Basics of deductive program verification

Claude Marché

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

13 décembre 2013
Preliminaries

- Very first question: lectures in English or in French?
Very first question: lectures in English or in French?
Lectures 1,2,3,4: Claude Marché
No lecture on January 10th
Lecture 4 (January 31): practical lab, support for project
Lectures 5,6,7,8: Arthur Charguéraud
Preliminaries

- Very first question: lectures in English or in French?
- Lectures 1,2,3,4: Claude Marché
- No lecture on January 10th
- Lecture 4 (January 31): practical lab, support for project
- Lectures 5,6,7,8: Arthur Charguéraud
- Evaluation:
  - project P, using the Why3 tool (http://why3.lri.fr)
  - final exam E in March 2013 (7th or 14th)
  - final mark = \((2E + P + \max(E, P))/4\)
Preliminaries

- Very first question: lectures in English or in French?
- Lectures 1,2,3,4: Claude Marché
- No lecture on January 10th
- Lecture 4 (January 31): practical lab, support for project
- Lectures 5,6,7,8: Arthur Charguéraud
- Evaluation:
  - project P, using the Why3 tool (http://why3.lri.fr)
  - final exam E in March 2013 (7th or 14th)
  - final mark = \((2E + P + \text{max}(E, P))/4\)
- internships (*stages*)
- Slides, lectures notes on web page
  http://www.lri.fr/~marche/MPRI-2-36-1/
Outline

Introduction, Short History

Classical Hoare Logic
   A Simple Programming Language
   Hoare Logic
   Dijkstra’s Weakest Preconditions

Exercises

“Modern” Approach, Blocking Semantics
   A ML-like Programming Language
   Blocking Operational Semantics
General Objectives

Ultimate Goal

Verify that software is free of bugs

Famous software failures:
http://www.cs.tau.ac.il/~nachumd/horror.html

This lecture

Computer-assisted approaches for verifying that a software conforms to a specification
Some general approaches to Verification

Static analysis, Algorithmic Verification
- *model checking* (automata-based models)
- *abstract interpretation* (domain-specific model, e.g. numerical)
- verification: fully automatic dedicated algorithms

Deductive verification
- formal models using expressive logics
- verification = computer-assisted mathematical proof
Some general approaches to Verification

Refinement
- Formal models
- Code derived from model, correct by construction
A long time before success

Computer-assisted verification is an old idea
- Turing, 1948
- Floyd-Hoare logic, 1969

Success in practice: only from the mid-1990s
- Importance of the *increase of performance of computers*

A first success story:
- Paris metro line 14, using *Atelier B* (1998, refinement approach)
  
  http://www.methode-b.com/documentation_b/ 
  ClearSy-Industrial_Use_of_B.pdf
Other Famous Success Stories

- **Flight control software of A380**: *Astree* verifies absence of run-time errors (2005, abstract interpretation)
  
  http://www.astree.ens.fr/

- **Microsoft’s hypervisor**: using Microsoft’s *VCC* and the *Z3* automated prover (2008, deductive verification)
  

- **Certified C compiler**, developed using the *Coq* proof assistant (2009, correct-by-construction code generated by a proof assistant)
  
  http://compcert.inria.fr/

- **L4.verified micro-kernel**, using tools on top of *Isabelle/HOL* proof assistant (2010, Haskell prototype, C code, proof assistant)
  
Outline

Introduction, Short History

Classical Hoare Logic
  A Simple Programming Language
  Hoare Logic
  Dijkstra’s Weakest Preconditions

Exercises

“Modern” Approach, Blocking Semantics
  A ML-like Programming Language
  Blocking Operational Semantics
Syntax: expressions

\[ e ::= n \quad \text{(integer constants)} \]

\[ \quad \mid x \quad \text{(variables)} \]

\[ \quad \mid e \ op \ e \quad \text{(binary operations)} \]

\[ op ::= + \mid - \mid * \]

\[ \quad \mid = \mid \neq \mid < \mid > \mid \leq \mid \geq \]

\[ \quad \mid \text{and} \mid \text{or} \]

- Only one data type: unbounded integers
- Comparisons return an integer: 0 for “false”, −1 for “true”
- There is no division

Consequences:
- Expressions are always well-typed
- Expressions always evaluate without error
- Expressions do not have any side effect
Syntax: statements

\[
S ::= \begin{align*}
skip & \quad \text{(no effect)} \\
x & := e & \quad \text{(assignment)} \\
s; s & \quad \text{(sequence)} \\
\text{if } e \text{ then } s \text{ else } s & \quad \text{(conditional)} \\
\text{while } e \text{ do } s & \quad \text{(loop)}
\end{align*}
\]

- Condition in `if` and `while`: 0 is “false”, non-zero is “true”
- `if` without `else`: syntactic sugar for `else skip`.

Consequences:
- Statements have side effects
- All programs are well-typed
- There is no possible `runtime error`: all programs execute until their end or infinitely
Running Example

Three global variables $n$, count, and sum:

```plaintext
count := 0; sum := 1;
while sum ≤ n do
  count := count + 1; sum := sum + 2 * count + 1
```

What does this program compute?

(assuming input is $n$ and output is count)

Informal specification:

▶ at the end of execution of this program, count contains the square root of $n$, rounded downward

▶ e.g. for $n=42$, the final value of count is 6.
Running Example

Three global variables \( n \), \( \text{count} \), and \( \text{sum} \)

\[
\text{count} := 0; \text{sum} := 1; \\
\text{while} \text{sum} \leq n \text{ do} \\
\quad \text{count} := \text{count} + 1; \text{sum} := \text{sum} + 2 * \text{count} + 1
\]

What does this program compute?
(assuming input is \( n \) and output is \( \text{count} \))
Running Example

Three global variables $n$, $\text{count}$, and $\text{sum}$

```plaintext
count := 0; sum := 1;
while sum $\leq$ n do
  count := count + 1; sum := sum + 2 * count + 1
```

What does this program compute?
(assuming input is $n$ and output is $\text{count}$)

Informal specification:

- at the end of execution of this program, $\text{count}$ contains the square root of $n$, rounded downward
- e.g. for $n=42$, the final value of $\text{count}$ is 6.
Propositions about programs

- To formally express properties of programs, we need a **formal specification language**
- We use **standard first-order logic**
- Syntax of formulas:

  \[ p ::= e \mid p \land p \mid p \lor p \mid \neg p \mid p \Rightarrow p \mid \forall v, p \mid \exists v, p \]

- \(v\): **logical variable** identifiers
- \(e\): program expressions, augmented with logical variables
Hoare triples

- **Hoare triple**: notation $\{ P \} s \{ Q \}$
- $P$: formula called the **precondition**
- $Q$: formula called the **postcondition**

**Intended meaning**

$\{ P \} s \{ Q \}$ is true if and only if:
when the program $s$ is executed in any state satisfying $P$, then
(if execution terminates) its resulting state satisfies $Q$

This is a **Partial Correctness**: we say nothing if $s$ does not terminates
Examples of valid triples for partial correctness:

- $\{ x = 1 \} x := x + 2 \{ x = 3 \}$
- $\{ x = y \} x := x + y \{ x = 2 \cdot y \}$
- $\{ \exists v, \ x = 4 \cdot v \} x := x + 42 \{ \exists w, \ x = 2 \cdot w \}$
- $\{ true \} \text{while 1 do skip} \{ false \}$

Our running example:

$\{ ? \} ISQRT \{ ? \}$
Examples of valid triples for partial correctness:

- $\{x = 1\} x := x + 2 \{x = 3\}$
- $\{x = y\} x := x + y \{x = 2 \times y\}$
- $\{\exists v, x = 4 \times v\} x := x + 42 \{\exists w, x = 2 \times w\}$
- $\{true\}\text{while 1 do skip}\{false\}$

Our running example:

$\{n \geq 0\} ISQRT\{?\}$
Examples

Examples of valid triples for partial correctness:

- \( \{ x = 1 \} x := x + 2 \{ x = 3 \} \)
- \( \{ x = y \} x := x + y \{ x = 2 \ast y \} \)
- \( \{ \exists v, \ x = 4 \ast v \} x := x + 42 \{ \exists w, \ x = 2 \ast w \} \)
- \( \{ \text{true} \} \text{while } 1 \text{ do skip} \{ \text{false} \} \)

Our running example:

\( \{ n \geq 0 \} \text{ISQRT} \{ \text{count} \ast \text{count} \leq n \land n < (\text{count} + 1) \ast (\text{count} + 1) \} \)
Running Example: Demo


See file *imp_isqrt.mlw*

(This is the tool to use for the project, version 0.82)
Hoare logic as an Axiomatic Semantics

Original Hoare logic \([\sim 1970]\)

**Axiomatic Semantics** of programs

Set of *inference rules* producing triples

\[
\begin{align*}
\{P\}\text{skip}\{P\} \\
\{P[x \leftarrow e]\}x := e\{P\} \\
\{P\}s_1\{Q\} \quad \{Q\}s_2\{R\} \\
\{P\}s_1; s_2\{R\}
\end{align*}
\]

- Notation \(P[x \leftarrow e]\) : replace all occurrences of program variable \(x\) by \(e\) in \(P\).
Hoare Logic, continued

Consequence rule:

\[
\begin{align*}
\{P'\}s\{Q'\} & \quad \models P \Rightarrow P' \\
\{P\}s\{Q\} & \quad \models Q' \Rightarrow Q
\end{align*}
\]

- Example: proof of

\[
\{x = 1\}x := x + 2\{x = 3\}
\]
Hoare Logic, continued

Rules for if and while:

\[
\begin{align*}
\{P \land e \neq 0\} & s_1 \{Q\} & \{P \land e = 0\} & s_2 \{Q\} \\
\{P\} & \text{if } e \text{ then } s_1 \text{ else } s_2 \{Q\} \\
\{I \land e \neq 0\} & s\{I\} \\
\{I\} & \text{while } e \text{ do } s\{I \land e = 0\}
\end{align*}
\]

- \(I\) is called a \textit{loop invariant}.
Example: \text{isqrt}(42)

Exercise: prove of the triple

\[
\{ n \geq 0 \} \text{ISQRT} \ \{ count \ast count \leq n \land n < (count + 1) \ast (count + 1) \}\]
Example: isqrt(42)

Exercise: prove of the triple

\(\{ n \geq 0 \} \ ISQRT \ \{ \text{count} \cdot \text{count} \leq n \land n < (\text{count}+1) \cdot (\text{count}+1) \}\)

Could we do that by hand?
Example: \( \text{isqrt}(42) \)

Exercise: prove of the triple

\( \{ n \geq 0 \} \text{ ISQRT } \{ \text{count} \times \text{count} \leq n \wedge n < (\text{count}+1) \times (\text{count}+1) \} \)

Could we do that by hand?

Back to demo: file \texttt{imp_isqrt.mlw}
Example: isqrt(42)

Exercise: prove of the triple

\{n \geq 0\} ISQRT \{count \times count \leq n \land n < (count + 1) \times (count + 1)\}

Could we do that by hand?

Back to demo: file imp_isqrt.mlw

Warning
Finding an adequate loop invariant is a major difficulty
Beyond Axiomatic Semantics

- Operational Semantics
- Semantic Validity of Hoare Triples
- Hoare logic as correct deduction rules
Operational semantics

[Plotkin 1981, structural operational semantics (SOS)]

- we use a standard small-step semantics
- **program state**: describes content of global variables at a given time. It is a finite map $\Sigma$ associating to each variable $x$ its current value denoted $\Sigma(x)$.
- Value of an expression $e$ in some state $\Sigma$:
  - denoted $[e]_\Sigma$
  - always defined, by the following recursive equations:

\[
\begin{align*}
[n]_\Sigma &= n \\
[x]_\Sigma &= \Sigma(x) \\
[e_1 \ op \ e_2]_\Sigma &= [e_1]_\Sigma [op] [e_2]_\Sigma
\end{align*}
\]

- $[op]$ natural semantic of operator $op$ on integers (with relational operators returning $0$ for false and $-1$ for true).
Semantics of statements

Semantics of statements: defined by judgment

$$\Sigma, s \rightsquigarrow \Sigma', s'$$

meaning: in state $\Sigma$, executing one step of statement $s$ leads to the state $\Sigma'$ and the remaining statement to execute is $s'$. The semantics is defined by the following rules.

$$\Sigma, x := e \rightsquigarrow \Sigma\{x \leftarrow [e]_\Sigma\}, \text{skip}$$

$$\Sigma, s_1 \rightsquigarrow \Sigma', s'_1$$
$$\Sigma, (s_1; s_2) \rightsquigarrow \Sigma', (s'_1; s_2)$$

$$\Sigma, (\text{skip}; s) \rightsquigarrow \Sigma, s$$
Semantics of statements, continued

\[ \text{if } e \text{ then } s_1 \text{ else } s_2 \leadsto \Sigma, s_1 \]

\[ \text{while } e \text{ do } s \leadsto \Sigma, \text{skip} \]

\[ \text{skip} \leadsto \Sigma \]

\[ \text{true} \leadsto \Sigma \]

\[ \text{false} \leadsto \Sigma \]

\[ \text{true} \leadsto \Sigma \]

\[ \text{false} \leadsto \Sigma \]

\[ \text{if } e \text{ then } s_1 \text{ else } s_2 \leadsto \Sigma, s_1 \]

\[ \text{while } e \text{ do } s \leadsto \Sigma, (s; \text{while } e \text{ do } s) \]

\[ \text{if } e \text{ then } s_1 \text{ else } s_2 \leadsto \Sigma, s_1 \]
Execution of programs

▶ $\rightsquigarrow$: a binary relation over pairs (state, statement)
▶ transitive closure: $\rightsquigarrow^+$
▶ reflexive-transitive closure: $\rightsquigarrow^*$

In other words:

$$\Sigma, s \rightsquigarrow^* \Sigma', s'$$

means that statement $s$, in state $\Sigma$, reaches state $\Sigma'$ with remaining statement $s'$ after executing some finite number of steps.

Running example:

$$\{ n = 42, \text{count} = ?, \text{sum} = ? \}, \text{ISQRT} \rightsquigarrow^* \{ n = 42, \text{count} = 6, \text{sum} = 49 \}, \text{skip}$$
Execution and termination

- any statement except `skip` can execute in any state
- the statement `skip` alone represents the final step of execution of a program
- there is no possible runtime error.

Definition

Execution of statement \( s \) in state \( \Sigma \) **terminates** if there is a state \( \Sigma' \) such that \( \Sigma, s \leadsto^* \Sigma', \text{skip} \)

- since there are no possible runtime errors, \( s \) does not terminate means that \( s \) **diverges** (i.e. executes infinitely).
Semantics of formulas

\[ [p]_\Sigma : \]

- semantics of formula \( p \) in program state \( \Sigma \)
- is a logic formula where no program variables appear anymore
- defined recursively as follows.

\[
\begin{align*}
[e]_\Sigma &= [e]_\Sigma \neq 0 \\
[p_1 \land p_2]_\Sigma &= [p_1]_\Sigma \land [p_2]_\Sigma \\
&\vdots
\end{align*}
\]

where semantics of expressions is augmented with

\[
\begin{align*}
[v]_\Sigma &= v \\
[x]_\Sigma &= \Sigma(x)
\end{align*}
\]

Notations:

- \( \Sigma \models p \) : the formula \( [p]_\Sigma \) is valid
- \( \models p \) : formula \( [p]_\Sigma \) holds in all states \( \Sigma \).
Soundness

**Definition (Partial correctness)**

Hoare triple \( \{P\} s \{Q\} \) is said valid if:

for any states \( \Sigma, \Sigma' \), if

- \( \Sigma, s \leadsto^* \Sigma', \text{skip} \) and
- \( \Sigma \models P \)

then \( \Sigma' \models Q \)

**Theorem (Soundness of Hoare logic)**

*The set of rules is correct: any derivable triple is valid.*

This is *proved by induction on the derivation tree* of the considered triple.

For each rule: assuming that the triples in premises are valid, we show that the triple in conclusion is valid too.
Completeness

Two major difficulties for proving a program

- **guess the appropriate intermediate formulas** (for sequence, for the loop invariant)
- **prove the logical premises of consequence rule**

Theoretical question: completeness. Are all valid triples derivable from the rules?

**Theorem (Relative Completeness of Hoare logic)**

*The set of rules of Hoare logic is relatively complete: if the logic language is expressive enough, then any valid triple $\{P\} \mathcal{S} \{Q\}$ can be derived using the rules.*

*Cook, 1978*

“Expressive enough” is for example Peano arithmetic (non-linear integer arithmetic)

Gives only hints on how to effectively determine suitable loop invariants (see the theory of abstract interpretation *Cousot, 1990*)
Annotated Programs

Goal
Add automation to the Hoare logic approach

We augment our simple language with **explicit loop invariants**

\[
\begin{align*}
  s & ::= \text{skip} & \text{(no effect)} \\
  & | x := e & \text{(assignment)} \\
  & | s; s & \text{(sequence)} \\
  & | \text{if } e \text{ then } s \text{ else } s & \text{(conditional)} \\
  & | \text{while } e \text{ invariant } / \text{ do } s & \text{(annotated loop)}
\end{align*}
\]

- The operational semantics is unchanged.
Weakest liberal precondition

[Dijkstra 1975]

Function $WLP(s, Q)$:
- $s$ is a statement
- $Q$ is a formula
- returns a formula

It should return the minimal precondition $P$ that validates the triple $\{P\}s\{Q\}$
Definition of $\text{WLP}(s, Q)$

Recursive definition:

\[
\begin{align*}
\text{WLP}(\text{skip}, Q) & = Q \\
\text{WLP}(x := e, Q) & = Q[x \leftarrow e] \\
\text{WLP}(s_1; s_2, Q) & = \text{WLP}(s_1, \text{WLP}(s_2, Q)) \\
\text{WLP}(\text{if } e \text{ then } s_1 \text{ else } s_2, Q) & = (e \neq 0 \Rightarrow \text{WLP}(s_1, Q)) \land (e = 0 \Rightarrow \text{WLP}(s_2, Q))
\end{align*}
\]
Definition of $WLP(s, Q)$, continued

\[
WLP(\text{while } e \text{ invariant } I \text{ do } s, Q) = \\
I \land \\
\forall v_1, \ldots, v_k, \\
(((e \neq 0 \land I) \Rightarrow WLP(s, I)) \land ((e = 0 \land I) \Rightarrow Q))[w_i \leftarrow v_i]
\]

(invariant true initially)

(invariant preserved)

(invariant implies post)

where $w_1, \ldots, w_k$ is the set of assigned variables in statement $s$ and $v_1, \ldots, v_k$ are fresh logic variables
Examples

\[ WLP(x := x + y, x = 2y) \equiv x + y = 2y \]
Examples

\[
\text{WLP}(x := x + y, x = 2y) \equiv x + y = 2y
\]

\[
\text{WLP(while } y > 0 \text{ invariant } \text{even}(y) \text{ do } y := y - 2, \text{even}(y)) \equiv
\]

Examples

\[ \text{WLP}(x := x + y, x = 2y) \equiv x + y = 2y \]

\[ \text{WLP(while } y > 0 \text{ invariant } \text{even}(y) \text{ do } y := y - 2, \text{even}(y)) \equiv \]
\[ \text{even}(y) \land \]
\[ \forall v, ((v > 0 \land \text{even}(v)) \Rightarrow \text{even}(v - 2)) \land ((v \leq 0 \land \text{even}(v)) \Rightarrow \text{even}(v)) \]
Soundness

**Theorem (Soundness)**

For all statement $s$ and formula $Q$, $\{WLP(s, Q)\}s\{Q\}$ is valid.

Proof by induction on the structure of statement $s$.

**Consequence**

For proving that a triple $\{P\}s\{Q\}$ is valid, it suffices to prove the formula $P \Rightarrow WLP(s, Q)$.

This is basically what Why3 does.
Outline

Introduction, Short History

Classical Hoare Logic
  A Simple Programming Language
  Hoare Logic
  Dijkstra’s Weakest Preconditions

Exercises

“Modern” Approach, Blocking Semantics
  A ML-like Programming Language
  Blocking Operational Semantics
Exercise 1

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

\[
x := a; \ y := b;
\] \hspace{1cm}
\[
\text{while } y > 0 \ \text{do}
\hspace{1cm}
\quad x := x + 1; \ y := y - 1
\]

- Propose a post-condition stating that the final value of $x$ is the sum of the values of $a$ and $b$
- Find an appropriate loop invariant
- Prove the program.
Exercise 2

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

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

▷ Propose a formal precondition to express that \( x \) is assumed non-negative, \( y \) is assumed positive, and a formal post-condition expressing that \( q \) and \( r \) are respectively the quotient and the remainder of the Euclidean division of \( x \) by \( y \).

▷ Find appropriate loop invariant and prove the correctness of the program.
Exercise 3

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

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

- Assuming that the power function exists in the logic, specify appropriate pre- and post-conditions for this program.
- Find an appropriate loop invariant, and prove the program.
The Fibonacci sequence is defined recursively by $fib(0) = 0$, $fib(1) = 1$ and $fib(n + 2) = fib(n + 1) + fib(n)$. The following program is supposed to compute $fib$ in linear time, the result being stored in $y$.

```plaintext
y := 0; x := 1; i := 0;
while i < n do
    aux := y; y := x; x := x + aux; i := i + 1
```

- Assuming $fib$ 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.
Outline

Introduction, Short History

Classical Hoare Logic
- A Simple Programming Language
- Hoare Logic
- Dijkstra’s Weakest Preconditions

Exercises

“Modern” Approach, Blocking Semantics
- A ML-like Programming Language
- Blocking Operational Semantics
Summary of Previous Section

- Very simple programming language
  - program = sequence of statements
  - only global variables
  - only the integer data type, always well typed
- Formal operational semantics
  - small steps
  - no run-time errors
- Hoare logic:
  - Deduction rules for triples \( \{Pre\}s\{Post\} \)
- Weakest Liberal Precondition (WLP):
  - if \( Pre \Rightarrow WLP(s, Post) \) then \( \{Pre\}s\{Post\} \) valid
- In lecture notes: extensions for termination
  - Total correctness of triples
  - Weakest (Strict) Precondition
Next step

Extend the 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`.
- Former pure expressions are now called *terms*.
- No difference between expressions and statements anymore.

<table>
<thead>
<tr>
<th>previous section</th>
<th>now</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>term</td>
</tr>
<tr>
<td>formula</td>
<td>formula</td>
</tr>
<tr>
<td>statement</td>
<td>expression</td>
</tr>
</tbody>
</table>

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`

\[
t ::= val \quad \text{(values, i.e. constants)}
\]
\[
| \quad v \quad \text{(logic variables)}
\]
\[
| \quad x \quad \text{(program variables)}
\]
\[
| \quad t \ op \ t \quad \text{(binary operations)}
\]
\[
| \quad \text{if } t \ \text{then } t \ \text{else } t \quad \text{(if-expression)}
\]
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 (Alt-Ergo, CVC3, Z3) typically support these types
- 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 last lecture, but also addition of local binding:

\[
\begin{align*}
p & ::= t & \text{(boolean term)} \\
& | p \land p \mid p \lor p \mid \neg p \mid p \Rightarrow p & \text{(connectives)} \\
& | \forall v : \tau, p \mid \exists v : \tau, p & \text{(quantification)} \\
& | \text{let } v = t \text{ in } p & \text{(local binding)}
\end{align*}
\]
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 \( \nu : \tau \) (logic variable, immutable)
  - either \( x : \text{ref } \tau \) (program variable, mutable)

Important

- a reference is not a value
- there is no “reference on a reference”
- no aliasing
Typing rules

Constants:

\[ \Gamma \vdash n : \text{int} \quad \Gamma \vdash r : \text{real} \]

\[ \Gamma \vdash \text{True} : \text{bool} \quad \Gamma \vdash \text{False} : \text{bool} \]

Variables:

\[ \frac{\nu : \tau \in \Gamma}{\Gamma \vdash \nu : \tau} \quad \frac{x : \text{ref} \tau \in \Gamma}{\Gamma \vdash x : \tau} \]

Let binding:

\[ \frac{\Gamma \vdash t_1 : \tau_1 \quad \{ \nu : \tau_1 \} \cdot \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash \text{let} \ \nu = t_1 \ \text{in} \ t_2 : \tau_2} \]

- All terms have a base type (not a reference)
- In practice: Why3, as in OCaml, requires to write \(!x\) for references
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)

$\lbrack val \rbrack_{\Sigma,\Pi} = val$  \hspace{1cm} (values)
$\lbrack x \rbrack_{\Sigma,\Pi} = \Sigma(x)$ \hspace{1cm} if $x : \text{ref } \tau$
$\lbrack v \rbrack_{\Sigma,\Pi} = \Pi(v)$ \hspace{1cm} if $v : \tau$

$\lbrack t_1 \; op \; t_2 \rbrack_{\Sigma,\Pi} = \lbrack t_1 \rbrack_{\Sigma,\Pi} \; [op] \; \lbrack t_2 \rbrack_{\Sigma,\Pi}$

$\lbrack \text{let } v = t_1 \; \text{in } t_2 \rbrack_{\Sigma,\Pi} = \lbrack t_2 \rbrack_{\Sigma,(\{v=\lbrack t_1 \rbrack_{\Sigma,\Pi}\} \cdot \Pi)}$

**Warning**

Semantics is now a partial function
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 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

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

- 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 \( e_2 \)
Toy Examples

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

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

\[ \text{while } (x := x - 1; x > 0) \text{ do ()} \]

\[ \text{while (let } v = x \text{ in } x := x - 1; v > 0) \text{ do ()} \]
Typing Rules for Expressions

Assignment:
\[ x : \text{ref} \tau \in \Gamma \quad \Gamma \vdash e : \tau \]
\[ \Gamma \vdash x := e : \text{unit} \]

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

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

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

Novelties

- Need for context rules
- Precise the order of evaluation: left to right

- One-step execution has the form

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

- Values do not reduce
Operational Semantics

- **Assignment**

\[
\begin{align*}
\Sigma, \Pi, e \rightsquigarrow \Sigma', \Pi', e' \\
\Sigma, \Pi, x := e \rightsquigarrow \Sigma', \Pi', x := e'
\end{align*}
\]

\[
\Sigma, \Pi, x := \text{val} \rightsquigarrow \Sigma[x \leftarrow \text{val}], \Pi, ()
\]

- **Let binding**

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

\[
\Sigma, \Pi, \text{let } v = \text{val} \text{ in } e \rightsquigarrow \Sigma, \{v = \text{val}\} \cdot \Pi, e
\]
Operational Semantics, Continued

- Binary operations

\[
\begin{align*}
\Sigma, \Pi, e_1 & \rightsquigarrow \Sigma', \Pi', e_1' \\
\Sigma, \Pi, e_1 + e_2 & \rightsquigarrow \Sigma', \Pi', e_1' + e_2 \\
\Sigma, \Pi, e_2 & \rightsquigarrow \Sigma', \Pi', e_2' \\
\Sigma, \Pi, val_1 + e_2 & \rightsquigarrow \Sigma', \Pi', val_1 + e_2' \\
val & = val_1 + val_2 \\
\Sigma, \Pi, val_1 + val_2 & \rightsquigarrow \Sigma, \Pi, val
\end{align*}
\]
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 `let v = ... in ...` instead, e.g.:

  ![Math notation]

  \[
  \nu_1, \nu_2 \text{ fresh} \\
  \Sigma, \Pi, e_1 + e_2 \rightsquigarrow \Sigma, \Pi, \text{let } \nu_1 = e_1 \text{ in let } \nu_2 = e_2 \text{ in } \nu_1 + \nu_2
  \]

- 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

\[
e ::= \text{assert } p \quad \text{(assertion)}
\mid \text{while } e \text{ invariant } \mid \text{do } e \quad \text{(annotated loop)}
\]
Toy Examples

\[
z := \text{if } x \geq y \text{ then } x \text{ else } y ;
assert \ z \geq x \land z \geq y
\]

\[
\text{while } (x := x - 1; x > 0) \\
\quad \text{invariant } x \geq 0 \text{ do ()};
assert (x = 0)
\]

\[
\text{while } (\text{let } v = x \text{ in } x := x - 1; v > 0) \\
\quad \text{invariant } x \geq -1 \text{ do ()};
assert (x < 0)
\]
Blocking Semantics: Modified Rules

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

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

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 \( Q \)

New addition in the specification language:

- keyword result in post-conditions
- denotes the value of the expression executed
Toy Examples, Continued

{ true }
if \( x \geq y \) then \( x \) else \( y \)
{ result \geq x \land result \geq y \}

{ x \geq 0 }
c := 0; sum := 1;
while sum \leq x do
c := c + 1; sum := sum + 2 \ast c + 1
done;
c
{ result \geq 0 \land
result \ast result \leq x < (result+1)\ast (result+1) \}
To be continued...

To be continued in next lecture