Basics of deductive program verification

Claude Marché

Cours MPRI 2-36-1 "Preuve de Programme"

11 décembre 2013

Preliminaries

- Very first question: lectures in English or in French?
- Lectures 1,2,3,4: Claude Marché
  - To be confirmed no lecture on January 8th
- Lectures 5,6,7,8: Arthur Charguéraud
- one week in February: lecture replaced by practical lab, support for project
- Evaluation:
  - project P, using the Why3 tool (http://why3.lri.fr)
    - return date: Monday, February 16th, 2015
  - final exam E: Thursday, March 12th, 2015, 16:15, same room as the lecture.
  - 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:


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

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)

Other Famous Success Stories

  http://www.astree.ens.fr/
- Microsoft's hypervisor: using Microsoft's *VCC* and the *Z3* automated prover (2008, deductive verification)
  More recently: verification of PikeOS
- 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

\[
\begin{align*}
e & ::= \ n \\
    & \mid \ x \\
    & \mid \ e \ op \ e \\
\end{align*}
\]

- 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

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

- Condition in \text{if} and \text{while}: 0 is “false”, non-zero is “true”
- \text{if} without \text{else}: syntactic sugar for \text{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 \(sum\)

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

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.
Propositions about programs

- To formally express properties of programs, we need a formal specification language
- We use standard first-order logic
- Syntax of formulas:
  \[ p ::= e | p \land p | p \lor p | \neg p | p \Rightarrow p | \forall v, p | \exists v, p \]
- \( v \): logical variable
- \( e \): program expressions, augmented with logical variables

Examples

Examples of valid triples for partial correctness:
- \( \{x = 1\} x := x + 2\{x = 3\} \)
- \( \{x = y\} x := x + y\{x = 2 \ast y\} \)
- \( \{\exists v, x = 4 \ast v\} x := x + 42\{\exists w, x = 2 \ast w\} \)
- \( \{true\} while 1 do skip\{false\} \)

Our running example:
\( \{?n \geq 0\} ISQRT\{?count \ast count \leq n \land n < (count+1) \ast (count+1)\} \)

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

Running Example: Demo

Demo with the Why3 tool (http://why3.lri.fr/)

See file imp_isqrt.mlw

(This is the tool to use for the project, version 0.85)
Hoare logic as an Axiomatic Semantics

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

Axiomatic Semantics of programs

Set of inference rules producing triples

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

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

Example: proof of

\[\{x = 1\} x := x + 2\{x = 3\}\]

Hoare Logic, continued

Consequence rule:

\[
\{P\} s\{Q\} \quad \models P \Rightarrow P' \quad \models Q' \Rightarrow Q \\
\{P\} s\{Q\}
\]

- Example: proof of

\[\{x = 1\} x := x + 2\{x = 3\}\]

Example: isqrt(42)

Exercise: prove of the triple

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

Could we do that by hand?

Back to demo: file \texttt{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 $\Sigma(x)$.
- 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

$$
\begin{align*}
\Sigma, \text{if } e \text{ then } s_1 \text{ else } s_2 &\rightsquigarrow \Sigma, s_1 \\
\Sigma, \text{if } e \text{ then } s_1 \text{ else } s_2 &\rightsquigarrow \Sigma, s_2 \\
\Sigma, \text{while } e \text{ do } s &\rightsquigarrow \Sigma, (s; \text{while } e \text{ do } s) \\
\Sigma, \text{while } e \text{ do } s &\rightsquigarrow \Sigma, \text{skip}
\end{align*}
$$
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:

\( \{n = 42, \text{count} = ?, \text{sum} = ?\}, \text{ISQR}T \rightsquigarrow^* \{n = 42, \text{count} = 6, \text{sum} = 49\}, \text{skip} \)

Execution and termination

- any statement except \( \text{skip} \) can execute in any state
- the statement \( \text{skip} \) 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 \( \Sigma, s \rightsquigarrow^* \Sigma', \text{skip} \)

- 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.

\[ [e]_\Sigma = [e]_\Sigma \neq 0 \]
\[ [p_1 \land p_2]_\Sigma = [p_1]_\Sigma \land [p_2]_\Sigma \]
\[ ... \]

where semantics of expressions is augmented with

\[ [v]_\Sigma = v \]
\[ [x]_\Sigma = \Sigma(x) \]

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

- \( \Sigma, s \rightsquigarrow^* \Sigma', \text{skip} \) 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 \( \{P\} s \{Q\} \) 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

\[

table = 
\begin{array}{ll}
\text{s} & \text{skip} \\
\text{x := e} & \text{assignment} \\
\text{s; s} & \text{sequence} \\
\text{if e then s else s} & \text{conditional} \\
\text{while e invariant / do s} & \text{annotated loop}
\end{array}
\]

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 WLP\( (s, Q) \)

Recursive definition:

\[
\begin{align*}
\text{WLP(\text{skip}, Q)} &= Q \\
\text{WLP(x := e, Q)} &= Q[x \leftarrow e] \\
\text{WLP(s1; s2, Q)} &= \text{WLP(s1, WLP(s2, Q))} \\
\text{WLP(if e then s1 else s2, Q)} &= (e \neq 0 \Rightarrow \text{WLP(s1, Q)}) \land (e = 0 \Rightarrow \text{WLP(s2, Q)})
\end{align*}
\]
Definition of $WLP(s, Q)$, continued

$$WLP(\text{while } e \text{ invariant } l \text{ do } s, Q) =$$

$$l \land \forall v_1, \ldots, v_k,$$

$$(((e \neq 0 \land l) \Rightarrow WLP(s, l)) \land ((e = 0 \land l) \Rightarrow Q)) \ [w_i \leftarrow v_i]$$

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

$$WLP(x := x + y, x = 2y) \equiv x + y = 2y$$

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

$$\forall v, ((v > 0 \land \text{even}(v)) \Rightarrow \text{even}(v - 2)) \land ((v \leq 0 \land \text{even}(v)) \Rightarrow \text{even}(v))$$

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 $P \Rightarrow WLP(s, Q)$.

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 $a + b$.

```plaintext
x := a; y := b;
while y > 0 do
  x := x + 1; y := y - 1
```

(Why3 file to fill in: `imp_sum.mlw`)

▶ 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
```

(Why3 file to fill in: `imp_euclide.mlw`)

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

Exercise 3

Let's assume given in the underlying logic the functions `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);
```

(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.

Exercise 4

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

\[
\{P\} x := e \{\exists v, e[x ← v] = x ∧ P[x ← v]\}
\]

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

\[
\{P\} x := e \{\exists v, e[x ← v] = x ∧ P[x ← v]\}
\]

3. Show that the triple \(\{P[x ← e]\} x := e\{P\}\) 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 \(\{\text{Pre}\} s \{\text{Post}\}\)
  - Weakest Liberal Precondition (WLP):
    - if \(\text{Pre} \Rightarrow \text{WLP}(s, \text{Post})\) then \(\{\text{Pre}\} s \{\text{Post}\}\) 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

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 ...
```

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 \( t_1 \) else \( t_2 \)

\[
\begin{align*}
t & ::= \text{val} & & \text{(values, i.e. constants)} \\
& | v & & \text{(logic variables)} \\
& | x & & \text{(program variables)} \\
& | t \text{ op } t & & \text{(binary operations)} \\
& | \text{if } t \text{ then } t_1 \text{ else } t_2 & & \text{(if-expression)}
\end{align*}
\]

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 previous syntax, but also addition of local binding:

\[ p ::= t \mid p \land p \mid p \lor p \mid \neg p \mid p \Rightarrow p \mid \forall v : \tau, p \mid \exists v : \tau, p \mid \text{let } v = t \text{ in } p \]

(connectives)

(Typing)

Types:

\[ \tau ::= \text{int} | \text{real} | \text{bool} | \text{unit} \]

Typing judgment:

\[ \Gamma \vdash t : \tau \]

where \( \Gamma \) maps identifiers to types:

- either \( v : \tau \) (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:

\[ v : \tau \in \Gamma \]

\[ \Gamma \vdash v : \tau \]

\[ x : \text{ref } \tau \in \Gamma \]

\[ \Gamma \vdash x : \tau \]

Let binding:

\[ \Gamma \vdash t_1 : \tau_1 \]

\[ \{ v : \tau_1 \} \cdot \Gamma \vdash t_2 : \tau_2 \]

\[ \Gamma \vdash \text{let } v = t_1 \text{ in } t_2 : \tau_2 \]

Important

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

Warning

Semantics is now a partial function
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 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

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

- 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

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

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

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

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

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

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

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

Novelties

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

- one-step execution has the form
  \[ \Sigma, \Pi, e \leadsto \Sigma', \Pi', e' \]

- values do not reduce

Operational Semantics, Continued

- Assignment
  \[
  \frac{\Sigma, \Pi, e \leadsto \Sigma', \Pi', e'}{\Sigma, \Pi, x := e \leadsto \Sigma', \Pi', x := e'}
  \]
  \[
  \Sigma, \Pi, x := \text{val} \leadsto \Sigma[x \leftarrow \text{val}], \Pi, ()
  \]

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

- Binary operations
  \[
  \frac{\Sigma, \Pi, e_1 \leadsto \Sigma', \Pi', e'_1}{\Sigma, \Pi, e_1 + e_2 \leadsto \Sigma', \Pi', e'_1 + e_2}
  \]
  \[
  \frac{\Sigma, \Pi, e_2 \leadsto \Sigma', \Pi', e'_2}{\Sigma, \Pi, \text{val}_1 + e_2 \leadsto \Sigma', \Pi', \text{val}_1 + e'_2}
  \]
  \[
  \frac{\Sigma, \Pi, \text{val}_1 + \text{val}_2 \leadsto \Sigma, \Pi, \text{val}}{\text{val} = \text{val}_1 + \text{val}_2}
  \]
Operational Semantics, Continued

- **Conditional**

  \[ \Sigma, \Pi, c \rightarrow \Sigma', \Pi', c' \]
  \[ \Sigma, \Pi, \text{if } c \text{ then } e_1 \text{ else } e_2 \rightarrow \Sigma', \Pi', \text{if } c' \text{ then } e_1 \text{ else } e_2 \]

  \[ \Sigma, \Pi, \text{if } True \text{ then } e_1 \text{ else } e_2 \rightarrow \Sigma, \Pi, e_1 \]

  \[ \Sigma, \Pi, \text{if } False \text{ then } e_1 \text{ else } e_2 \rightarrow \Sigma, \Pi, e_2 \]

- **Loop**

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

Context Rules versus Let Binding

Remark: most of the context rules can be avoided

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

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

- 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

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

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

while \((x := x - 1; x > 0)\)
  invariant \( x \geq 0 \) do ()
assert \( x = 0 \)

while (let \( v = x \in x := x - 1; v > 0 \))
  invariant \( x \geq -1 \) do ()
assert \( x < 0 \)

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

Blocking Semantics: Modified Rules

\[ \llbracket P \rrbracket_{\Sigma, \Pi} \text{ holds} \]
\[ \Sigma, \Pi, \text{assert } P \leftrightarrow \Sigma, \Pi, () \]

\[ \llbracket \bot \rrbracket_{\Sigma, \Pi} \text{ holds} \]
\[ \Sigma, \Pi, \text{while } c \text{ invariant } \bot \text{ do } e \leftrightarrow \Sigma, \Pi, \text{if } c \text{ then } (\ell; \text{while } c \text{ invariant } \bot \text{ do } e) \text{ else } () \]

Important
Execution blocks as soon as an invalid annotation is met

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 in next lecture