Basics of Deductive Program Verification

Claude Marché

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

3 décembre 2017

Preliminaries

Very first question
Lectures in English or in French?

Schedule on the Web page
http://www.lri.fr/~marche/MPRI-2-36-1/

Lectures 1,2,3,4: Claude Marché
Lectures 5,6,7,8: Jean-Marie Madiot
January 28th: lecture replaced by practical lab, support for project. Project due on February 7th.

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 = $\frac{2E + P + \max(E, P)}{4}$
internships (stages)

Outline

Introduction, Short History
Preliminary on Automated Deduction
Classical Propositional Logic
First-order logic
Logic Theories
Limitations of Automatic Provers
Introduction to Deductive Verification
Formal contracts
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

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)

Other Famous Success Stories
  http://www.astree.ens.fr/
- Microsoft's hypervisor: using Microsoft's VCC and the Z3 automated prover (2008, deductive verification)
  More recently: verification of PikeOS
- Certified C compiler, developed using the Coq proof assistant (2009, correct-by-construction code generated by a proof assistant)
  http://compcert.inria.fr/
- L4.verified micro-kernel, using tools on top of Isabelle/HOL proof assistant (2010, Haskell prototype, C code, proof assistant)
Other Success Stories at Industry

- Frama-C
  - EDF: abstract interpretation
  - Airbus: deductive verification
- Spark/Ada: Verification of Ada programs

[https://www.adacore.com/industries](https://www.adacore.com/industries)

Remark

The two above use Why3 internally

Outline

Introduction, Short History

Preliminary on Automated Deduction
  - Classical Propositional Logic
  - First-order logic
  - Logic Theories
  - Limitations of Automatic Provers

Introduction to Deductive Verification
  - Formal contracts
  - Hoare Logic
  - Dijkstra’s Weakest Preconditions

Exercises

“Modern” Approach, Blocking Semantics
  - A ML-like Programming Language
  - Blocking Operational Semantics
  - Weakest Preconditions Revisited

Proposition logic in a nutshell

Syntax:

\[ \varphi ::= \bot \mid \top \mid A, B \quad \text{(atoms)} \]
\[ \varphi \land \varphi \mid \varphi \lor \varphi \mid \neg \varphi \]
\[ \varphi \rightarrow \varphi \mid \varphi \leftrightarrow \varphi \]

Semantics, models: truth tables

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

SAT is **decidable** \( \iff \) SAT solvers

Demo with Why3

**propositional.mlw**

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

First-order logic in a nutshell

Syntax:

\[ \varphi ::= \ldots \]
\[ P(t, \ldots, t) \quad \text{(predicates)} \]
\[ \forall x. \varphi \mid \exists x. \varphi \]
\[ t ::= x \mid f(t, \ldots, t) \quad \text{variables} \]

Semantics: models must interpret variables. C

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:

if they answer ‘unsat’ then formula is unsatisfiable

Demo with Why3

**first-order.mlw**

Notice that Why3 logic is **typed**, and application is curried
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 \( P, \overline{P} \) are both theorems
  equivalently: 'false' is not a theorem
  equivalently: the theory has models
- **Consistent Axiomatization**
  'false' is not derivable

Example: theory of equality

\[
\forall x. x = x \\
\forall x, y. x = y \rightarrow y = x \\
\forall x, y, z. x = y \land y = z \rightarrow x = z
\]
(congruence) for all function symbols \( f \) of arity \( k \):
\[
\forall x_1, y_1, \ldots, x_k, y_k. x_1 = y_1 \land \cdots \land x_k = y_k \rightarrow f(x_1, \ldots, x_k) = f(y_1, \ldots, y_k)
\]
and for all predicates \( p \) of arity \( k \):
\[
\forall x_1, y_1, \ldots, x_k, y_k. x_1 = y_1 \land \cdots \land x_k = y_k \rightarrow p(x_1, \ldots, x_k) \rightarrow p(y_1, \ldots, y_k)
\]

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

Logic Theories

- **Theory of a given model**
  = formulas true in this model

  - Central example: theory of linear integer arithmetic, i.e. formulas using \(+, \times, \leq\)
    - 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
Digression about Non-linear Integer Arithmetic

**Representation Theorem (Gödel)**
Every recursive function \( f \) is representable by a predicate \( \phi_f \) such that
\[
\phi_f(x_1, \ldots, x_k, y)
\]
is true if and only if
\[
y = f(x_1, \ldots, x_k)
\]

**First incompleteness Theorem (Gödel)**
That theory is not recursively axiomatizable

---

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, Z3)
  - several theories are built-in
  - weaker with quantifiers
- None support reasoning by induction

---

Outline

- Introduction, Short History
- Preliminary on Automated Deduction
  - Classical Propositional Logic
  - First-order logic
  - Logic Theories
  - Limitations of Automatic Provers
- Introduction to Deductive Verification
  - Formal contracts
  - Hoare Logic
  - Dijkstra’s Weakest Preconditions
- Exercises
  - “Modern” Approach, Blocking Semantics
  - A ML-like Programming Language
  - Blocking Operational Semantics
  - Weakest Preconditions Revisited

---

IMP language

A very basic imperative programming language
- only global variables
- only integer values for expressions
- basic statements:
  - assignment \( x \leftarrow e \)
  - sequence \( S_1; S_2 \)
  - conditionals \( \text{if } e \text{ then } S_1 \text{ else } S_2 \)
  - loops \( \text{while } e \text{ do } S \)
  - no-op \( \text{skip} \)
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:

- $\{x = 1\} x \leftarrow x + 2 \{x = 3\}$
- $\{x = y\} x \leftarrow x + y \{x = 2 \ast y\}$
- $\{\exists v. x = 4 \ast v\} x \leftarrow x + 42 \{\exists w. x = 2 \ast w\}$
- $\{true\} \text{while 1 do skip}\{false\}$

Running Example

Three global variables $n$, $\text{count}$, and $\text{sum}$

```
count <- 0; sum <- 1;
while sum <= n do
  count <- count + 1; sum <- sum + 2 \ast count + 1
```

**What does this program compute?**

(assuming input is $n$ and output is $\text{count}$)

Informal specification:

- at the end of execution of this program, $\text{count}$ contains the square root of $n$, rounded downward
- e.g. for $n=42$, the final value of $\text{count}$ is 6.

See file imp_isqrt.mlw
Hoare logic as an Axiomatic Semantics

Original Hoare logic [\sim 1970]

**Axiomatic Semantics** of programs

Set of *inference rules* producing triples

\[
\begin{align*}
\{P\}\text{skip}\{P\} \\
\{P[x \leftarrow e]\} x \leftarrow e\{P\} \\
\{P\}s_1\{Q\} \quad \{Q\}s_2\{R\} \\
\{P\}s_1; s_2\{R\}
\end{align*}
\]

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

Hoare Logic, continued

**Frame rule:**

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

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

**Consequence rule:**

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

- Example: proof of

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

Hoare Logic, continued

Rules for if and while:

\[
\frac{\{P \land e\}s_1\{Q\} \quad \{P \land \neg e\}s_2\{Q\}}{\{P\}\text{if }e \text{ then } s_1 \text{ else } s_2\{Q\}}
\]

\[
\frac{\{I \land e\}s\{I\}}{\{I\}\text{while }e \text{ do } s\{I \land \neg e\}}
\]

- \(I\) is called a *loop invariant.*

Example: isqrt(42)

Exercise: prove of the triple

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

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 $\Sigma$ associating to each variable $x$ its current value denoted $\Sigma(x)$.
- Value of an expression $e$ in some state $\Sigma$:
  - denoted $[e]_{\Sigma}$
  - always defined, by the following recursive equations:
    - $[n]_{\Sigma} = n$
    - $[x]_{\Sigma} = \Sigma(x)$
    - $[e_1 \text{ op } e_2]_{\Sigma} = [e_1]_{\Sigma} \text{ op } [e_2]_{\Sigma}$
- $[\text{op}]$ natural semantic of operator $\text{op}$ on integers (with relational operators returning 0 for false and $-1$ for true).

Semantics of statements

Semantics of statements: defined by judgment

$$\Sigma, s \leadsto \Sigma', s'$$

meaning: in state $\Sigma$, executing one step of statement $s$ leads to the state $\Sigma'$ and the remaining statement to execute is $s'$.

The semantics is defined by the following rules.

$$\Sigma, x \leftarrow e \leadsto \Sigma\{x \leftarrow [e]_{\Sigma}\}, \text{skip}$$

$$\Sigma, s_1 \leadsto \Sigma', s'_1$$

$$\Sigma, (s_1; s_2) \leadsto \Sigma', (s'_1; s_2)$$

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

Semantics of statements, continued

$$[[e]]_{\Sigma} \neq 0$$

$$\Sigma, \text{if } e \text{ then } s_1 \text{ else } s_2 \leadsto \Sigma, s_1$$

$$[[e]]_{\Sigma} = 0$$

$$\Sigma, \text{if } e \text{ then } s_1 \text{ else } s_2 \leadsto \Sigma, s_2$$

$$[[e]]_{\Sigma} \neq 0$$

$$\Sigma, \text{while } e \text{ do } s \leadsto \Sigma, (s; \text{while } e \text{ do } s)$$

$$[[e]]_{\Sigma} = 0$$

$$\Sigma, \text{while } e \text{ do } s \leadsto \Sigma, \text{skip}$$
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 $\Sigma$, reaches state $\Sigma'$ with remaining statement $s'$ after executing some finite number of steps.

Running example:

$$\{n = 42, count = ?, sum = ?\}, ISRT \rightsquigarrow^* \{n = 42, count = 6, sum = 49\}, skip$$

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 $\Sigma, s \rightsquigarrow^* \Sigma', \text{skip}$

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

Semantics of formulas

- $[p]_\Sigma$:
  - semantics of formula $p$ in program state $\Sigma$
  - is a logic formula where no program variables appear anymore
  - defined recursively as follows.

$$[\mathbf{true}]_\Sigma = \top$$
$$[\mathbf{false}]_\Sigma = \bot$$
$$[p_1 \land p_2]_\Sigma = [p_1]_\Sigma \land [p_2]_\Sigma$$

where semantics of expressions is augmented with

$$[\exists \mathbf{v}. e]_\Sigma, \nu = \top$$
$$[\forall \mathbf{x}. e]_\Sigma = \top$$

Notations:

- $\Sigma \vdash p$ : the formula $[p]_\Sigma$ is valid
- $\vdash p$ : formula $[p]_\Sigma$ holds in all states $\Sigma$.  

Semantics of formulas

Other presentation of the semantics: $[p]_\Sigma$

- inline semantic of first-order formula
- $[e]_{\Sigma, \nu}$ with $\nu$ mapping of logic variables to integers.
- defined recursively as follows.

$$[\mathbf{true}]_{\Sigma, \nu} = \top$$
$$[\mathbf{false}]_{\Sigma, \nu} = \bot$$
$$[p_1 \land p_2]_{\Sigma, \nu} = [p_1]_{\Sigma, \nu} \land [p_2]_{\Sigma, \nu}$$

where semantics of expressions is augmented with

$$[\exists \mathbf{v}. e]_{\Sigma, \nu} = \nu(\mathbf{v})$$
$$[\forall \mathbf{x}. e]_{\Sigma, \nu} = \Sigma(\mathbf{x})$$
Soundness

**Definition (Partial correctness)**

Hoare triple \{P\}s\{Q\} is said valid if:
for any states \(\Sigma, \Sigma'\), if
- \(\Sigma, s \leadsto *\Sigma', \text{skip and}\)
- \(\Sigma \models P\)

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.

Annotated Programs

**Goal**

Add automation to the Hoare logic approach

We augment IMP with *explicit loop invariants*

```plaintext
while e invariant l do s
```

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:

\[
\begin{align*}
\text{WLP(skip, } Q) &= Q \\
\text{WLP}(x < e, Q) &= Q[x \leftarrow e] \\
\text{WLP}(s_1; s_2, Q) &= \text{WLP}(s_1, \text{WLP}(s_2, Q)) \\
\text{WLP}(\text{if } e \text{ then } s_1 \text{ else } s_2, Q) &= (e \rightarrow \text{WLP}(s_1, Q)) \land (\neg e \rightarrow \text{WLP}(s_2, Q))
\end{align*}
\]
Definition of $WLP(s, Q)$, continued

\[ WLP(\text{while } \varnothing \text{ invariant } l \text{ do } s, Q) = \]
\[ l \land (\text{invariant true initially}) \]
\[ \forall v_1, \ldots, v_k. \]
\[ (((e \land l) \rightarrow WLP(s, I)) \land ((\neg e \land l) \rightarrow Q))[w_i \leftarrow v] \land (\text{invariant preserved}) \]
\[ (\text{invariant implies post}) \]

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

Examples

\[ WLP(x \leftarrow x + y, x = 2y) \equiv x + y = 2y \]

\[ WLP(\text{while } y > 0 \text{ invariant } even(y) \text{ do } y \leftarrow y - 2, \text{even}(y)) \equiv \]
\[ even(y) \land \forall v, ((v > 0 \land even(v)) \rightarrow even(v - 2)) \land ((v \leq 0 \land even(v)) \rightarrow even(v)) \]

Soundness

Theorem (Soundness)

For all statement \( s \) and formula \( Q \), \( \{WLP(s, Q)\}s\{Q\} \) is valid.

Proof by induction on the structure of statement \( s \).

Consequence

For proving that a triple \( \{P\}s\{Q\} \) is valid, it suffices to prove the formula \( P \rightarrow WLP(s, Q) \).

This is basically the goal that Why3 produces

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.


Yet, this does not provide an effective recipe to discover suitable loop invariants (see also the theory of abstract interpretation [Cousot, 1990])
Outline

Introduction, Short History
Preliminary on Automated Deduction
  Classical Propositional Logic
  First-order logic
  Logic Theories
  Limitations of Automatic Provers
Introduction to Deductive Verification
  Formal contracts
  Hoare Logic
  Dijkstra’s Weakest Preconditions
Exercises

“Modern” Approach, Blocking Semantics
  A ML-like Programming Language
  Blocking Operational Semantics
  Weakest Preconditions Revisited

Exercises

Exercise 1

Consider the following (inefficient) program for computing the sum $a + b$.

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

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

▶ Propose a post-condition stating that the final value of $x$ is the sum of the values of $a$ and $b$
▶ Find an appropriate loop invariant
▶ Prove the program.

Exercise 2

The following program is one of the original examples of Floyd.

```plaintext
q <- 0; r <- x;
while r >= y do
  r <- r - y; q <- q + 1
```

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

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

Exercise 3

Let’s assume given in the underlying logic the functions $\text{div2}(x)$ 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 $x^n$.

```plaintext
r <= 1; p <- x; e <- n;
while e > 0 do
  if mod2(e) != 0 then r <- r * p;
  p <- p * p;
  e <- div2(e);
```

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

▶ Assuming that the power function exists in the logic, specify appropriate pre- and post-conditions for this program.
▶ Find an appropriate loop invariant, and prove the program.
Exercise 4

The Fibonacci sequence is defined recursively by \( \text{fib}(0) = 0 \), \( \text{fib}(1) = 1 \) and \( \text{fib}(n + 2) = \text{fib}(n + 1) + \text{fib}(n) \). The following program is supposed to compute \( \text{fib} \) in linear time, the result being stored in \( y \).

\[
y \leftarrow 0; \quad x \leftarrow 1; \quad i \leftarrow 0;
\]
\[
\text{while } i < n \text{ do}
\]
\[
\quad \text{aux} \leftarrow y; \quad y \leftarrow x; \quad x \leftarrow x + \text{aux}; \quad i \leftarrow i + 1
\]

▶ Assuming \( \text{fib} \) exists in the logic, specify appropriate pre- and post-conditions.

▶ Prove the program.

----

Outline

Introduction, Short History

Preliminary on Automated Deduction
  Classical Propositional Logic
  First-order logic
  Logic Theories
  Limitations of Automatic Provers

Introduction to Deductive Verification
  Formal contracts
  Hoare Logic
  Dijkstra’s Weakest Preconditions

Exercises

“Modern” Approach, Blocking Semantics
  A ML-like Programming Language
  Blocking Operational Semantics
  Weakest Preconditions Revisited

Beyond IMP and classical Hoare Logic

Extended language
  ▶ more data types
  ▶ \textit{logic variables}: local and immutable
  ▶ \textit{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 \textit{mutable} variables, functions
  ▶ complex data types

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. \textit{Prove that the triple}

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

\textit{is valid with respect to the operational semantics.}

2. \textit{Show that the triple above can be proved using the rules of Hoare logic.}

Let us assume that we replace the standard Hoare rule for assignment by the rule

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

3. \textit{Show that the triple} \( \{ P[x \leftarrow e] \} x \leftarrow e \{ P \} \) \textit{can be proved with the new set of rules.}
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 +, −, × (no division)
- reals: type real, operators +, −, × (no division)
- Comparisons of integers or reals, returning a boolean
- "if-expression": written if b then t₁ else t₂

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

Local logic variables

We extend the syntax of terms by

\[
t ::= \text{let } v = t \ \text{in } t
\]

Example: approximated cosine

```plaintext
let cos_x = 
  let y = x*x in 
  1.0 - 0.5 * y + 0.04166666 * y * y 
  in
...
```

Practical Notes

- Theorem provers (Alt-Ergo, CVC4, Z3) typically support these types
- may also support if-expressions and let bindings

Alternatively, Why3 manages to transform terms and formulas when needed (e.g. transformation of if-expressions and/or let-expressions into equivalent formulas)
Syntax: Formulas

Unchanged w.r.t to previous syntax, but also addition of local binding:

\[ p ::= t \mid p \land p \mid p \lor p \mid \neg p \mid p \Rightarrow p \]  (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 \( \Gamma \) maps identifiers to types:

- either \( v : \tau \) (logic variable, immutable)
- either \( x : \text{ref } \tau \) (program variable, mutable)

Important

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

Typing rules

Constants:
\[ \Gamma \vdash n : \text{int} \quad \Gamma \vdash r : \text{real} \]
\[ \Gamma \vdash \text{True} : \text{bool} \quad \Gamma \vdash \text{False} : \text{bool} \]

Variables:
\[ v : \tau \in \Gamma \quad x : \text{ref } \tau \in \Gamma \]
\[ \Gamma \vdash v : \tau \quad \Gamma \vdash x : \tau \]

Let binding:
\[ \Gamma \vdash t_1 : \tau_1 \quad \{ v : \tau_1 \} \cdot \Gamma \vdash t_2 : \tau_2 \]
\[ \Gamma \vdash \text{let } v = t_1 \text{ in } t_2 : \tau_2 \]

- All terms have a base type (not a reference)
- In practice: Why3, unlike OCaml, does not require to write \( \text{let } \) \( \text{x} \) 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)

\[\begin{align*}
\llbracket \text{val} \rrbracket_{\Sigma, n} &= \text{val} \\
\llbracket x \rrbracket_{\Sigma, n} &= \Sigma(x) \\
\llbracket v \rrbracket_{\Sigma, n} &= \Pi(v) \\
\llbracket \text{op } t_1 \text{ op } t_2 \rrbracket_{\Sigma, n} &= \llbracket t_1 \rrbracket_{\Sigma, n} \text{ op } \llbracket t_2 \rrbracket_{\Sigma, n} \\
\llbracket \text{let } v = t_1 \text{ in } t_2 \rrbracket_{\Sigma, n} &= \llbracket t_2 \rrbracket_{\Sigma, \{ v = \llbracket t_1 \rrbracket_{\Sigma, n} \}}
\end{align*}\]

Warning

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

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

- sequence `e1; e2` : syntactic sugar for

  \[
  \text{let } v = e_1 \text{ in } e_2
  \]

  when `e1` has type `unit` and `v` not used in `e2`

Toy Examples

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

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

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

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

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

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

Conditional:
\[
\frac{\Gamma \vdash c : \text{bool} \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \text{if } c \text{ then } e_1 \text{ else } e_2 : \tau}
\]

Loop:
\[
\frac{\Gamma \vdash c : \text{bool} \quad \Gamma \vdash e : \text{unit}}{\Gamma \vdash \text{while } c \text{ do } e : \text{unit}}
\]

Operational Semantics

Novelty w.r.t. IMP
Need to precise the order of evaluation: left to right

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

- values do not reduce

Operational Semantics, continued

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

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

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

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

- **Loop**
  \[\Sigma, \Pi, \text{while } c \text{ do } e \Rightarrow \Sigma, \Pi, \text{if } c \text{ then } (e; \text{while } c \text{ do } e) \text{ else } ()\]

Context Rules versus Let Binding

Remark: most of the context rules can be avoided

- An equivalent operational semantics can be defined using
  \[\text{let } v = ... \text{ in } ...\] instead, e.g.:
  \[v_1, v_2 \text{ fresh}\]
  \[
  \Sigma, \Pi, e_1 + e_2 \Rightarrow \Sigma, \Pi, \text{let } v_1 = e_1 \text{ in let } v_2 = e_2 \text{ in } v_1 + v_2
  \]

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

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

\[
\begin{align*}
\llbracket P \rrbracket_{\Sigma, \Pi} & \text{ holds} \\
\Sigma, \Pi, \text{assert } P & \hookrightarrow \Sigma, \Pi, ()
\end{align*}
\]

\[
\begin{align*}
\llbracket \_ \rrbracket_{\Sigma, \Pi} & \text{ holds} \\
\Sigma, \Pi, \text{while } c \text{ invariant } l \text{ do } e & \hookrightarrow \\
\Sigma, \Pi, \text{if } c \text{ then } e \text{; while } c \text{ invariant } l \text{ do } e \text{ else } ()
\end{align*}
\]

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 $WP(e, Q)$

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:
  $WP(t, Q) = Q[result ← t]$

- Let binding:
  $WP(\text{let } x = e_1 \text{ in } e_2, Q) = WP(e_1, WP(e_2, Q)[x ← result])$

Weakest Preconditions, continued

- Assignment:
  $WP(x ← e, Q) = WP(e, Q[result ← ()]; x ← result)$

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

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

Soundness of WP

Lemma (Preservation by Reduction)

If \( \Sigma, \Pi \models \text{WP}(e, Q) \) and \( \Sigma, \Pi, e \rightsquigarrow \Sigma', \Pi', e' \) 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 \( \Sigma', \Pi', e' \) 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 \( \Sigma, \Pi \).
- if execution terminates, \( Q \) holds in the final state

Bibliography


Bibliography

