), x > \text{result})[v \leftarrow \text{result}]))) \\
= WP(x, (WP(x < x + 1, WP(v, x > \text{result}))[v \leftarrow \text{result}]))) \\
= WP(x, (WP(x < x + 1, x > v))[v \leftarrow \text{result}]))) \\
= WP(x, (x + 1 > v)[v \leftarrow \text{result}]))) \\
= WP(x, (x + 1 > \text{result})) \\
= x + 1 > x
\]

Weakest Preconditions, continued

▶ Assignment:

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

▶ Alternative:

\[ WP(x < e, Q) = WP(\text{let } v = e \text{ in } x < v, Q) \]
\[ WP(x < t, Q) = Q[\text{result} \leftarrow ()]; x \leftarrow t \]

Weakest Preconditions, continued

▶ Conditional

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

▶ Alternative with let: (exercise!)
Weakest Preconditions, continued

- **Assertion**
  \[
  \text{WP}(\text{assert } P, Q) = P \land Q = P \land (P \rightarrow Q)
  \]
  (second version useful in practice)

- **While loop**
  \[
  \text{WP}(\text{while } c \text{ invariant } I \text{ do } e, Q) = I \land \forall \vec{v}, (I \rightarrow \text{WP}(c, \text{if } \text{result then WP}(e, I) \text{ else } Q))[w_1 \leftarrow v_1]
  \]
  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.

Outline

"Modern" Approach, Blocking Semantics

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

Termination, Variants

Advanced Modeling of Programs

Programs on Arrays

Soundness of WP

- **Lemma (Preservation by Reduction)**
  If \( \Sigma, \Pi \models \text{WP}(e, Q) \) and \( \Sigma, \Pi, e \leadsto \Sigma', \Pi', e' \) then \( \Sigma', \Pi' \models \text{WP}(e', Q) \)

  **Proof**: predicate induction of \( \leadsto \).

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

  **Proof**: structural induction of \( e \).

- **Corollary (Soundness)**
  If \( \Sigma, \Pi \models \text{WP}(e, Q) \) then
  - \( e \) executes safely in \( \Sigma, \Pi \).
  - if execution terminates, \( Q \) holds in the final state.

Labels: motivation

- Ability to refer to past values of variables

```
{ true }
let v = r in (r <- v + 42; v)
{ r = r@Old + 42 \land \text{result} = r@Old }

{ true }
let tmp = x in x <- y; y <- tmp
{ x = y@Old \land y = x@Old }
```

SUM revisited:

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

Implicitly declared labels:

- \( \text{Here} \), available in every formula
- \( \text{Old} \), available in post-conditions

Labels: Operational Semantics

**Program state**

- becomes a collection of maps indexed by labels
- value of variable \( x \) at label \( L \) is denoted \( \Sigma(x,L) \)

**New semantics of variables in terms:**

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

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

New rules for WP

**New rules for computing WP:**

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

**Exercise:**

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

Example: computation of the GCD

**Euclidean’s algorithm:**

\[
\begin{align*}
\text{requires} & \quad \{ x \geq 0 \land y \geq 0 \} \\
\text{ensures} & \quad \{ \text{result} = \gcd(x@\text{Old}, y@\text{Old}) \} \\
& = L: \\
\text{while} & \quad y > 0 \text{ do} \\
\text{invariant} & \quad \{ x \geq 0 \land y \geq 0 \} \\
\text{invariant} & \quad \{ \gcd(x,y) = \gcd(x@\text{L}, y@\text{L}) \} \\
\text{let} & \quad r = \text{mod} \ x \ y \ \text{in} \ x \leftarrow y; \ y \leftarrow r \\
\text{done;} \\
\text{let} & \quad x \\
\text{See file} & \quad \text{gcd.euclid.labels.mlw}
\end{align*}
\]
Mutable Local Variables

We extend the syntax of expressions with

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

Example: isqrt revisited

```ocaml
val ref x : int
val ref res : int
res <- 0;
let ref sum = 1
while sum ≤ 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

\[
\begin{align*}
\text{WP(let ref } x = e_1 \text{ in } e_2, Q) &= \text{WP}(e_1, \text{WP}(e_2, Q)[x \leftarrow \text{result}]) \\
\text{WP}(x \leftarrow e, Q) &= \text{WP}(e, Q[x \leftarrow \text{result}])
\end{align*}
\]

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

Operational Semantics

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

\[
\begin{align*}
\Sigma, \Pi, e_1 \mapsto \Sigma', \Pi', e'_1 \\
\Sigma, \Pi, \text{let ref } x = e_1 \text{ in } e_2 \mapsto \Sigma', \Pi', \text{let ref } x = e'_1 \text{ in } e_2
\end{align*}
\]

\[
\begin{align*}
\Sigma, \Pi, \text{let ref } x = \nu \text{ in } e \mapsto \Sigma, \Pi\{\{x, \text{Here} \leftarrow \nu\}, e
\end{align*}
\]

And labels too

Functions

Program structure:

\[
\begin{align*}
\text{prog} & ::= \text{decl}^* \\
\text{decl} & ::= \text{vardecl} | \text{fundecl} \\
\text{vardecl} & ::= \text{val ref } id : \text{basetype} \\
\text{fundecl} & ::= \text{let } id((\text{param},)^*) : \text{basetype} \text{ contract body } e \\
\text{param} & ::= \text{id : basetype} \\
\text{contract} & ::= \text{requires } t \text{ writes } (id,)^* \text{ ensures } t
\end{align*}
\]

Function definition:

- Contract:
  - pre-condition
  - post-condition (label \text{Old} available)
  - assigned variables: clause \text{writes}
- Body: expression
Example: isqrt

```plaintext
let isqrt(x:int): int
requires x ≥ 0
ensures result ≥ 0 ∧
    s(r(result)) ≤ x < s(r(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
```

Typing

Definition $d$ of function $f$:

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

Well-formed definitions:

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

where $\Gamma$ contains the global declarations

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: function calls

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

Well-typed function calls:

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

Note: the $t_i$ are immutable expressions
Operational Semantics

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

\[
\Pi' = \{ x_i \mapsto [t_i]_{\Sigma, \Pi} \} \\
\Sigma, \Pi, f(t_1, \ldots, t_n) \leadsto \Sigma, \Pi, (Old : frame(\Pi', Body, Post))
\]

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

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

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

Operational Semantics of Function Call

\( \text{frame} \) is a dummy expression that keeps track of the local variables of the callee:
\[
\Sigma, \Pi, e \mapsto \Sigma', \Pi', e' \\
\Sigma, \Pi'', (frame(\Pi, e, P)) \mapsto \Sigma', \Pi'', (frame(\Pi', e', P))
\]

It also checks that the post-condition holds at the end:
\[
\Sigma, \Pi' \models P[\text{result} \leftarrow v] \\
\Sigma, \Pi, (frame(\Pi, v, P)) \leadsto \Sigma, \Pi , v
\]

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

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

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

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

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

Soundness Theorem for a Complete Program

Assuming that for each function defined as

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

we have

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

then for any formula \( Q \) and any expression \( e \), if \( \Sigma, \Pi \models WP(e, Q) \) then execution of \( \Sigma, \Pi, e \) is safe.

Remark: (mutually) recursive functions are allowed

Outline

- “Modern” Approach, Blocking Semantics
- Syntax extensions
- Termination, Variants
- Advanced Modeling of Programs
- Programs on Arrays

Termination

Goal

Prove that a program terminates (on all inputs satisfying the precondition)

Amounts to show that

- \( \triangleright \) loops never execute infinitely many times
- \( \triangleright \) (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:

\[
x \prec y = x < y \land 0 \leq y
\]

Syntax

New syntax construct:

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

Example:

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

Operational semantics

\[
[I, \Sigma, \Pi]\text{ holds}
\]

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

Weakest Precondition

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

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;
```

```plaintext
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(x_1: \tau_1, ..., x_n: \tau_n): \tau
  requires Pre
  variant var, ≪
  writes w
  ensures Post
  body Body
```

WP for function call:

```plaintext
WP(f(t_1, ..., t_n), Q) = Pre[x_i ← t_i] \land var[x_i ← t_i] ≪ var@Init \land 
∀ \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 `\nu_f` (resp. `\nu_g`) in their contract
- with the *same* well-founded ordering `≪`

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

```plaintext
\nu_g[\vec{y} ← \vec{i}] ≪ \nu_f@Init
```

and symmetrically when `g` calls `f`

Home Work 1: McCarthy’s 91 Function

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

Find adequate specifications

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

Use canvas file `mccarthy.mlw`
Outline

“Modern” Approach, Blocking Semantics

Syntax extensions

Termination, Variants

Advanced Modeling of Programs
   (First-Order) Logic as a Modeling Language
   Ghost code
   Axiomatic Definitions

Programs on Arrays

About Specification Languages

Specification languages:
   ▶ Algebraic Specifications: CASL, Larch
   ▶ Set theory: VDM, Z notation, Atelier B
   ▶ Higher-Order Logic: PVS, Isabelle/HOL, HOL4, Coq
   ▶ Object-Oriented: Eiffel, JML, OCL
   ▶ ...

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 \( \tau, \tau_i \) are logic types (not references)
   ▶ No recursion allowed (yet)
   ▶ No side effects
   ▶ Defines total functions and predicates
Logic Symbols: Examples

**Function**

\[ \text{sqrt}(x: \text{int}) = x \times x \]

**Predicate**

\[
\text{prime}(x: \text{int}) =
\begin{align*}
& x \geq 2 \wedge \\
& \forall y, z : \text{int}. \ y \geq 0 \wedge z \geq 0 \wedge x = y \times z \rightarrow \\
& y = 1 \vee z = 1
\end{align*}
\]

Definition of new logic types: Product Types

- **Tuples types are built-in:**
  
  ```
  \text{type} \ \text{pair} = (\text{int}, \text{int})
  ```

- **Record types can be defined:**
  
  ```
  \text{type} \ \text{point} = \{ \ x: \text{real}; \ y: \text{real} \ \}
  ```

  **Fields are immutable**

  - We allow `let` with pattern, e.g.
    ```
    \text{let} \ (a, b) = \ldots \ \text{in} \ \ldots
    \text{let} \ (x = a; \ y = b) = \ldots \ \text{in} \ \ldots
    ```

  - Dot notation for records fields, e.g.
    ```
    p.x + p.y
    ```

Introducing Ghost Code

**Example: Euclidean division / just compute the remainder:**

```plaintext
q <- 0; r <- x;
while r \geq y do
    invariant \{ x = q \times y + r \}
    r <- r - y; q <- q + 1
```

Introducing Ghost Code

**Example: Euclidean division / just compute the remainder:**

```plaintext
r <- x;
while r \geq y do
    invariant \{ \exists q. x = q \times y + r \}
    r <- r - y;
```
Introducing Ghost Code

Example: Euclidean division / just compute the remainder:

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

Ghost code, ghost variables
▶ Cannot interfere with regular code (checked by typing)
▶ Visible only in annotations
(See Why3 file `euclid_rem.mlw`)

---

Home Work 2

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

```
∃a, b, result = x * a + y * b
```

▶ Prove the program by adding appropriate ghost local variables

Use canvas file `exo_bezout.mlw`

---

Axiomatic Definitions

*Function* and *predicate* declarations of the form

```
function f(τ₁,...,τₙ) : τ
predicate p(τ₁,...,τₙ)
```

together with *axioms*

```
axiom id : formula
```

specify that *f* (resp. *p*) is any symbol satisfying the axioms
## Axiomatic Definitions

**Example: division**

```latex
\begin{align*}
\text{function } \text{div}(\text{real}, \text{real}): \text{real} \\
\text{axiom } \text{mul\_div}: \\
&\forall x, y. y \neq 0 \rightarrow \text{div}(x, y) \cdot y = x
\end{align*}
```

**Example: factorial**

```latex
\begin{align*}
\text{function } \text{fact}(\text{int}): \text{int} \\
\text{axiom } \text{fact0}: \\
&\text{fact}(0) = 1 \\
\text{axiom } \text{factn}: \\
&\forall n: \text{int}. n \geq 1 \rightarrow \text{fact}(n) = n \cdot \text{fact}(n-1)
\end{align*}
```

## Underspecified Logic Functions and Run-time Errors

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

```latex
\begin{align*}
\text{val } \text{div\_real}(x: \text{real}, y: \text{real}): \text{real} \\
&\text{requires } y \neq 0.0 \\
&\text{ensures } \text{result} = \text{div}(x, y)
\end{align*}
```

**Reminder**

Execution blocks when an invalid annotation is met

## More Ghosts: Programs turned into Logic Functions

If the program \( f \) is

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

then there exists a logic function:

```latex
\begin{align*}
\text{let } f(x_1: \tau_1, \ldots, x_n: \tau_n): \tau \\
&\text{requires } \text{Pre} \\
&\text{variant } \text{var}, < \\
&\text{ensures } \text{Post} \\
&\text{body } \text{Body}
\end{align*}
```

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

```latex
\begin{align*}
\text{lemma } f_{\text{spec}}: \forall x_1, \ldots, x_n. \text{Pre} \rightarrow \text{Post}[\text{result } \leftarrow f(x_1, \ldots, x_n)]
\end{align*}
```

Offers an important alternative to axiomatic definitions

In Why3: done using keywords `let function`
Example: axiom-free specification of factorial

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

generates the logic context

```
function fact int : int

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

More Ghosts: Lemma functions

▶ if a program function is \textit{without side effects} and \textit{terminating}:

\begin{itemize}
  \item let \( f(x_1 : \tau_1, \ldots, x_n : \tau_n) : \text{unit} \)
  \begin{itemize}
    \item requires \textit{Pre}
    \item variant \textit{var}, <
    \item ensures \textit{Post}
    \item body \textit{Body}
  \end{itemize}

then it is a proof of

\[ \forall x_1, \ldots, x_n. \textit{Pre} \rightarrow \textit{Post} \]

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

Example: sum of odds

```
function sum_of_odd_numbers int : int

(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 \texttt{arith.lemma.function.mlw}
Example: sum of odds as lemma function

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

Home work 3

Prove the helper lemmas stated for the fast exponentiation algorithm

Home Work 4

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

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

using a lemma function

Outline

“Modern” Approach, Blocking Semantics
Syntax extensions
Termination, Variants
Advanced Modeling of Programs
Programs on Arrays
Higher-order logic as a built-in theory

▶ type of maps: \( \tau_1 \to \tau_2 \)
▶ lambda-expressions: \( \text{fun } x : \tau \to t \)

Definition of selection function:

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

Definition of function update:

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

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

\[
\text{lemma select_store_eq: forall } f : \alpha \to \beta, x : \alpha, v : \beta. \\
\text{select(store}(f, v), x) = v
\]

\[
\text{lemma select_store_neq: forall } f : \alpha \to \beta, x : \alpha, v : \beta. \\
x \neq y \to \text{select(store}(f, v), y) = \text{select}(f, j)
\]

Arrays as Mutable Variables of type “Map”

▶ Array variable: mutable variable of type \( \text{int} \to \alpha \)
▶ In a program, the standard assignment operation

\[
a[i] \gets e
\]

is interpreted as

\[
a \gets \text{store}(a, i, e)
\]

Simple Example

\[
\text{val } \text{ref } a : \text{int} \to \text{int}
\]

\[
\text{let test()} \\
\text{writes } a \\
\text{ensures select}(a, 0) = 13 \quad (* a[0] = 13 *)
\]

\[
\text{body a } \gets \text{store}(a, 0, 13); \quad (* a[0] \gets 13 *)
\]

\[
a \gets \text{store}(a, 1, 42) \quad (* a[1] \gets 42 *)
\]

Exercise: prove this program

Simple Example

\[
\text{WP}(a \gets \text{store}(a, 0, 13); \quad a \gets \text{store}(a, 1, 42), \text{select}(a, 0) = 13) \quad = \\
\text{WP}(a \gets \text{store}(a, 0, 13), \quad \text{WP}(a \gets \text{store}(a, 1, 42), \text{select}(a, 0) = 13)) \quad = \\
\text{WP}(a \gets \text{store}(a, 0, 13), \text{select}(\text{store}(a, 1, 42), 0) = 13) \quad = \\
\text{select}(\text{store}(a, 0, 13), 1, 42), 0) = 13 \quad = \\
\text{select}(\text{store}(a, 0, 13), 0) = 13 \quad = \\
13 = 13 \quad = \quad true
\]

Note how we use both lemmas \text{select_store_eq} and \text{select_store_neq}
Example: Swap

Permute the contents of cells $i$ and $j$ in an array $a$:

```why3
val ref a: int → int

let swap(i:int,j:int)
  writes a
  ensures select(a,i) = select(a@Old,j) ∧
    select(a,j) = select(a@Old,i) ∧
    forall k:int. k ≠ i ∧ k ≠ j →
      select(a,k) = select(a@Old,k)

body
  let tmp = select(a,i) in (* tmp <-a[i]*)
  a <- store(a,i,select(a,j)); (* a[i]<-a[j]*)
  a <- store(a,j,tmp) (* a[j]<-tmp *)
```

Example: Swap again

```why3
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 *)
```

Arrays as Variables of Type (length × map)

- Goal: model “out-of-bounds” run-time errors
- Array variable: mutable variable of type array $\alpha$

```why3
type array $\alpha$ = { length : int; elts : int → $\alpha$ }

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

val set (ref a:array $\alpha$) (i:int) (v:$\alpha$) : 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)
```

Note about Arrays in Why3

```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] }
    { 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
```

Note about Arrays in Why3

```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] }
    { forall k:int. 0 ≤ k < a.length ∧ k ≠ i ∧ k ≠ j →
      a[k] = old a[k] }
  =
    let tmp = a[i] in a[i]<- a[j]; a[j]<- tmp
```
Exercises on Arrays

▶ Prove Swap using WP
▶ Prove the program

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

```ml
var a: array real
let search(n:int, v:real): int
requires 0 \leq n
ensures \{ ? \}
= ?
```

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

2. Implement and prove **linear search**:

```
res <- -1;
for each \(i\) from 0 to \(n−1\): if \(a[i] = v\) then \(res \leftarrow i\);
return \(res\)
```

See file `lin_search.mlw`

Home Work 4: Binary Search

\(low = 0; high = n−1;\)
while \(low \leq high:\)
  let \(m\) be the middle of \(low\) and \(high\)
  if \(a[m] = v\) then return \(m\)
  if \(a[m] < v\) then continue search between \(m\) and \(high\)
  if \(a[m] > v\) then continue search between \(low\) and \(m\)

See file `bin_search.mlw`

Home Work 5: “for” loops

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

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

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

\[
\begin{align*}
\Sigma, \Pi, for \ i = v_1 to v_2 \ do \ e \Rightarrow \Sigma, \Pi, (\) \\
\Sigma, \Pi, for \ i = v_1 to v_2 \ do \ e \Rightarrow \Sigma, \Pi, \ (let \ i = v_1 \ in \ e); \\
\Sigma, \Pi, for \ i = v_1 + 1 to v_2 \ do \ e \Rightarrow \Sigma, \Pi, (for \ i = v_1 + 1 \ to \ v_2 \ do \ e) \\
\end{align*}
\]

Home Work: “for” loops

Propose a Hoare logic rule for the for loop:

\[
\{?\} e(?)
\{?\} \text{for } i = v_1 \text{ to } v_2 \text{ do } e(?)
\]

Propose a rule for computing the WP:

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

That's all for today, Merry Christmas!

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