Proof Engineering for Program Logics in Isabelle/HOL

Lecture 1: Introduction to Program Semantics

Chelsea Edmonds
University of Western Australia
chelsea.edmonds@uwa.edu.au

ANU Logic Summer School 2025

Course Overview

Lectures:

1. **Basic reasoning on programs in Isabelle/HOL**
2. Program Logics: Hoare and Rely-Guarantee
3. A side quest: intro to coinduction in Isabelle/HOL
4. Formally defining Rely-guarantee reasoning
5. Modular proofs in Isabelle/HOL

Mix of theory and Isabelle/HOL implementations/proofs.

Pre-Requisite Knowledge

- Intro to Isabelle by Liam in Week 1!
- Some familiarity with basic imperative programming language concepts will be beneficial.
- No assumed knowledge of formal program semantics/program logics etc.

Course Resources

- Download Isabelle if you haven't already: `https://isabelle.in.tum.de/`
- Course materials: `https://cledmonds.github.io/anulss2025/`
- Isabelle documentation (prog-prove, coinductive datatypes, locales tutorials)
- Concrete Semantics: `http://concrete-semantics.org/`

Lecture 1 Overview

- Intro to Program Verification in Proof Assistants
- Program Syntax
- Small-step program semantics for a sequential language
- Basic properties and proofs on programs
- Introducing a parallel operator.

Acknowledgement: This lecture and Isabelle material draws on ideas from Nipkow and Klein's concrete semantics textbook.

Some Background

What is "Program Verification"?

The most basic version of program verification is considering if a program results in the correct outcome with respect to some specification. But there is a lot more we might want to consider beyond just "correctness"

There's a lot to think about even in a simple verification problem:

- What are the language features/properties?
- What environmental assumptions are we making?
- Are there any constraints/expectations on the variables?
- What properties do we want to verify?
- How do we state the specification?
- ...

Why use proof assistants?

Just a few reasons include ...

- Strong correctness guarantees (no human error!)
- Modern day ITP tasks benefit from existing large libraries.
- Automation during development (including counter-example generators).
- One part of larger tool-chains.
- Immediate feedback during development of new ideas!

Proof assistants have been widely used in different software verification projects, from Intel using HOL-Light to verify floating point arithmetic to the Sel4 fully verified operating system using Isabelle (done here in Australia!)

Software verification using proof assistants can be found in both academia and industry.

Program Semantics

Introduction

How do we describe a programming language?

- Syntax of programs (i.e. how we represent programs).
- Semantics (i.e. meaning of a program).

From a formal verification perspective (or just good PL design!), we require precise, mathematical descriptions of both these elements.

A simple IMP Language

We'll work with a simple imperative language, which contains:

- A store consisting only of integer values.
- Variable assignment
- Basic boolean tests and integer arithmetic
- Sequential composition
- If-else statements
- Iteration (while statements)

This is enough to represent our "sample" programs and to test out theory, but would obviously need to be significantly expanded to represent "real" programs.

Syntax

We define some basic grammars for arithmetic expressions, boolean expressions, and our program commands:

$$
\begin{array}{lll}
\text{aexp} & ::= & \text{N } \textit{int} \mid \text{V } \textit{vname} \mid \text{Plus aexp aexp} \\
\text{bexp} & ::= & \text{Bc } \textit{bool} \mid \text{Not bexp} \mid \text{And bexp bexp} \mid \text{Less aexp aexp} \\
\text{Com} & ::= & \text{done} \mid \text{Assign } \textit{vname}\ \text{aexp} \mid \text{seq Com Com} \mid \\
 & & \text{if bexp Com Com} \mid \text{while bexp Com}
\end{array}
$$

State and Configurations

An assignment operator implies the presence of a store, i.e. some way of tracking the program's state.

We'll first consider a program state mapping variable names to integers.

$$\text{State} :: \mathit{vname} \Rightarrow \mathit{int}$$

For convenience, a state given by $< x := 5 >$ maps x to 5 and all other variables to 0.

A *configuration* refers to a command-state pair (c, s).

In more complex program verification tasks, correctly modelling the memory system can take considerable effort.

Semantics Approach

There are several different common ways to approach formally defining a programs semantics, such as:

- Operational Semantics: defining a relation that describes how a program executes
- Denotational Semantics: concerned with giving mathematical models of programming languages.

Operational semantics are commonly defined as either:

- Big-step semantics: the execution of a program is described as one “big step“ from initial to final state.
- Small-step semantics: considers individual atomic execution steps (i.e. partial program execution).

Semantics Approach

Before defining our operational semantics based on the available program commands, we must also consider:

- How to evaluate arithmetic expressions
- How to evaluate boolean expressions

i.e. give our expression syntax meaning!

The functions aval and bval evaluate the expressions as we would expect.

Isabelle Demo

Semantics Approach

We'll take the small-step approach:

- Enables reasoning on intermediate program steps.
- Considers how far execution has progressed.
- Added granularity is particularly important for reasoning on concurrent programs.
- Program executions are a sequence of execution steps.

Small-step operational semantics considers how the program can change after a single program step. It offers a level of granularity that is particularly useful when reasoning on concurrent programs.

Full Semantics

$$\frac{}{(x := v, s) \Rightarrow (\text{done}, s(x := \text{aval } v\ s))}(\text{Assign})$$

$$(\text{seq done } c, s) \Rightarrow (c, s) \quad (\text{Seq-Done})$$

$$\frac{(c_1, s) \Rightarrow (c_1', s')}{(\text{seq } c_1\ c_2, s) \Rightarrow (\text{seq } c_1'\ c_2, s')}(\text{Seq})$$

$$\frac{\text{bval } t\ s = \text{True}}{(\text{if } t\ c_1\ c_2, s) \Rightarrow (c_1, s)}(\text{If-True}) \qquad \frac{\text{bval } t\ s = \text{False}}{(\text{if } t\ c_1\ c_2, s) \Rightarrow (c_2, s)}(\text{If-False})$$

$$(\text{while } t\ c, s) \Rightarrow (\text{if } t\ (\text{seq } c\ (\text{while } t\ c))\ \text{done}, s) \quad (\text{While})$$

Figure 1: Small-step operational semantics

A sample execution

Take the simple program: $(x := 1); (y := 2)$.

Using our small step semantics

$$
\begin{aligned}
(\,\texttt{seq}\,(x := 1)\,(y := 2),\ s\,) &\Rightarrow (\,\texttt{seq done}\,(y := 2),\ s[x \mapsto 1]\,) \\
&\Rightarrow (\,y := 2,\ s[x \mapsto 1]\,) \\
&\Rightarrow (\,\texttt{done},\ s[x \mapsto 1,\ y \mapsto 2]\,)
\end{aligned}
$$

The *first* step of this execution is justified by the following proof tree:

$$
\dfrac{\dfrac{}{(x := 1,\ s) \Rightarrow (\text{done},\ s[x \mapsto 1])}\ \text{Assign}}{(\text{seq}\ (x := 1)\ (y := 2),\ s) \Rightarrow (\text{seq done}\ (y := 2),\ s[x \mapsto 1])}\ \text{Seq}
$$

Isabelle Demo

Some basic proofs

Basic Program Definitions/Properties

Termination:

- Will a program terminate?
- What does a finished program look like?

We introduce a final definition:

$$\text{final}\ (c, s) \iff \neg\, \exists\, cf.(c, s) \to cf$$

Reachability:

- Even with a small-step semantics, we want to be able to reason over multiple computation steps.
- Reachability is defined as the transitive closure of the small-step relation.

We use the standard star notation for our small step semantics.

$$(c, s) \rightarrow^* (c', s')$$

means there is a series of computation steps from (c, s) to (c', s') in our small-step semantics.

Basic Program Definitions/Properties

Deterministic Execution:

Is the execution of a program under our semantics deterministic? i.e:

$$\forall c\ s\ c'\ s'\ c''\ s''.(c, s) \to (c', s')\ \wedge\ (c, s) \to (c'', s'') \Longrightarrow (c', s') = (c'', s'')$$

Semantic Equivalence:

Two commands (in our simple IMP language) are considered semantically *equivalent* if:

- they both terminate with the same state.
- they both diverge.

$$c \sim c' \equiv ((c, s) \rightarrow^* (\text{done}, t) \longleftrightarrow (c', s) \rightarrow^* (\text{done}, t))$$

Isabelle Demo

Introducing Concurrency

Modelling Concurrent Programs

Concurrency is an integral part of modern computing (hardware, multi-threading, networks etc).

It can, however, introduce a lot of additional considerations to PL semantics, e.g:

- Managing different threads
- Atomic vs non-atomic execution steps
- Shared memory
- communication between different threads.

Modelling Concurrent Programs

In our simple IMP language there are a few choices from a modelling (or "proof engineering") choices, e.g:

- Extending our existing semantics with a simple parallel operator
- Modelling threads (e.g. lists) of computation steps.
- Having a local (thread-based) and global semantics.

For simplicity, we'll go with the first one.

A Parallel Operator

We first extend the syntax to include a parallel operator:

$$\text{Com} \quad ::= \quad \ldots \mid \text{par Com Com}$$

We then add the following rules to our small-step operational semantics:

$$\frac{(c_1, s) \Rightarrow (c_1', s')}{(\text{par } c_1\ c_2, s) \Rightarrow (\text{par } c_1'\ c_2, s')}(\text{Par-L}) \qquad \frac{(c_2, s) \Rightarrow (c_2', s')}{(\text{par } c_1\ c_2, s) \Rightarrow (\text{par } c_1\ c_2', s')}(\text{Par-R})$$

$$(\text{par done done}, s) \Rightarrow (\text{done}, s) \quad (\text{Par-Done})$$

Figure 2: Small-step operational semantics

Parallel Operator Consequences

By embedding the parallel operator into our small-step semantics alongside all other operators, we make assumptions such as:

- Parallel operations can be interleaved at will and nested.
- No locking - all other single computation steps are considered atomic, and any thread can progress at any time.
- All memory is effectively "shared memory".

Parallel Program Properties

How might concurrency impact the few basic program properties we've already considered?

Parallel Program Properties

Consider the program below:

$$x := 1 \,||\, x := 2$$

What is the value of x after this program terminates?

Parallel Program Properties

Consider the program below:

$$x := 1 \,||\, x := 2$$

It could be 1 or 2! *We no longer have a deterministic semantics.*

Isabelle Demo

Next Time

Next Lecture:

- Hoare Logic
- Rely Guarantee Logic
- Soundness and Proof systems

Isabelle exercises/extended work

- Read up on big-step semantics in the Concrete Semantics Textbook, chapter 7
- Try out some of the Isabelle exercises from chapter 7.