Basics of deductive program verification

Claude Marché

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

7 décembre 2016

Preliminaries

- Very first question: lectures in English or in French?
- Lectures 1,2,3,4: Claude Marché
  no lecture on January 4th
- 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)
  - final exam E: Thursday, March 1st or 8th, 2017, 16:15,
    same room as the lecture.
  - final mark = \((2E + P + \max(E, P))/4\)
- 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
  Weakest Preconditions Revisited

General Objectives

**Ultimate Goal**

*Verify that software is free of bugs*

Famous software failures:
http://www.cs.tau.ac.il/~nachum/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

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

- **Flight control software of A380**: **Astree** verifies absence of run-time errors (2005, abstract interpretation)
  

- **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/](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
Weakest Preconditions Revisited

Syntax: expressions

\[ e ::= n \quad \text{(integer constants)} \]
\[ x \quad \text{(variables)} \]
\[ e \ op \ e \quad \text{(binary operations)} \]
\[ op ::= + \ | \ - \ | \ * \]
\[ | = \ | \neq \ | < \ | \geq \ | \leq \ | \and \ | \or \]

- 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

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

- 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

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

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** 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

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:

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

Running Example: Demo

Demo with the **Why3** tool


See file `imp_isqrt.mlw`

(This is the tool to use for the project, version 0.87.2)
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

Frame rule:

\[
\frac{\{P\}s\{Q\}}{\{P \land R\}s\{Q \land R\}}
\]

with \( R \) a formula where no variables assigned in \( s \) occur

Consequence rule:

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

- Example: proof of

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

Example: isqrt(42)

Exercise: prove of the triple

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

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 short-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:
    \[
    [n]_{\Sigma} = n \\
    [x]_{\Sigma} = \Sigma(x) \\
    [e_1 \text{ op } e_2]_{\Sigma} = [e_1]_{\Sigma} [\text{op}] [e_2]_{\Sigma}
    \]
- [op] natural semantic of operator $\text{op}$ 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

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

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

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

\[
\Sigma, \text{while } e \text{ do } s \rightsquigarrow \Sigma, \text{skip}
\]
Execution of programs

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

In other words:

\[ \Sigma, s \rightarrow^+ \Sigma', s' \]

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

Running example:

\[ \{ n = 42, \text{count } = ?, \text{sum } = ? \}, \text{ISQRT} \rightarrow^* \{ n = 42, \text{count } = 6, \text{sum } = 49 \}, \text{skip} \]

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 \( \Sigma, s \rightarrow^* \Sigma', \text{skip} \)

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

Semantics of formulas

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

\[
\begin{align*}
\llbracket e \rrbracket_\Sigma & = 1 & \text{if } e \neq 0 \\
\llbracket p_1 \land p_2 \rrbracket_\Sigma & = \llbracket p_1 \rrbracket_\Sigma \land \llbracket p_2 \rrbracket_\Sigma \\
& \vdots
\end{align*}
\]

where semantics of expressions is augmented with

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

Notations:

- \( \Sigma \models p \): the formula \( \llbracket p \rrbracket_\Sigma \) is valid
- \( p \models \): formula \( \llbracket p \rrbracket_\Sigma \) holds in all states \( \Sigma \).

Soundness

Definition (Partial correctness)

Hoare triple \( \{ P \} s \{ Q \} \) is said valid if:
for any states \( \Sigma, \Sigma' \), if

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

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

Weakest liberal precondition

[Diikstra 1975]

Function \text{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

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

$$(I \land (\forall v_1, \ldots, v_k, (((e \neq 0 \land I) \Rightarrow WLP(s, I)) \land ((e = 0 \land I) \Rightarrow Q))[w_i \leftarrow v_j]))$$

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

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

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

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
  Weakest Preconditions Revisited
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**

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

   *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

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

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
    - Weakest Preconditions Revisited

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

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

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

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, CVC4, 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:

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

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:

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

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

Variables:

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

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

Let binding:

\[ \Gamma \vdash t_1 : \tau_1 \quad \{ v \mapsto \tau_1 \} \cdot \Gamma \vdash t_2 : \tau_2 \]

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

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

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)

\[ [\text{val}]_{\Sigma,n} = \text{val} \quad \text{(values)} \]
\[ [X]_{\Sigma,n} = \Sigma(X) \quad \text{if } X : \text{ref} \ \tau \]
\[ [V]_{\Sigma,n} = \Pi(V) \quad \text{if } V : \tau \]
\[ [t_1 \ \text{op} \ t_2]_{\Sigma,n} = [t_1]_{\Sigma,n} \ [\text{op}] \ [t_2]_{\Sigma,n} \]
\[ \text{let } v = t_1 \text{ in } t_2 : \Sigma, n = [t_2]_{\Sigma, ((v=[t_1]_{\Sigma,n}) \cdot n)} \]

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  (pure term)  
| e op e  (binary operation)  
| x := e  (assignment)  
| let v = e in e  (local binding)  
| if e then e else e  (conditional)  
| while e do e  (loop)  
```

- sequence $e_1 ; e_2$ : syntactic sugar for
  
  ```
  let v = e_1 in e_2
  ```

  when $e_1$ has type `unit` and $v$ not used in $e_2$

Toy Examples

```
z := if x >= y then x else y

let v = r in (r := v + 42; v)

while (x := x - 1; x > 0) do ()

while (let v = x in x := x - 1; v > 0) do ()
```
Typing Rules for Expressions

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

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

▶ values do not reduce

Operational Semantics, Continued

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

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

▶ Binary operations
\[
\Sigma, \Pi, e_1 \cdot e_2 \rightsquigarrow \Sigma', \Pi', e'_1 \cdot e_2
\]
\[
\Sigma, \Pi, \text{val}_1 + \text{val}_2 \rightsquigarrow \Sigma, \Pi, \text{val}_1 + \text{val}_2
\]
\[
\text{val} = \text{val}_1 + \text{val}_2
\]
\[
\Sigma, \Pi, \text{val}_1 + \text{val}_2 \rightsquigarrow \Sigma, \Pi, \text{val}
\]
Operational Semantics, Continued

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

- 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
  \[ \text{let } v = ... \text{ in } ... \] 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

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

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

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

Result value in post-conditions

New addition in the specification language:
- keyword \texttt{result} in post-conditions
- denotes the value of the expression executed

Example:
\[ \{ \text{true} \}\]
\[ \text{if } x \geq y \text{ then } x \text{ else } y \]
\[ \{ \text{result} \geq x \land \text{result} \geq y \} \]

Blocking Semantics: Modified Rules

\[
\frac{[P]\Sigma,\Pi \text{ holds}}{\Sigma,\Pi,\text{assert } P \leadsto \Sigma,\Pi,()} \]

\[
\frac{[\square]\Sigma,\Pi \text{ holds}}{\Sigma,\Pi,\text{while } C \text{ invariant } I \text{ do } e \leadsto}
\]
\[\Sigma,\Pi,\text{if } C \text{ then } (e; \text{while } C \text{ invariant } I \text{ do } e) \text{ else } () \]

Important
Execution blocks as soon as an invalid annotation is met

Soundness of a program

Definition
Execution of an expression in a given state is \textit{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 \( P, e \text{ executes safely} \) 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 terms: $\text{WP}(t, Q) = Q[\text{result} \leftarrow t]$
- Let binding: $\text{WP}(\text{let } x = e_1 \text{ in } e_2, Q) = \text{WP}(e_1, \text{WP}(e_2, Q)[x \leftarrow \text{result}])$

Weakest Preconditions, continued

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

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

WP: Exercise

$$\text{WP}(\text{let } v = x \text{ in } (x := x + 1; v), x > \text{result}) = ?$$
Weakest Preconditions, continued

- Conditional
  \[
  \text{WP}(\text{if } e_1 \text{ then } e_2 \text{ else } e_3, Q) = \\
  \text{WP}(e_1, \text{if } \text{result} \text{ then } \text{WP}(e_2, Q) \text{ else } \text{WP}(e_3, Q))
  \]
- Alternative with let: (exercise!)

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

Bibliography


