# 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



## 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)
  `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/`

## Outline



## Syntax: expressions

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

- 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

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

- 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*
- We use *standard first-order logic*
- syntax of formulas:

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

- $v$ : *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 terminates

## Examples

Examples of valid triples for partial correctness:

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

Our running example:

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

## 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 0.87.2)

## Hoare logic as an Axiomatic Semantics

**Original Hoare logic** *[~ 1970]*

*Axiomatic Semantics* of programs

Set of *inference rules* producing triples

$$\frac{}{\{P\}\texttt{skip}\{P\}}$$

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

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

- 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:

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

- Example: proof of

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

## Hoare Logic, continued

Rules for if and while :

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

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

- $I$ is called a *loop invariant*.

## Example: isqrt(42)

Exercise: prove of the triple

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

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:

$$\begin{array}{rcl} [\![n]\!]_\Sigma & = & n \\ [\![x]\!]_\Sigma & = & \Sigma(x) \\ [\![e_1 \; op \; e_2]\!]_\Sigma & = & [\![e_1]\!]_\Sigma \; [\![op]\!] \; [\![e_2]\!]_\Sigma \end{array}$$

  - $[\![op]\!]$ 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

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

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.

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

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

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

## Semantics of statements, continued

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

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

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

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

## Execution of programs

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

In other words:

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

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

Running example:

$$\{n = 42, count = ?, sum = ?\}, ISQRT \leadsto^* \\ \{n = 42, count = 6, sum = 49\}, \texttt{skip}$$

## 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 $\Sigma, s \leadsto^* \Sigma', \texttt{skip}$

- 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.

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

where semantics of expressions is augmented with

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

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

- $\Sigma, s \leadsto^* \Sigma', \texttt{skip}$ 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*

| | | | |
|---|---|---|---|
| $s$ | $::=$ | `skip` | (no effect) |
| | $\mid$ | $x := e$ | (assignment) |
| | $\mid$ | $s; s$ | (sequence) |
| | $\mid$ | `if` $e$ `then` $s$ `else` $s$ | (conditional) |
| | $\mid$ | `while` $e$ `invariant` $I$ `do` $s$ | (annotated loop) |

- The operational semantics is unchanged.

## Weakest liberal precondition

*[Dijkstra 1975]*

Function $\mathrm{WLP}(s, Q)$ :

- $s$ 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 $\mathrm{WLP}(s, Q)$

Recursive definition:

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

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

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

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

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

$$
\begin{array}{l}
\mathrm{WLP}(\texttt{while } y > 0 \texttt{ invariant } even(y) \texttt{ do } y := y - 2, even(y)) \equiv \\
\quad even(y) \wedge \\
\quad \forall v, ((v > 0 \wedge even(v)) \Rightarrow even(v - 2)) \\
\qquad \wedge((v \leq 0 \wedge even(v)) \Rightarrow even(v))
\end{array}
$$

## Soundness

**Theorem (Soundness)**

*For all statement $s$ and formula $Q$, $\{\mathrm{WLP}(s, Q)\}s\{Q\}$ 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 $P \Rightarrow \mathrm{WLP}(s, Q)$.

This is basically what Why3 does

## Outline

## 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 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 $x^n$.

```
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 $y$.

```
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 $Pre \Rightarrow \mathrm{WLP}(s, Post)$ 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 $\texttt{if}\ b\ \texttt{then}\ t_1\ \texttt{else}\ t_2$

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

## Local logic variables

We extend the syntax of terms by

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

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:

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

## Typing

- Types:

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

- Typing judgment:

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

where $\Gamma$ maps identifiers to types:
  - either $v : \tau$ (logic variable, immutable)
  - either $x : \texttt{ref } \tau$ (program variable, mutable)

**Important**

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

## Typing rules

Constants:

$$\frac{}{\Gamma \vdash n : \texttt{int}} \qquad \frac{}{\Gamma \vdash r : \texttt{real}}$$

$$\frac{}{\Gamma \vdash \mathit{True} : \texttt{bool}} \qquad \frac{}{\Gamma \vdash \mathit{False} : \texttt{bool}}$$

Variables:

$$\frac{v : \tau \in \Gamma}{\Gamma \vdash v : \tau} \qquad \frac{x : \texttt{ref } \tau \in \Gamma}{\Gamma \vdash x : \tau}$$

Let binding:

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

- 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)

$$
\begin{array}{rcll}
[\![val]\!]_{\Sigma,\Pi} & = & val & \text{(values)} \\
[\![x]\!]_{\Sigma,\Pi} & = & \Sigma(x) & \text{if } x : \texttt{ref } \tau \\
[\![v]\!]_{\Sigma,\Pi} & = & \Pi(v) & \text{if } v : \tau \\
[\![t_1\ op\ t_2]\!]_{\Sigma,\Pi} & = & [\![t_1]\!]_{\Sigma,\Pi}\ [\![op]\!]\ [\![t_2]\!]_{\Sigma,\Pi} & \\
[\![\texttt{let } v = t_1 \texttt{ in } t_2]\!]_{\Sigma,\Pi} & = & [\![t_2]\!]_{\Sigma,(\{v=[\![t_1]\!]_{\Sigma,\Pi}\}\cdot\Pi)} &
\end{array}
$$

**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

$$
\begin{array}{rcll}
e & ::= & t & \text{(pure term)} \\
 & | & e \; op \; e & \text{(binary operation)} \\
 & | & x := e & \text{(assignment)} \\
 & | & \mathtt{let}\; v = e \;\mathtt{in}\; e & \text{(local binding)} \\
 & | & \mathtt{if}\; e \;\mathtt{then}\; e \;\mathtt{else}\; e & \text{(conditional)} \\
 & | & \mathtt{while}\; e \;\mathtt{do}\; e & \text{(loop)}
\end{array}
$$

- sequence $e_1 ; e_2$ : syntactic sugar for

  $$\mathtt{let}\; v = e_1 \;\mathtt{in}\; e_2$$

  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:

$$\frac{x : \mathtt{ref}\ \tau \in \Gamma \qquad \Gamma \vdash e : \tau}{\Gamma \vdash x := e : \mathtt{unit}}$$

Let binding:

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \{v : \tau_1\} \cdot \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \mathtt{let}\ v = e_1\ \mathtt{in}\ e_2 : \tau_2}$$

Conditional:

$$\frac{\Gamma \vdash c : \mathtt{bool} \qquad \Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \mathtt{if}\ c\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 : \tau}$$

Loop:

$$\frac{\Gamma \vdash c : \mathtt{bool} \qquad \Gamma \vdash e : \mathtt{unit}}{\Gamma \vdash \mathtt{while}\ c\ \mathtt{do}\ e : \mathtt{unit}}$$

## 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

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

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

- Let binding

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

$$\frac{}{\Sigma, \Pi, \mathtt{let}\ v = val\ \mathtt{in}\ e \rightsquigarrow \Sigma, \{v = val\} \cdot \Pi, e}$$

## 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

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

$$\frac{}{\Sigma, \Pi, \texttt{if } \mathit{True} \texttt{ then } e_1 \texttt{ else } e_2 \rightsquigarrow \Sigma, \Pi, e_1}$$

$$\frac{}{\Sigma, \Pi, \texttt{if } \mathit{False} \texttt{ then } e_1 \texttt{ else } e_2 \rightsquigarrow \Sigma, \Pi, e_2}$$

- Loop

$$\frac{}{\begin{array}{l}\Sigma, \Pi, \texttt{while } c \texttt{ do } e \rightsquigarrow \\ \qquad \Sigma, \Pi, \texttt{if } c \texttt{ then } (e; \texttt{while } c \texttt{ do } e) \texttt{ else } ()\end{array}}$$

## Context Rules versus Let Binding

Remark: most of the context rules can be avoided

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

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

- 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

$$\begin{array}{lcll} e & ::= & \texttt{assert } p & \text{(assertion)} \\ & | & \texttt{while } e \texttt{ invariant } I \texttt{ do } e & \text{(annotated loop)} \end{array}$$

## 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

$$\frac{[\![P]\!]_{\Sigma,\Pi} \text{ holds}}{\Sigma,\Pi,\texttt{assert } P \rightsquigarrow \Sigma,\Pi,()}$$

$$\frac{[\![I]\!]_{\Sigma,\Pi} \text{ holds}}{\begin{array}{l}\Sigma,\Pi,\texttt{while } c \texttt{ invariant } I \texttt{ do } e \rightsquigarrow \\ \quad \Sigma,\Pi,\texttt{if } c \texttt{ then } (e;\texttt{while } c \texttt{ invariant } I \texttt{ do } e) \texttt{ else } ()\end{array}}$$

**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 $Q$

## Weakest Preconditions Revisited

Goal:

- construct a new calculus $\mathrm{WP}(e, Q)$

Expected property: in any state satisfying $\mathrm{WP}(e, Q)$,

- $e$ 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:

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

## Weakest Preconditions, continued

- Assignment:

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

- Alternative:

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

## WP: Exercise

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

## Weakest Preconditions, continued

- Conditional

$$\mathrm{WP}(\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3, Q) = \\ \quad \mathrm{WP}(e_1, \texttt{if } \mathit{result} \texttt{ then } \mathrm{WP}(e_2, Q) \texttt{ else } \mathrm{WP}(e_3, Q))$$

- Alternative with let: (exercise!)

## Weakest Preconditions, continued

- Assertion

$$\begin{array}{rcl} \mathrm{WP}(\texttt{assert } P, Q) & = & P \land Q \\ & = & P \land (P \Rightarrow Q) \end{array}$$

(second version useful in practice)

- While loop

$$\mathrm{WP}(\texttt{while } c \texttt{ invariant } I \texttt{ do } e, Q) = \\ \quad I \land \\ \quad \forall \vec{v}, (I \Rightarrow \mathrm{WP}(c, \texttt{if } \mathit{result} \texttt{ then } \mathrm{WP}(e, I) \texttt{ else } Q))[w_i \leftarrow v_i]$$

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

## Soundness of WP

**Lemma (Preservation by Reduction)**

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

Proof: predicate induction of $\rightsquigarrow$.

**Lemma (Progress)**

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

Proof: structural induction of $e$.

**Corollary (Soundness)**

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

- $e$ *executes safely in* $\Sigma, \Pi$.
- *if execution terminates,* $Q$ *holds in the final state*

## 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.