Basics of Deductive Program Verification

Claude Marché (François bobot)

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

December 11th, 2024

Preliminaries

Very first question

Lectures in English or in French?

- Schedule on the Web page https://marche.gitlabpages.inria.fr/lecture-deductive-verif/
- Lectures 1,2,3,4: François Bobot
- Lectures 5,6,7,8: Jean-Marie Madiot
- Evaluation:
  - project *P* using the Why3 tool (http://why3.lri.fr)
  - final exam *E*: date to decide
  - final mark = Max((2/3 E + 1/3 P), (3/4 E + 1/4 P))
- Project:
  - provided end of December
  - due date around mid-February
- Internships (*stages*)

Outline

General Objectives

Ultimate Goal

Verify that software is free of bugs

Famous software failures: many web sites, e.g.

- http://www.cs.tau.ac.il/~nachumd/horror.html
- http://catless.ncl.ac.uk/Risks/1/1#subj4

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

Some general approaches to Verification

Refinement

- Formal models
- Code derived from model, correct by construction

A long time before success

Turing Text

Friday, 24th June [1949]

Checking a large routine by Dr A. Turing.

How can one check a routine in the sense of making sure that it is right?

In order that the man who checks may not have too difficult a task the programmer should make a number of definite assertions which can be checked individually, and from which the correctness of the whole programme easily follows.

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)
  `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

- Frama-C
  - EDF: abstract interpretation
  - Airbus: deductive verification
  - Thales: deductive verification
- Spark/Ada: Verification of Ada programs
  `https://www.adacore.com/industries`

Remark

The two above use Why3 internally

Outline

Proposition logic in a nutshell

- Syntax:

$$
\begin{array}{rcll}
\varphi & ::= & \bot \mid \top \mid A, B & \text{(atoms)} \\
 & \mid & \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \neg\varphi & \\
 & \mid & \varphi \rightarrow \varphi \mid \varphi \leftrightarrow \varphi &
\end{array}
$$

- Semantics, models: truth tables

φ is satisfiable if it has a model
φ is valid if true in all models
(equivalently $\neg\varphi$ is not satisfiable)

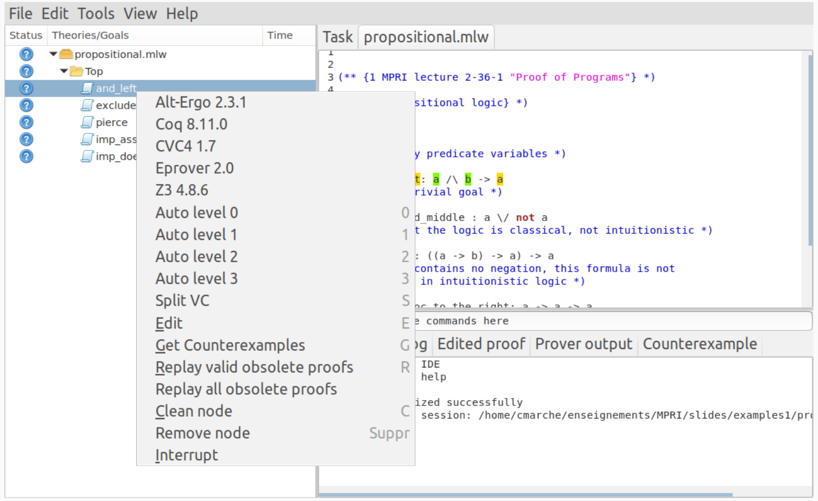
SAT is *decidable* $\rightsquigarrow$ SAT solvers

Demo with Why3

`$ why3 ide propositional.mlw`

Notice that Why3 indeed queries solvers for satisfiability of $\neg\varphi$

Focus on the “Tools” menu of Why3



First-order logic in a nutshell

- Syntax:

$$
\begin{array}{lcll}
\varphi & ::= & \cdots & \\
 & \mid & P(t,\dots,t) & \text{(predicates)} \\
 & \mid & \forall x.\ \varphi \mid \exists x.\ \varphi & \\
t & ::= & x & \text{variables} \\
 & \mid & f(t,\dots,t) & \text{(function symbols)}
\end{array}
$$

- Semantics: models must interpret variables
- Satisfiability *undecidable*, but still *semi-decidable*: there exists complete systems of deduction rules (sequent calculus, natural deduction, superposition calculus)
- Examples of solvers: E, Spass, Vampire

  Implement *refutationally complete* procedure:
  Eventually answer ’unsat’ for unsatisfiable formula

Demo with Why3

`first-order.mlw`

Notice that Why3 logic is *typed*, and application is curryied

Logic Theories

- *Theory* = set of formulas (called *theorems*) closed by logical consequence
- *Axiomatic Theory* = set of formulas generated by axioms (or axiom schemas)
- *Consistent Theory*

  for no P, P and $\neg P$ are both theorems
  equivalently: ’false’ is not a theorem
  equivalently: the theory has models
- *Consistent Axiomatization*
  ’false’ is not derivable

Logic Theories

Theory alternative definition by interpretation:

- Associate a mathematical set for each sort
- Associate a mathematical function for each symbol of the theory
- Given a model of the variables and uninterpreted symbols, an interpretation is defined inductively

Theory of Equality

$$\forall x.\ x = x$$

$$\forall x, y.\ x = y \rightarrow y = x$$

$$\forall x, y, z.\ x = y \wedge y = z \rightarrow x = z$$

(congruence) for all function symbols f of arity k:

$$\forall x_1, y_1 \dots, x_k, y_k.\ x_1 = y_1 \wedge \dots \wedge x_k = y_k \rightarrow f(x_1, \dots, x_k) = f(y_1, \dots, y_k)$$

and for all predicates p of arity k:

$$\forall x_1, y_1 \dots, x_k, y_k.\ x_1 = y_1 \wedge \dots \wedge x_k = y_k \rightarrow p(x_1, \dots, x_k) \rightarrow p(y_1, \dots, y_k)$$

Theory of Equality, Continued

$$\forall x.\ x = x$$

$$\forall x, y.\ x = y \rightarrow y = x$$

$$\forall x, y, z.\ x = y \wedge y = z \rightarrow x = z$$

(congruence) ...

- General first-order deduction bad in such a case $\rightsquigarrow$ dedicated methods
  - paramodulation
  - congruence closure (for quantifier-free fragment)
- SMT solvers (Alt-Ergo, CVC4, Z3) implement dedicated (semi-)decision procedures

Demo with Why3

`equality.mlw`

Theories Continued

Theory of a given model

= formulas true in this model

- Central example: theory of linear integer arithmetic, i.e. formulas using $+$ and $\leq$
  - First-order theory is known to be decidable (Presburger)
  - SMT solvers typically implement a procedure for the existential fragment
- Also: theory of (non-linear) real arithmetic is decidable (Tarski)

Non-linear Integer Arithmetic

(a.k.a. Peano Arithmetic)

First-Order Integer Arithmetic

All valid first-order formulas on integers with $+$, $\times$ and $\leq$

- This theory is not even semi-decidable
- SMT solvers implement incomplete satisfiability checks: if solver answers 'unsat' then it is unsatisfiable

Demo with Why3

`arith.mlw`

Summary of prover limitations

- Superposition solvers (E, Spass,)
  - do not support well theories except equality
  - do quite well with quantifiers
- SMT solvers (Alt-Ergo, CVC4, CVC5 Z3)
  - several theories are built-in
  - weaker with quantifiers
- None support reasoning by induction
  - Strengthening for existential quantification not used by default in CVC5

Outline

IMP language

IMP language

A very basic imperative programming language

- only global variables
- only integer values for expressions
- basic statements:
  - assignment `x <- e`
  - sequence $s_1 ; s_2$
  - conditionals `if` e `then` s_1 `else` s_2
  - loops `while` e `do` s
  - no-op `skip`

Formal Contracts

General form of a program:

Contract

- *precondition*: expresses what is assumed before running the program
- *post-condition*: expresses what is supposed to hold when program exits

Demo with Why3

`contracts.mlw`

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:

- $\{x = 1\}x \texttt{<-}\ x + 2\{x = 3\}$
- $\{x = y\}x \texttt{<-}\ x + y\{x = 2 * y\}$
- $\{\exists v.\ x = 4 * v\}x \texttt{<-}\ x + 42\{\exists w.\ x = 2 * w\}$
- $\{true\}\texttt{while 1 do skip}\{false\}$

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.

See file `imp_isqrt.mlw`

Hoare logic as an Axiomatic Semantics

Original Hoare logic *[∼ 1970]*

Axiomatic Semantics of programs

Set of *inference rules* producing triples

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

$$\frac{}{\{P[x \leftarrow e]\}x \texttt{<-}\ e\{P\}}$$

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

- 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

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

Proof of the example

$$\frac{\dfrac{}{\{x + 2 = 3\}x \leftarrow x + 2\{x = 3\}} \qquad \begin{array}{l}\models x = 1 \rightarrow x + 2 = 3 \\ \models x = 3 \rightarrow x = 3\end{array}}{\{x = 1\}x \leftarrow x + 2\{x = 3\}}$$

Hoare Logic, continued

Rules for if and while :

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

$$\frac{\{I \wedge e\}s\{I\}}{\{I\}\texttt{while } e \texttt{ do } s\{I \wedge \neg e\}}$$

I is called a *loop invariant*

Informal justification of the while rule

$$\frac{\{I \wedge e\}s\{I\}}{\{I\}\texttt{while } e \texttt{ do } s\{I \wedge \neg e\}}$$

I	invariant initially valid
$I \wedge e$	condition assumed true
s	execution of loop body
I	invariant re-established
$I \wedge e$	condition assumed true
s	execution of loop body
I	invariant re-established
$\vdots$	any number of iterations
I	invariant re-established
$I \wedge \neg e$	loop exits when condition false

Example: isqrt(42)

Exercise: prove of the triple

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

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 $[\![e]\!]_\Sigma$
  - 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 $\neq 0$ for true).

Semantics of statements

Semantics of statements: defined by judgment

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

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.

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

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

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

Semantics of statements, continued

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

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

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

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

Execution of programs

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

In other words:

$$\Sigma, s \overset{*}{\leadsto} \Sigma', s'$$

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

Running example:

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

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 Σ *terminates* if there is a state Σ' 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

- $[\![p]\!]_{\Sigma,\mathcal{V}}$ denotes the semantics of formula p in program state Σ and mapping $\mathcal{V}$ of logic variables to integers
- defined recursively, e.g.

$$\begin{aligned}
[\![p_1 \wedge p_2]\!]_{\Sigma,\mathcal{V}} &= \begin{cases} \top & \text{if } [\![p_1]\!]_{\Sigma,\mathcal{V}} = \top \text{ and } [\![p_2]\!]_{\Sigma,\mathcal{V}} = \top \\ \bot \end{cases} \\
[\![\forall v.e]\!]_{\Sigma,\mathcal{V}} &= \top \text{ if for all } n.\ [\![e]\!]_{\Sigma,\mathcal{V}[v \leftarrow n]} = \top \\
[\![v]\!]_{\Sigma,\mathcal{V}} &= \mathcal{V}(v) \\
[\![x]\!]_{\Sigma,\mathcal{V}} &= \Sigma(x)
\end{aligned}$$

Notations:

- $\Sigma \models p$: the formula p is valid in Σ i.e. $[\![p]\!]_{\Sigma,\emptyset}$ is $\top$
- $\models p$: formula $[\![p]\!]_{\Sigma,\emptyset}$ holds in all states Σ.

Soundness

Definition (Partial correctness)

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

- $\Sigma, s \rightsquigarrow^* \Sigma', \mathtt{skip}$ 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.

Digression: Completeness of Hoare Logic

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": representability of any recursive function
Yet, this does not provide an effective recipe to discover suitable loop invariants (see also the theory of abstract interpretation *[Cousot, 1990]*)

Annotated Programs

Goal

Add automation to the Hoare logic approach

We augment IMP with *explicit loop invariants*

$$\mathtt{while}\ e\ \mathtt{invariant}\ I\ \mathtt{do}\ s$$

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:

$$
\begin{array}{rcl}
\mathrm{WLP}(\texttt{skip}, Q) & = & Q \\
\mathrm{WLP}(x \texttt{<-} e, Q) & = & Q[x \leftarrow e] \\
\mathrm{WLP}(s_1; s_2, Q) & = & \mathrm{WLP}(s_1, \mathrm{WLP}(s_2, Q)) \\
\mathrm{WLP}(\texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2, Q) & = & \\
(e \rightarrow \mathrm{WLP}(s_1, Q)) & \wedge & (\neg e \rightarrow \mathrm{WLP}(s_2, Q))
\end{array}
$$

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

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

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

$$\mathrm{WLP}(x \texttt{<-} x + y, x = 2y) \equiv x + y = 2y$$

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

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 the goal that Why3 produces

Outline

Beyond IMP and classical Hoare Logic

Extended 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
- *No difference between expressions and statements anymore*

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 $+, -, \times$ (no division)
- reals: type `real`, operators $+, -, \times$ (no division)
- Comparisons of integers or reals, returning a boolean
- "if-expression": written `if` b `then` t_1 `else` t_2

t	$::=$	val	(values, i.e. constants)
	$\mid$	v	(logic variables)
	$\mid$	x	(program variables)
	$\mid$	$t \ op \ t$	(binary operations)
	$\mid$	`if` t `then` t `else` t	(if-expression)

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 (inc. Alt-Ergo, CVC4, Z3) typically support such a typed logic
- 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

It is (typed) first-order logic, as in previous slides, but also with 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:

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

- Typing judgment:

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

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

Important

- a mutable variable is not a value (it is not a “reference” value)
- as such, 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:

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

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, unlike OCaml, does not require to write !x for mutable variables

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)

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

Warning

Semantics is a partial function, it is not defined on ill-typed formulas

Common notation for formulas

$\Sigma, \pi \models \varphi$ means $[\![\varphi]\!]_{\Sigma,\pi} = \mathrm{true}$

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 of IMP 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

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

- sequence $e_1; e_2$: syntactic sugar for

$$\texttt{let } v = e_1 \texttt{ in } e_2$$

when e_1 has type $\texttt{unit}$ 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)
      (* (--x > 0) in C *)
  do ()

while (let v = x in x <- x - 1; v > 0)
      (* (x-- > 0) in C *)
  do ()
```

Typing Rules for Expressions

Assignment:

$$\frac{x : \texttt{mut } \tau \in \Gamma \qquad \Gamma \vdash e : \tau}{\Gamma \vdash x \leftarrow e : \texttt{unit}}$$

Let binding:

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

Conditional:

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

Loop:

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

Operational Semantics

Novelty w.r.t. IMP

Need to precise the order of evaluation: left to right
(e.g. $x \leftarrow 0; ((x \leftarrow 1); 2) + x) = 2$ or 3 ?)

- one-step execution has the form

$$\Sigma, \pi, e \rightsquigarrow \Sigma', \pi', e'$$

π is the *stack of local variables*

- 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

$$\frac{\Sigma, \pi, e_1 \rightsquigarrow \Sigma', \pi', e_1'}{\Sigma, \pi, \texttt{let } v = e_1 \texttt{ in } e_2 \rightsquigarrow \Sigma', \pi', \texttt{let } v = e_1' \texttt{ in } e_2}$$

$$\frac{}{\Sigma, \pi, \texttt{let } v = val \texttt{ in } e \rightsquigarrow \Sigma, \{v = val\} \cdot \pi, e}$$

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

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

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

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

- 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

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

Toy Examples

```
z <- if x >= y then x else y ;
assert (z >= x /\ z >= y)
```

```
while (x <- x - 1; x > 0)
      (* (--x > 0) in C *)
  invariant x >= 0 do ();
assert (x = 0)
```

```
while (let v = x in x <- x - 1; v > 0)
      (* (x-- > 0) in C *)
  invariant x >= -1 do ();
assert (x = -1)
```

Blocking Semantics: Modified Rules

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

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

Important remark

Execution blocks as soon as an invalid annotation is met

Definition (Safety of execution)

Execution of an expression in a given state is *safe* if it does not block: either terminates on a value or runs infinitely.

Hoare triples: result value in post-conditions

New addition in the logic 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 }
```

Hoare triples: Soundness

Definition (validity of a triple)

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

Difference with historical Hoare triples

Validity of a triple now implies safety of its execution, even if it does not terminate

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

Difference with historical WLP calculus

This calculus is no more "liberal", the computed precondition guarantees safety of execution, even if it does not terminate

New Weakest Precondition Calculus

Pure expressions (i.e. without side-effects, a.k.a. "terms")

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

'let' binding

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

Reminder: sequence is a particular case of 'let'

$$\mathrm{WP}((e_1; e_2), Q) = \mathrm{WP}(e_1, \mathrm{WP}(e_2, Q))$$

Weakest Preconditions, continued

- Assignment:

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

- Alternative:

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

WP: Exercise

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

$$\begin{array}{rl} & \mathrm{WP}(\texttt{let } v = x \texttt{ in } (x \texttt{ <- } x + 1; v), x > result) \\ = & \mathrm{WP}(x, (\mathrm{WP}((x \texttt{ <- } x + 1; v), x > result)[v \leftarrow result])) \\ = & \mathrm{WP}(x, (\mathrm{WP}(x \texttt{ <- } x + 1, \mathrm{WP}(v, x > result)))[v \leftarrow result])) \\ = & \mathrm{WP}(x, (\mathrm{WP}(x \texttt{ <- } x + 1, x > v))[v \leftarrow result])) \\ = & \mathrm{WP}(x, (x + 1 > v)[v \leftarrow result])) \\ = & \mathrm{WP}(x, (x + 1 > result)) \\ = & x + 1 > x \end{array}$$

Weakest Preconditions, continued

- Conditional

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

- Alternative with let: (exercise!)

Weakest Preconditions, continued

- Assertion

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

(second version useful in practice)

- While loop

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

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 Σ', π, 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 Σ, π.*
- *if execution terminates, Q holds in the final state*

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_euclidean_div.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 invariants 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 (original Floyd rule for assignment)

1. *Prove that the triple*

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

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 Floyd rule

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

3. *Show that the triple* $\{P[x \leftarrow e]\}x \leftarrow e\{P\}$ *can be proved with the new set of rules.*

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.