Proof Engineering for Program Logics in Isabelle/HOL

Lecture 2: Program Logics

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

ANU Logic Summer School 2025

Course Overview

Lectures:

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

Mix of theory and Isabelle/HOL implementations/proofs.

Lecture 2 Overview

- What is a Program Logic?
- Hoare Logic: Intuition, Semantics, and Rules
- Hoare Logic: Soundness
- Hoare Logic in Isabelle
- Rely-Guarantee Logic

Acknowledgment: Hoare Logic Isabelle content inspired by Nipkow & Klein's Concrete Semantics textbook.

Program Logics: An Overview

What is a Program Logic?

A *program logic* is a formal language (based on mathematical logic) for expressing and proving various properties of programs.

Well-known examples

There are numerous existing program logics (and various extensions) that can target a variety of different program properties and environments. Some examples include:

- 1967 - 1969: Hoare Logic (or Floyd-Hoare Logic). [2]
- 1983: Rely-Guarantee Logic [3]
- 2000: Separation Logic [5]
- 2020: Incorrectness Logic [4]

Hoare Logic

Hoare Logic Intuition

For a sequential program, we can consider a *Hoare Triple*

$$\{P\}\ C\ \{Q\}$$

Where:

- P is a predicate that represents the *pre-condition*
- Q is a predicate that represents the *post-condition*
- C is the program.

The Hoare triple states that if the pre-condition P is satisfied on the initial program state, then *if* the program terminates, the post-condition Q will be satisfied.

Some Example Programs

Lets consider some basic example programs:

Listing 1: basic assignment

```
{ x = n }
x := x + 1
{ x = n + 1 }
```

Listing 2: basic swap

```
{ x = a ∧ y = b }
t := x;
x := y;
y := t
{ x = b ∧ y = a }
```

Partial vs Total Correctness

Our Hoare triple definition is dependent on *termination*. Hence it represents *partial correctness*.

Total correctness ($[P]\ C\ [Q]$) has the following two conditions:

- If a program C starts in a state satisfying P, then the program terminates
- When the program terminates, the state satisfies Q.

In other words:

$$\text{Total Correctness} = \text{Termination} + \text{Partial Correctness}$$

More examples...

Listing 3: Terminating Loop

```
{ n >= 0 }
while n > 0 do
  n := n - 1
{ n = 0 }
```

Partial Correctness: ✓

Total Correctness: ✓

Listing 4: Divergent Loop

```
{ x >= 0 }
while x >= 0 do
  x := x + 1
{ x < 0 }
```

Partial Correctness: ✓

Total Correctness: $\times$

We'll focus on partial correctness for now.

A More Formal Semantics

More formally, for partial correctness, our Hoare triple is valid ($\models$) according to the following definition:

$$\models \{P\}C\{Q\} \longleftrightarrow (\forall s\ t.\ P\ s \wedge (c, s) \rightarrow^* (c', t) \wedge \text{final}\ (c', t) \longrightarrow Q\ t)$$

Syntactic vs Semantic Assertions

In our examples so far we've been using *syntactic* assertions, i.e.

$$\{x = n\}$$

is an assertion on a syntactic expression, such as bexp, which is nice and intuitive for simple examples.

However, in our formal semantics, it's it becomes much easier to reason using *semantic* assertions, i.e. predicates over a program state s. For example, the syntactic assertion above corresponds to the semantic assertion:

$$\{\lambda s.s(x) = n\}$$

A Hoare Logic Proof System

We can use inference rules to specify a formal proof system for Hoare logic.

Under this proof system, we use:

$$\vdash \{P\}\, C\, \{Q\}$$

means the Hoare triple $\{P\}\ C\ \{Q\}$ can be derived.

Some Useful Relation Notation

Some notes on notation, assuming P and Q are unary predicates.

- We use $<$ and $>$ to be the standard ordering on predicates. e.g. $P < Q$ means that $\forall\ s.\ P\ s \longrightarrow Q\ s$
- $\sqcap$ and $\sqcup$ denote the infimum and supremum in the lattice of predicates, which are component-wise conjunction and disjunction. e.g. $(P \sqcap Q)\ s = P\ s \wedge Q\ s$

Later we'll also be using binary predicates, e.g. R.

- refl R means R is a reflexive relation, i.e. $\forall s\ s'.R\ s\ s' \longleftrightarrow R\ s'\ s$
- stable $P\ R$ means P is stable with respect to R, i.e. $\forall s\ s'.P\ s \wedge R\ s\ s' \longrightarrow P\ s'$

A Hoare Logic Proof System

$$\frac{}{\vdash \{P\}\,\mathsf{done}\,\{P\}}\text{(DoneH)} \qquad \frac{}{\vdash \{(\lambda s.\ P\ s[a/x])\}\ x ::= a\ \{P\}}\text{(AssignH)}$$

$$\frac{\vdash \{P\}\ c_1\ \{P'\} \qquad \vdash \{P'\}\ c_2\ \{Q'\}}{\vdash \{P\}\ \mathsf{seq}\ c_1\ c_2\ \{Q\}}\text{(SeqH)}$$

$$\frac{\vdash \{P \sqcap (\mathsf{bval}\ t)\}\ c_1\ \{Q\} \qquad \vdash \{P \sqcap (\neg\ (\mathsf{bval}\ t))\}\ c_2\ \{Q\}}{\vdash \{P\}\ \mathsf{if}\ t\ c_1\ c_2\ \{Q\}}\text{(IfH)}$$

$$\frac{\vdash \{P \sqcap (\mathsf{bval}\ t)\}\ c\ \{P\}}{\vdash \{P\}\ \mathsf{while}\ t\ c\ \{P \sqcap \neg\mathsf{bval}\ t\}}\text{(WhileH)}$$

Hoare Logic: Assignment Rule

Let's take a closer look at the Assignment Rule:

$$\overline{\vdash \{(\lambda s.\ P\ s[a/x])\}\, x ::= a\, \{P\}}\,(\text{AssignH})$$

This can feel a little backwards, modifying the pre-condition instead of the post-condition. Why wouldn't the other way around work?

$$\overline{\vdash \{P\}\, x ::= a\, \{\lambda s.\ P\ s[a/x]\}}\,(\text{AssignHBad})$$

Hoare Logic: Assignment Rule

Let's take a close look at the Assignment Rule:

$$\frac{}{\vdash \{(\lambda s.\ P\ s[a/x])\}\, x ::= a\, \{P\}}(\text{AssignH})$$

This can feel a little backwards, modifying the pre-condition instead of the post-condition. Why wouldn't the other way around work?

$$\frac{}{\vdash \{P\}\, x ::= a\, \{\lambda s.\ P\ s[a/x]\}}(\text{AssignHBad})$$

We could use it to prove a triple like this is valid!

$$\{x = 0\}\, x ::= 1 \{1 = 0\}$$

The While Rule

$$\frac{\vdash \{P \sqcap (\text{bval } t)\}\; c\; \{P\}}{\vdash \{P\}\; \text{while } t\; c\; \{P \sqcap \neg\text{bval } t\}}\,(\text{WhileH})$$

In this rule P acts as a loop invariant.

- It is true at the beginning and end of every loop iteration
- If the loop terminates, the condition t must be false.

The Consequence Rule

$$\frac{\vdash \{P'\}\, c\, \{Q'\} \qquad P \leq P' \qquad Q' \leq Q}{\vdash \{P\}\, c\, \{Q\}}\,(\text{MonoH})$$

Figure 1: Consequence Rule

This allows us to

- Strengthen the pre-condition
- Weaken the post-condition.

Deriving Rules

We can derive rules, that might be easier to work with. For example, here is an alternate rule for Assign:

$$\frac{\forall\, s. P\ s \longrightarrow Q\ s[a/x]}{\vdash \{P\}\, x ::= a\, \{Q\}}\,(\text{AssignH'})$$

Figure 2: Derived Assign Rule

This is derived via the consequence rule (strengthen the precondition), and original AssignH rule.

And for while:

$$\frac{\vdash \{P \sqcap (\text{bval } t)\}\, c\, \{P\} \qquad P \sqcap (\neg\, (\text{bval } t)) \leq Q}{\vdash \{P\}\, \text{while } t\ c\, \{Q\}}\,(\text{WhileH'})$$

Figure 3: Derived While Rule

Isabelle Demo

Soundness and Completeness

There are two important properties we typically want to consider when developing a program logic proof system w.r.t. an operational semantics.

First is *Soundness*: if a triple is derivable, then it is also valid.

$$\vdash \{P\}\, c\, \{Q\} \longrightarrow \models \{P\}\, c\, \{Q\}$$

Next is *Completeness*: if a triple is valid, then it is also derivable.

$$\models \{P\}\, c\, \{Q\} \longrightarrow \vdash \{P\}\, c\, \{Q\}$$

We'll focus on soundness proofs in this course. Completeness requires the introduction of the weakest pre-condition, which is left as further reading.

Isabelle Demo

Rely-Guarantee Logic

Additional Concurrency Considerations

For reasoning on the correctness of concurrent programs, we also need to consider:

- What do we *require* of the environment for a given sequential command to hold.
- How could our sequential command *impact* the environment.

The Rely-Guarantee Approach

Rely-Guarantee Logic extends Hoare Logic to reason about concurrent programs:

$$\{P, R\}\ C\ \{G, Q\}$$

Where C, P and Q are as before, and:

- R a binary predicate representing the *rely-condition*.
- G is a binary predicate representing the *guarantee-condition*.

Our triple now also requires that the environment only makes changes to the state that satisfy the rely-condition R, and that the program only makes changes to the state that satisfy the guarantee-condition G.

We focus on *partial correctness* again.

The Rely-Guarantee Approach

Slightly more formally, consider a command whose execution trace has environment steps $\epsilon(\sigma_i, \sigma_{i+1})$ and program steps $\tau(\sigma_i, \sigma_{i+1})$, where σ_i represents the state after i steps:

$$\sigma_0 \ldots \tau(\sigma_i, \sigma_{i+1}) \ldots \epsilon(\sigma_j, \sigma_{j+1}) \ldots \sigma_f$$

$\{P, R\}\ C\ \{G, Q\}$ holds means:

- $P\ \sigma_0$ holds
- $Q\ \sigma_f$ holds if the command terminates
- Every environment step ϵ satisfies the rely condition, i.e. $R\ \sigma_j\ \sigma_{j+1}$
- Every program step τ satisfies the guarantee condition, i.e. $G\ \sigma_i\ \sigma_{i+1}$

An (Intuitive) Assignment Example

Consider the below Hoare triple:

Listing 5: basic assignment RG

```
{ x = 0 }
x := x + 1
{ x = 1 }
```

Say this command is running in a parallel environment. For it to hold under our RG logic, we additionally:

- *rely* on the condition that x is not changed by the environment.
- *guarantee* that our program at most increments x by 1.

A rely-guarantee proof system

$$\frac{\text{stable } Q\ R \qquad P \leq Q}{\vdash \{\, P, R \,\}\, \text{done} \, \{\, G, Q \,\}}\text{(DoneRG)}$$

$$\frac{\begin{array}{cc} \text{stable } P\ R & \text{stable } Q\ R \\ (\lambda s'.\ \exists s.\ P\ s \wedge s' = s[a/x]) \leq Q & (\lambda s, s'.\ P\ s \wedge s' = s[a/x]) \leq G \end{array}}{\vdash \{\, P, R \,\}\, x := a \, \{\, G, Q \,\}}\text{(AssignRG)}$$

$$\frac{\vdash \{\, P, R \,\}\, c_1 \, \{\, G, P' \,\} \qquad \vdash \{\, P', R \,\}\, c_2 \, \{\, G, Q \,\} \qquad \text{refl } G}{\vdash \{\, P, R \,\}\, \text{seq } c_1\ c_2 \, \{\, G, Q \,\}}\text{(SeqRG)}$$

A rely guarantee proof system

$$\frac{\vdash \{\, P \sqcap (\mathsf{bval}\ t), R \,\}\, c_1 \,\{\, G, Q \,\} \qquad \vdash \{\, P \sqcap (\neg\, (\mathsf{bval}\ t)), R \,\}\, c_2 \,\{\, G, Q \,\} \qquad \mathsf{stable}\ P\ R \qquad \mathsf{refl}\ G}{\vdash \{\, P, R \,\}\, \mathsf{if}\ t\ c_1\ c_2 \,\{\, G, Q \,\}}\,(\mathrm{IfRG})$$

$$\frac{\vdash \{\, P \sqcap (\mathsf{bval}\ t), R \,\}\, c \,\{\, G, P \,\} \qquad P \sqcap (\neg\, (\mathsf{bval}\ t)) \leq Q \qquad \mathsf{stable}\ P\ R \qquad \mathsf{stable}\ Q\ R \qquad \mathsf{refl}\ G}{\vdash \{\, P, R \,\}\, \mathsf{while}\ t\ c \,\{\, G, Q \,\}}\,(\mathrm{WhileRG})$$

$$\frac{\begin{array}{c}\vdash \{\, P_1, R_1 \,\}\, c_1 \,\{\, G_1, Q_1 \,\} \qquad \vdash \{\, P_2, R_2 \,\}\, c_2 \,\{\, G_2, Q_2 \,\} \\ P \leq P_1 \sqcap P_2 \qquad R \sqcup G_2 \leq R_1 \qquad R \sqcup G_1 \leq R_2 \\ G_1 \sqcup G_2 \leq G \qquad Q_1 \sqcap Q_2 \leq Q \qquad \mathsf{refl}\ G\end{array}}{\vdash \{\, P, R \,\}\, \mathsf{par}\ c_1\ c_2 \,\{\, G, Q \,\}}\,(\mathrm{ParRG})$$

The Parallel Rule

$$\frac{\begin{array}{c}\vdash \{\, P_1, R_1 \,\}\, c_1 \,\{\, G_1, Q_1 \,\} \qquad \vdash \{\, P_2, R_2 \,\}\, c_2 \,\{\, G_2, Q_2 \,\} \\ P \leq P_1 \sqcap P_2 \qquad R \sqcup G_2 \leq R_1 \qquad R \sqcup G_1 \leq R_2 \\ G_1 \sqcup G_2 \leq G \qquad Q_1 \sqcap Q_2 \leq Q \qquad \mathsf{refl}\ G\end{array}}{\vdash \{\, P, R \,\}\, \mathsf{par}\ c_1\ c_2 \,\{\, G, Q \,\}}\,(\mathsf{ParRG})$$

The parallel rule is our critical new rule in the rely-guarantee reasoning proof system. It states for a parallel step to be derived:

- Both $c1$ and $c2$ satisfy their respective RG tuples.
- The precondition is equivalent to (or stronger) then the conjunction of $P1$ and $P2$
- The postcondition is equivalent to (or weaker) than the conjunction of $Q1$ and $Q2$
- The guarantee condition is equivalent to (or weaker) than the disjunction of $G1$ and $G2$
- $G2$ is compatible with $R1$
- $G1$ is compatible with $R2$

An example: Rely-Guarantee Conditions

Consider $c1$ where:

- $G1\ s\ s' \equiv s\ y = s'\ y$
- $R1\ s\ s' \equiv s'\ x < s\ x$

and $c2$ where:

- $G2\ s\ s' \equiv s'\ x < s\ x$
- $R2\ s\ s' \equiv s'\ y \geq s\ y$

Clearly we have that:

$$G2\ s\ s' \longrightarrow R1\ s\ s' \wedge G1\ s\ s' \longrightarrow R2\ s\ s'$$

So $c1$ and $c2$ could run in parallel with no interference issues.

The Rely-Guarantee Consequence Rule

$$\frac{\vdash \{\, P', R' \,\}\, c \,\{\, G', Q' \,\} \qquad P \le P' \qquad R \le R' \qquad G' \le G \qquad Q' \le Q}{\vdash \{\, P, R \,\}\, c \,\{\, G, Q \,\}}\,(\text{MonoRG})$$

Figure 4: Rely-Guarantee Consequence Rule

Isabelle Demo

Rely-Guarantee Semantics?

Ok, so we have a proof system - but what about our formal Rely-Guarantee semantics? i.e. how do we define:

$$\models \{P, R\}\, c\, \{G, Q\}$$

Concurrency introduces some challenges for a formal definition:

- How do we model environment vs program steps?
- How do we capture the rely/guarantee conditions?

We hinted out this with our slightly more formal definition earlier.

Rely-Guarantee Semantics Approaches

Multiple approaches exist to address these challenges, including:

- Trace-Based
- Reachability
- Inductive
- Coinductive?

We'll discuss these in more detail in Lecture 4.

Note: other approaches to rely-guarantee reasoning also exist, including a more algebraic refinement calculus style of reasoning by Hayes et al [1], and extensions such as total correctness and relational post-conditions.

Next Time

Next Lecture: A side quest into coinduction!

- What is coinduction?
- How does it relate to inductive principles?
- How do we work with coinduction in Isabelle?
- We'll return to Rely-Guarantee in Isabelle in Lecture 4.

Isabelle exercises/extended work

Hoare Logic is covered in Chapter 12 of the Concrete Semantics Textbook.

- Try out some of the exercises from section 12.1/12.2
- Read up on weakest pre-conditions and potential for automation (section 12.4)

Ian J. Hayes and Cliff B. Jones.
A guide to rely/guarantee thinking.
In Jonathan P. Bowen, Zhiming Liu, and Zili Zhang, editors, *Engineering Trustworthy Software Systems*, pages 1–38, Cham, 2018. Springer International Publishing.

C. A. R. Hoare.
An axiomatic basis for computer programming.
Commun. ACM, 12(10):576–580, October 1969.

Cliff B. Jones.
Specification and design of (parallel) programs.
In *IFIP Congress*, 1983.

Peter W. O'Hearn.
Incorrectness logic.
Proc. ACM Program. Lang., 4(POPL), December 2019.

John C. Reynolds.
Separation logic: A logic for shared mutable data structures.
In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, LICS '02, page 55–74, USA, 2002. IEEE Computer Society.