

Basics of deductive program verification

Bobot François

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

3 décembre 2017

Preliminaries

- ▶ Very first question: lectures in English or in French?
- ▶ Lectures 1,2,3,4: François Bobot
- ▶ Lectures 5,6,7,8: Jean-Marie Madiot
- ▶ one week in january, february: lecture replaced by practical lab, support for project
- ▶ Evaluation:
 - ▶ project P using the Why3 tool (<http://why3.lri.fr>)
 - ▶ final exam E : Monday, March 11th, 2018, 16:15, same room as the lecture.
 - ▶ final mark = $(2E + P + \max(E, P))/4$
- ▶ internships (*stages*)
- ▶ Slides, lectures notes on web page
<http://francois.bobot.eu/mpri2018/>

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/~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)

Deductive verification

- ▶ formal models using expressive logics
- ▶ verification = computer-assisted mathematical proof

Fortunately techniques are cross-fertilizinng

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)
<http://research.microsoft.com/en-us/projects/vcc/>
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)
<http://www.ertos.nicta.com.au/research/l4.verified/>

Other Success Stories at Industry

- ▶ **EDF**: Using Frama-C Eva
- ▶ **Airbus**: Using Frama-C WP
- ▶ **Spark2014**: Verification of a subset of Ada program

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

```

$$\begin{array}{ll} e ::= n & \text{(integer constants)} \\ & | x \quad \text{(variables)} \\ & | e \text{ op } e \quad \text{(binary operations)} \\ \text{op} ::= + \mid - \mid * & \\ & | = \mid \neq \mid < \mid > \mid \leq \mid \geq \\ & | \text{and} \mid \text{or} \end{array}$$

```

- ▶ 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

```

$$\begin{array}{ll} S ::= \text{skip} & \text{(no effect)} \\ & | x := e \quad \text{(assignment)} \\ & | S; S \quad \text{(sequence)} \\ & | \text{if } e \text{ then } S \text{ else } S \quad \text{(conditional)} \\ & | \text{while } e \text{ do } S \quad \text{(loop)} \end{array}$$

```

- ▶ 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`

```
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.

Propositions about programs

- ▶ To formally express properties of programs, we need a *formal specification language* (JML, ACSL, SPARK)
- ▶ We use *standard first-order logic*
- ▶ syntax of formulas:

$$p ::= e \mid p \wedge p \mid p \vee p \mid \neg p \mid p \Rightarrow p \mid \forall v, p \mid \exists v, p$$

- ▶ 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 terminate

Examples

Examples of valid triples for partial correctness:

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

Our running example:

$$\{?n \geq 0\} \text{ISQRT} \{?count^2 \leq n \wedge n < (count + 1)^2\}$$

Running Example: Demo

Demo with the *Why3* tool

- ▶ <http://why3.lri.fr/>
- ▶ Web interface: <http://why3.lri.fr/try/>

See file `imp_isqrt.mlw`

(This is the tool to use for the project, version 1.1.0)

Hoare logic as an Axiomatic Semantics

Original Hoare logic [*~ 1970*]

Axiomatic Semantics of programs

Set of *inference rules* producing triples

$$\frac{}{\{P\}\text{skip}\{P\}}$$

$$\frac{}{\{P[x \leftarrow e]\}x := e\{P\}}$$

$$\frac{\{P\}s_1\{Q\} \quad \{Q\}s_2\{R\}}{\{P\}s_1; s_2\{R\}}$$

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

Hoare Logic, continued

Frame rule:

$$\frac{\{P\}s\{Q\}}{\{P \wedge R\}s\{Q \wedge R\}}$$

with R a formula where no variables assigned in s occur

Consequence rule:

$$\frac{\{P'\}s\{Q'\} \quad \models P \Rightarrow P' \quad \models Q' \Rightarrow Q}{\{P\}s\{Q\}}$$

- ▶ Example: proof of

$$\{x = 1\}x := x + 2\{x = 3\}$$

Hoare Logic, continued

Rules for if and while :

$$\frac{\{P \wedge e \neq 0\}s_1\{Q\} \quad \{P \wedge e = 0\}s_2\{Q\}}{\{P\}\text{if } e \text{ then } s_1 \text{ else } s_2\{Q\}}$$

$$\frac{\{I \wedge e \neq 0\}s\{I\}}{\{I\}\text{while } e \text{ do } s\{I \wedge e = 0\}}$$

- ▶ I is called a *loop invariant*.

Example: isqrt(42)

Exercise: prove of the triple

$$\{n \geq 0\} \text{ ISQRT } \{count^2 \leq n \wedge n < (count + 1)^2\}$$

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 Σ associating to each variable x its current value denoted $\Sigma(x)$.
- ▶ Value of an expression e in some state Σ :
 - ▶ denoted $\llbracket e \rrbracket_\Sigma$
 - ▶ always defined, by the following recursive equations:

$$\begin{aligned}\llbracket n \rrbracket_\Sigma &= n \\ \llbracket x \rrbracket_\Sigma &= \Sigma(x) \\ \llbracket e_1 \text{ op } e_2 \rrbracket_\Sigma &= \llbracket e_1 \rrbracket_\Sigma \llbracket \text{op} \rrbracket \llbracket e_2 \rrbracket_\Sigma\end{aligned}$$

- ▶ $\llbracket \text{op} \rrbracket$ 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 Σ , executing one step of statement s leads to the state Σ' and the remaining statement to execute is s' .
The semantics is defined by the following rules.

$$\frac{}{\Sigma, x := e \rightsquigarrow \Sigma \{x \leftarrow \llbracket e \rrbracket_\Sigma\}, \text{skip}}$$

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

$$\frac{}{\Sigma, (\text{skip}; s) \rightsquigarrow \Sigma, s}$$

Semantics of statements, continued

$$\frac{[[e]]_{\Sigma} \neq 0}{\Sigma, \text{if } e \text{ then } s_1 \text{ else } s_2 \rightsquigarrow \Sigma, s_1}$$

$$\frac{[[e]]_{\Sigma} = 0}{\Sigma, \text{if } e \text{ then } s_1 \text{ else } s_2 \rightsquigarrow \Sigma, s_2}$$

$$\frac{[[e]]_{\Sigma} \neq 0}{\Sigma, \text{while } e \text{ do } s \rightsquigarrow \Sigma, (s; \text{while } e \text{ do } s)}$$

$$\frac{[[e]]_{\Sigma} = 0}{\Sigma, \text{while } e \text{ do } s \rightsquigarrow \Sigma, \text{skip}}$$

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 Σ , reaches state Σ' 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 Σ **terminates** if there is a state Σ' 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 Σ
- ▶ is a logic formula where no program variables appear anymore
- ▶ defined recursively as follows.

$$\begin{aligned} [[e]]_{\Sigma} &= [[e]]_{\Sigma} \neq 0 \\ [[p_1 \wedge p_2]]_{\Sigma} &= [[p_1]]_{\Sigma} \wedge [[p_2]]_{\Sigma} \\ &\vdots \end{aligned}$$

where semantics of expressions is augmented with

$$\begin{aligned} [[v]]_{\Sigma} &= v \\ [[x]]_{\Sigma} &= \Sigma(x) \end{aligned}$$

Notations:

- ▶ $\Sigma \models p$: the formula $[[p]]_{\Sigma}$ is **valid**
- ▶ $\models p$: formula $[[p]]_{\Sigma}$ holds in all states Σ .

Semantics of formulas

Other presentation of the semantics: $\llbracket p \rrbracket_\Sigma$:

- ▶ inline semantic of first-order formula
- ▶ $\llbracket e \rrbracket_{\Sigma, \mathcal{V}}$ with \mathcal{V} mapping of logic variables to integers.
- ▶ defined recursively as follows.

$$\begin{aligned}\llbracket p_1 \wedge p_2 \rrbracket_{\Sigma, \mathcal{V}} &= \begin{cases} \top & \text{if } \llbracket p_1 \rrbracket_{\Sigma, \mathcal{V}} = \top \text{ and } \llbracket p_2 \rrbracket_{\Sigma, \mathcal{V}} = \top \\ \perp & \text{otherwise} \end{cases} \\ \llbracket \forall x. e \rrbracket_{\Sigma, \mathcal{V}} &= \top \text{ si pour tout } v. \llbracket e \rrbracket_{\Sigma, \mathcal{V}[x \leftarrow v]} = \top \\ &\vdots\end{aligned}$$

where semantics of expressions is augmented with

$$\begin{aligned}\llbracket v \rrbracket_{\Sigma, \mathcal{V}} &= \mathcal{V}(v) \\ \llbracket x \rrbracket_{\Sigma, \mathcal{V}} &= \Sigma(x)\end{aligned}$$

Soundness

Definition (Partial correctness)

Hoare triple $\{P\}s\{Q\}$ is said *valid* if:
for any states Σ, Σ' , 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*

$s ::=$	<code>skip</code>	(no effect)
	<code> x := e</code>	(assignment)
	<code> s; s</code>	(sequence)
	<code> if e then s else s</code>	(conditional)
	<code> while e invariant I do s</code>	(annotated loop)

- ▶ The operational semantics is unchanged.

Weakest liberal precondition

[Dijkstra 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:

$$\begin{aligned}\text{WLP}(\text{skip}, Q) &= Q \\ \text{WLP}(x := e, Q) &= Q[x \leftarrow e] \\ \text{WLP}(s_1; s_2, Q) &= \text{WLP}(s_1, \text{WLP}(s_2, Q)) \\ \text{WLP}(\text{if } e \text{ then } s_1 \text{ else } s_2, Q) &= \\ & (e \neq 0 \Rightarrow \text{WLP}(s_1, Q)) \wedge (e = 0 \Rightarrow \text{WLP}(s_2, Q))\end{aligned}$$

Definition of $\text{WLP}(s, Q)$, continued

$$\begin{aligned}\text{WLP}(\text{while } e \text{ invariant } I \text{ do } s, Q) = & \\ I \wedge & \quad (\text{invariant true initially}) \\ \forall v_1, \dots, v_k, & \\ & (((e \neq 0 \wedge I) \Rightarrow \text{WLP}(s, I)) \quad (\text{invariant preserved}) \\ & \wedge ((e = 0 \wedge I) \Rightarrow Q)[w_i \leftarrow v_i] \quad (\text{invariant implies post})\end{aligned}$$

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

Examples

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

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

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 $a + b$.

```
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.

```
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 $\text{div2}(x)$ 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 := 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 $\text{fib}(0) = 0$, $\text{fib}(1) = 1$ and $\text{fib}(n+2) = \text{fib}(n+1) + \text{fib}(n)$. The following program is supposed to compute fib in linear time, the result being stored in y .

```
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

$$\{P\}x := e\{\exists v, e[x \leftarrow v] = x \wedge P[x \leftarrow v]\}$$

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

$$\frac{}{\{P\}x := e\{\exists v, e[x \leftarrow v] = x \wedge P[x \leftarrow v]\}}$$

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

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

previous section	now
expression	term
formula	formula
statement	expression

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* *t*₁ *else* *t*₂

$t ::=$	<i>val</i>	(values, i.e. constants)
	<i>v</i>	(logic variables)
	<i>x</i>	(program variables)
	$t \text{ op } t$	(binary operations)
	<i>if</i> <i>t</i> <i>then</i> <i>t</i> <i>else</i> <i>t</i>	(if-expression)

Local logic variables

We extend the syntax of terms by

$$t ::= \text{let } v = t \text{ in } t$$

Example: approximated cosine

```
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:

$p ::= t$	(boolean term)
$p \wedge p \mid p \vee p \mid \neg p \mid p \Rightarrow p$	(connectives)
$\forall v : \tau, p \mid \exists v : \tau, p$	(quantification)
$\text{let } v = t \text{ in } p$	(local binding)

Typing

- ▶ Types:

$$\tau ::= \text{int} \mid \text{real} \mid \text{bool} \mid \text{unit}$$

- ▶ Typing judgment:

$$\Gamma \vdash t : \tau$$

where Γ 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:

$$\frac{}{\Gamma \vdash n : \text{int}} \quad \frac{}{\Gamma \vdash r : \text{real}}$$

Variables:

$$\frac{}{\Gamma \vdash \text{True} : \text{bool}} \quad \frac{}{\Gamma \vdash \text{False} : \text{bool}}$$

$$\frac{v : \tau \in \Gamma}{\Gamma \vdash v : \tau} \quad \frac{x : \text{ref } \tau \in \Gamma}{\Gamma \vdash x : \tau}$$

Let binding:

$$\frac{\Gamma \vdash t_1 : \tau_1 \quad \{v : \tau_1\} \cdot \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash \text{let } v = t_1 \text{ in } t_2 : \tau_2}$$

- ▶ 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

- ▶ Σ : maps program variables to values (a map)
- ▶ Π : maps logic variables to values (a stack)

$$\begin{aligned} \llbracket \text{val} \rrbracket_{\Sigma, \Pi} &= \text{val} && \text{(values)} \\ \llbracket x \rrbracket_{\Sigma, \Pi} &= \Sigma(x) && \text{if } x : \text{ref } \tau \\ \llbracket v \rrbracket_{\Sigma, \Pi} &= \Pi(v) && \text{if } v : \tau \\ \llbracket t_1 \text{ op } t_2 \rrbracket_{\Sigma, \Pi} &= \llbracket t_1 \rrbracket_{\Sigma, \Pi} \llbracket \text{op} \rrbracket \llbracket t_2 \rrbracket_{\Sigma, \Pi} \\ \llbracket \text{let } v = t_1 \text{ in } t_2 \rrbracket_{\Sigma, \Pi} &= \llbracket t_2 \rrbracket_{\Sigma, (\{v = \llbracket t_1 \rrbracket_{\Sigma, \Pi} \} \cdot \Pi)} \end{aligned}$$

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)
$e \text{ op } e$	(binary operation)
$x := e$	(assignment)
$\text{let } v = e \text{ in } e$	(local binding)
$\text{if } e \text{ then } e \text{ else } e$	(conditional)
$\text{while } e \text{ do } e$	(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:

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

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

Novelties

- ▶ 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

$$\frac{\Sigma, \Pi, e \rightsquigarrow \Sigma', \Pi', e'}{\Sigma, \Pi, x := e \rightsquigarrow \Sigma', \Pi', x := e'}$$

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

► Let binding

$$\frac{\Sigma, \Pi, e_1 \rightsquigarrow \Sigma', \Pi', e'_1}{\Sigma, \Pi, \text{let } v = e_1 \text{ in } e_2 \rightsquigarrow \Sigma', \Pi', \text{let } v = e'_1 \text{ in } e_2}$$

$$\overline{\Sigma, \Pi, \text{let } v = val \text{ in } e \rightsquigarrow \Sigma, \{v = val\} \cdot \Pi, e}$$

Operational Semantics, Continued

► Binary operations

$$\frac{\Sigma, \Pi, e_1 \rightsquigarrow \Sigma', \Pi', e'_1}{\Sigma, \Pi, e_1 + e_2 \rightsquigarrow \Sigma', \Pi', e'_1 + e_2}$$

$$\frac{\Sigma, \Pi, e_2 \rightsquigarrow \Sigma', \Pi', e'_2}{\Sigma, \Pi, val_1 + e_2 \rightsquigarrow \Sigma', \Pi', val_1 + e'_2}$$

$$\frac{val = val_1 + val_2}{\Sigma, \Pi, val_1 + val_2 \rightsquigarrow \Sigma, \Pi, val}$$

Operational Semantics, Continued

► Conditional

$$\frac{\Sigma, \Pi, c \rightsquigarrow \Sigma', \Pi', c'}{\Sigma, \Pi, \text{if } c \text{ then } e_1 \text{ else } e_2 \rightsquigarrow \Sigma', \Pi', \text{if } c' \text{ then } e_1 \text{ else } e_2}$$

$$\overline{\Sigma, \Pi, \text{if } True \text{ then } e_1 \text{ else } e_2 \rightsquigarrow \Sigma, \Pi, e_1}$$

$$\overline{\Sigma, \Pi, \text{if } False \text{ then } e_1 \text{ else } e_2 \rightsquigarrow \Sigma, \Pi, e_2}$$

► Loop

$$\frac{\Sigma, \Pi, \text{while } c \text{ do } e \rightsquigarrow}{\Sigma, \Pi, \text{if } c \text{ then } (e; \text{while } c \text{ do } e) \text{ else } ()}$$

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.:

$$\frac{v_1, v_2 \text{ fresh}}{\Sigma, \Pi, e_1 + e_2 \rightsquigarrow \Sigma, \Pi, \text{let } v_1 = e_1 \text{ in let } v_2 = e_2 \text{ in } v_1 + v_2}$$

- 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

```
e ::= assert p                (assertion)
      | while e invariant I do e (annotated loop)
```

Toy Examples

```
z := if x ≥ y then x else y ;
assert z ≥ x ∧ z ≥ y
```

```
while (x := x - 1; x > 0)
  invariant x ≥ 0 do ();
assert (x = 0)
```

```
while (let v = x in x := x - 1; v > 0)
  invariant x ≥ -1 do ();
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

$$\frac{\llbracket P \rrbracket_{\Sigma, \Pi} \text{ holds}}{\Sigma, \Pi, \text{assert } P \rightsquigarrow \Sigma, \Pi, ()}$$

$$\frac{\llbracket I \rrbracket_{\Sigma, \Pi} \text{ holds}}{\Sigma, \Pi, \text{while } c \text{ invariant } I \text{ do } e \rightsquigarrow \Sigma, \Pi, \text{if } c \text{ then } (e; \text{while } c \text{ invariant } I \text{ do } e) \text{ else } ()}$$

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 *safe* if it does not block: either terminates on a value or runs infinitely.

Definition

A triple $\{P\}e\{Q\}$ is valid if for any state Σ, Π satisfying P , e *executes safely* in Σ, Π , and if it terminates, the final state satisfies Q

Weakest Preconditions Revisited

Goal:

- ▶ construct a new calculus $WP(e, Q)$

Expected property: in any state satisfying $WP(e, Q)$,

- ▶ e is guaranteed to execute safely
- ▶ if it terminates, Q holds in the final state

New Weakest Precondition Calculus

- ▶ Pure terms:

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

- ▶ Let binding:

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

Weakest Preconditions, continued

► Assignment:

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

► Alternative:

$$\begin{aligned}\text{WP}(x := e, Q) &= \text{WP}(\text{let } v = e \text{ in } x := v, Q) \\ \text{WP}(x := t, Q) &= Q[\text{result} \leftarrow (); x \leftarrow t]\end{aligned}$$

WP: Exercise

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

Weakest Preconditions, continued

► Conditional

$$\begin{aligned}\text{WP}(\text{if } e_1 \text{ then } e_2 \text{ else } e_3, Q) = \\ \text{WP}(e_1, \text{if } \text{result} \text{ then } \text{WP}(e_2, Q) \text{ else } \text{WP}(e_3, Q))\end{aligned}$$

► Alternative with let: (exercise!)

Weakest Preconditions, continued

► Assertion

$$\begin{aligned}\text{WP}(\text{assert } P, Q) &= P \wedge Q \\ &= P \wedge (P \Rightarrow Q)\end{aligned}$$

(second version useful in practice)

► While loop

$$\begin{aligned}\text{WP}(\text{while } c \text{ invariant } I \text{ do } e, Q) = \\ I \wedge \\ \forall \vec{v}, (I \Rightarrow \text{WP}(c, \text{if } \text{result} \text{ then } \text{WP}(e, I) \text{ else } Q))[\vec{w}_i \leftarrow v_i]\end{aligned}$$

where w_1, \dots, w_k is the set of assigned variables in expressions c and e and v_1, \dots, v_k are fresh logic variables

Soundness of WP

Lemma (Preservation by Reduction)

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

Proof: predicate induction of \rightsquigarrow .

Lemma (Progress)

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

Proof: structural induction of e .

Corollary (Soundness)

If $\Sigma, \Pi \models \text{WP}(e, Q)$ then

- ▶ e executes safely in Σ, Π .
- ▶ if execution terminates, Q holds in the final state

Assume and Check

$e ::=$ $\text{assert } p$ (assertion)
 $\quad \quad \quad | \text{assume } p$ (assumption)
 $\quad \quad \quad | \text{check } p$ (check)

Assume and Check

$$\begin{array}{lcl} \frac{\llbracket P \rrbracket_{\Sigma, \Pi} \text{ holds}}{\Sigma, \Pi, \text{assert } P \rightsquigarrow \Sigma, \Pi, ()} & \text{WP}(\text{assert } P, Q) = & P \&\& Q \\ & & = P \wedge (P \Rightarrow Q) \\ \frac{\llbracket P \rrbracket_{\Sigma, \Pi} \text{ holds}}{\Sigma, \Pi, \text{assume } P \rightsquigarrow \Sigma, \Pi, ()} & \text{WP}(\text{assume } P, Q) = & P \Rightarrow Q \\ \frac{}{\Sigma, \Pi, \text{check } P \rightsquigarrow \Sigma, \Pi, ()} & \text{WP}(\text{check } P, Q) = & P \wedge Q \end{array}$$

Assert, Assume and Check

$$\begin{array}{lcl} \frac{\llbracket P \rrbracket_{\Sigma, \Pi} \text{ holds}}{\Sigma, \Pi, \text{assert } P \rightsquigarrow \Sigma, \Pi, ()} & \frac{\llbracket P \rrbracket_{\Sigma, \Pi} \text{ doesn't hold}}{\Sigma, \Pi, \text{assert } P \rightsquigarrow \text{Blocked}} \\ \frac{\llbracket P \rrbracket_{\Sigma, \Pi} \text{ holds}}{\Sigma, \Pi, \text{assume } P \rightsquigarrow \Sigma, \Pi, ()} & \frac{\llbracket P \rrbracket_{\Sigma, \Pi} \text{ doesn't hold}}{\Sigma, \Pi, \text{assume } P \rightsquigarrow \text{Infeasible}} \\ \frac{}{\Sigma, \Pi, \text{check } P \rightsquigarrow \Sigma, \Pi, ()} & \frac{}{\Sigma, \Pi, \text{check } P \rightsquigarrow \Sigma, \Pi, ()} \end{array}$$

States that leads to **Infeasible** are not taken into account.

Exercise: Restate and prove the soundness theorem

Bibliography

- [Cook\(1978\)](#) S. A. Cook. Soundness and completeness of an axiom system for program verification. *SIAM Journal on Computing*, 7(1):70–90, 1978. doi: 10.1137/0207005.
- [Cousot\(1990\)](#) P. Cousot. Methods and logics for proving programs. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 841–993. North-Holland, 1990.
- [Dijkstra\(1975\)](#) E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18:453–457, August 1975. ISSN 0001-0782. doi: 10.1145/360933.360975.

Bibliography

- [Floyd\(1967\)](#) R. W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposia in Applied Mathematics*, pages 19–32, Providence, Rhode Island, 1967. American Mathematical Society.
- [Hoare\(1969\)](#) C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580 and 583, Oct. 1969.
- [Plotkin\(2004\)](#) G. D. Plotkin. The origins of structural operational semantics. *Journal of Logic and Algebraic Programming*, 60–61:3–15, 2004. doi: 10.1016/j.jlap.2004.03.009.