Proof Engineering for Program Logics in Isabelle/HOL

Lecture 5: Modular Proofs in Isabelle/HOL

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 and more**.

Mix of theory and Isabelle/HOL implementations/proofs.

Lecture Overview

- Motivating modularity
- An introduction to Locales in Isabelle
- A slight aside - type classes!
- Locales for program verification abstraction
- Wrapping things up

Motivating modularity

Introducing Abstractness

Consider these definitions that we've encountered so far:

- Hoare Logic validity:

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

- Coinductive safety for RG Logic

$$\frac{\begin{array}{ll} 1. & \forall s'.\ R\ s\ s' \implies \mathsf{safeC}\ (c, s')\ (R, G, Q) \\ 2. & \mathsf{final}\,(c, s) \implies Q\ s \\ 3. & \forall c', s'.\ ((c, s) \Rightarrow (c', s')) \implies G\ s\ s' \wedge \mathsf{safeC}\ (c', s')\ (R, G, Q) \end{array}}{\mathsf{safeC}_{(R,G,Q)}\,(c, s)}\ (\mathsf{StepC})$$

What do we need to know about our programming language semantics to reason on these definitions?

Introducing Abstractness

Consider these definitions that we've encountered so far:

- Hoare Logic validity:

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

- Coinductive safety for RG Logic

$$\dfrac{\begin{array}{ll} 1. & \forall s'.\ R\ s\ s' \implies \mathsf{safeC}\ (c, s')\ (R, G, Q) \\ 2. & \mathsf{final}\,(c, s) \implies Q\ s \\ 3. & \forall c', s'.\ ((c, s) \Rightarrow (c', s')) \implies G\ s\ s' \wedge \mathsf{safeC}\ (c', s')\ (R, G, Q) \end{array}}{\mathsf{safeC}_{(R,G,Q)}\,(c, s)}\ \text{(StepC)}$$

Just two things: (1) a small step relation, and (2) a final definition!

Modularity

In our examples so far, we have:

- Worked with a specific operational semantics.
- Defined and proved properties that don't depend on specific details of the operational semantics. . .

i.e. we've still got some proof engineering to go!

So how do we introduce this abstractness in Isabelle?

Introduction to Locales

Locale Basics

Locales are Isabelle's module system. From a logical perspective, they are simply persistent contexts:

$$\bigwedge x_1 \ldots x_n.\ [A_1;\ A_2;\ \ldots;\ A_m] \implies C$$

- Provides fixed type and term variables
- Provides contextual assumptions (related to the above) within a local context.

```
locale semigroup_orig =
  fixes mult :: "'a ⇒ 'a ⇒ 'a" (infixl "⊗" 70) (* Parameter*)
  assumes assoc: "(x ⊗ y) ⊗ z = x ⊗ (y ⊗ z)" (* Assumption *)
```

We'll use basic group theory to demonstrate some locale features!

Locale Basics

Let's introduce an explicit *carrier set* to our semigroup locale:

```
locale semigroup_orig =
  fixes M and composition (infixl "·" 70) (* Parameters*)
  assumes composition_closed [intro, simp]:
    "⟦a ∈ M;  b ∈ M⟧ ⟹ a · b ∈ M" (* Assumption *)
  assumes assoc[intro]: "⟦a ∈ M;  b ∈ M;  c ∈ M⟧ ⟹
    (a · b) · c = a · (b · c)" (* Assumption *)
```

We've also:

- Added the intro and simp annotations to our locale assumptions
- The type is no longer necessary as we're working with a carrier set.

Locale Interpretation

We can *interpret* an instance of a locale for use anywhere in the theory.

Interpretation label   Locale being interpreted

```
interpretation ints: semigroup ℤ plus
  by unfold_locales simp_all
```

Locale proof tactic   Terms to "instantiate" locale parameters

We can now *use* inherited locale properties outside the locale context:

```
lemma "(1 + 2) + (3 ::int) = 1 + (2 + 3)"
  using ints.assoc by simp
```

Extending Locales

You can *extend* a locale with new assumptions and parameters:

```
locale monoidl = semigroup + fixes unit :: 'a ("1")
  assumes unit_closed [intro, simp]: "1 ∈ M"
  and unitl[intro, simp]: "x ∈ M ⟹ 1 · x = x"

locale monoidr = semigroup + fixes unit :: 'a ("1")
  assumes unit_closed [intro, simp]: "1 ∈ M"
  and unitr[intro, simp]: "x ∈ M ⟹ x · 1 = x"
```

$$\begin{tikzcd} & \textit{semigroup} \arrow[dl] \arrow[dr] & \\ \textit{monoidl} & & \textit{monoidr} \end{tikzcd}$$

Extending Locales

You can also combine existing locales to make a new one. Locales manage diamond inheritance patterns easily.

```
locale monoid = monoidl + monoidr
```

$$
\begin{tikzcd}
 & \textit{semigroup} \arrow[dl] \arrow[dr] & \\
\textit{monoidl} \arrow[dr] & & \textit{monoidr} \arrow[dl] \\
 & \textit{monoid} &
\end{tikzcd}
$$

Extending Locales

When extending a locale it is possible to pass in the parameter names/syntax you want to use for the new locale.

The for keyword can be useful for listing even more details (including type names, specifying parameter order etc).

```
locale submonoid = monoid M "(·)" 𝟏
  for N and M and composition (infixl "·" 70) and unit ("𝟏") +
  assumes subset: "N ⊆ M"
    and sub_composition_closed: "⟦a ∈ N; b ∈ N⟧ ⟹ a · b ∈ N"
    and sub_unit_closed: "𝟏 ∈ N"
```

Locale Contexts

To do proofs inside a locale context we can open the context immediately after the definition using begin and end once finished:

```
locale monoid = monoidl + monoidr
begin
lemma comp_one_is_one: "1 · 1 = 1 "
  by simp
end
```

Locale Contexts

We can also open the context at any time after the definition:

```
context monoid
begin
lemma comp_one_is_one: "𝟏 · 𝟏 = 𝟏 "
  by simp
end
```

…or indicate a single lemma is in a locale's context via an annotation:

```
lemma (in monoid) comp_one_is_one: "𝟏 · 𝟏 = 𝟏 "
  by simp
```

Indirect Inheritance

In addition to direct inheritance (extending the locale), we can also establish inheritance *indirectly* using the sublocale command.

```
sublocale submonoid ⊆ monoid N "(·)" 1
  by unfold_locales
  (auto simp: sub_composition_closed sub_unit_closed subset)
```

This example shows that a submonoid is also a monoid itself, giving it access to all the definitions and theorems on monoids!

Local Interpretations

- Use interpret to get an "instance" of a locale to use *within* your proof context.
- Locale proof tactics in the same proof context consider local interpretations.
- Particularly useful when working outside a locale context

```
theorem submonoid_transitive:
  assumes "submonoid K N composition unit"
    and "submonoid N M composition unit"
  shows "submonoid K M composition unit"
proof -
  interpret K: submonoid K N composition unit by fact
  interpret M: submonoid N M composition unit by fact
  show ?thesis by unfold_locales auto
qed
```

Proofs with Locales

Proof Tactics

- There are two main tactics for locale proofs: unfold_locales, and intro_locales
- unfold_locales unfolds all the locale assumptions (including from locales earlier in the hierarchy) and discharges any goals where the assumption is already in the proof context.
- intro_locales unfolds only one layer of the locale hierarchy.
- Using these before trying sledgehammer will make your life easier!!!

Isabelle Demo

An aside: Type Classes

Type classes Overview

Type classes introduce polymorphism and overloading into the Isabelle/HOL infrastructure, building on top of the Locale infrastructure.

Isabelle type classes are "Haskell-like". They enable you to:

- Specify abstract parameters together with corresponding specifications
- Instantiate those abstract parameters by a particular type
- In connection with a less ad-hoc approach to overloading.
- Inherit from existing locale declarations.

Use the class command, otherwise mirrors our existing locale syntax.

```
class semigroup =
  fixes mult :: "'a ⇒ 'a ⇒ 'a" (infixl "⊗" 70)
  assumes assoc: "(x ⊗ y) ⊗ z = x ⊗ (y ⊗ z)"
```

Type classes: Advantages vs Disadvantages

Advantages

- We can show a type is an instance of a type class, and get these properties "for free" when just using that type.
- We can use type classes directly in definition declarations etc.

Disadvantages

- Type class operations are restricted to a single type parameter, and can only be instantiated in one way per type: E.g. a list may be ordered multiple ways, but can only instantiate an order type class once.
- Parameters are fixed over the whole type class hierarchy and cannot be refined in specific situations
- Type class inheritance has limitations: e.g. We can't declare monoidr separately, then try to bring them together easily.

Back to Locales

Locales obviously aren't types - you can't define a function that takes a locale as a parameter.

...but they have proved to be a very powerful and effective alternate when Isabelle's type class limits are reached.

Taking a *locale-centric* approach, particularly for managing large hierarchies of structures/proof contexts is becoming increasingly common.

Using Locales in Verification

Abstract Small-Step

As explored at the start of the lecture, many of our definitions only require on an abstract notion of a small step relation, e.g:

- Hoare triple validity
- Rely-guarantee various semantics definitions

Proofs on these definitions also only require some kind of small step relation.

It is only soundness proofs (using a proof system) which are done w.r.t. a specific operational semantics

Abstract Small-Step

We introduce a basic locale with parameters that encapsulate a small-step relation and final relation.

```
locale Step =
  fixes small_step :: "'com × 'state ⇒ 'com × 'state ⇒ bool"
      (infix ":→" 60)
  and final :: "'com × 'state ⇒ bool"
```

Our definitions/lemmas can now go inside the locale.

Abstract Deterministic Small-Step

What if we wanted to prove many lemmas that relied on a certain program property?

Rather than passing this in as an assumption in every lemma, we could extend our locale:

```
locale Step_Deterministic = Step +
assumes determ: "((cs :→ cs') ∧ (cs :→ cs'')) ⟹ (cs'' = cs')"
```

Small-step Interpretation

Our concrete small-step semantics is a trivial interpretation of the basic step locale:

```
interpretation Step small_step final .
```

Using our deterministic lemma it also can be shown to be an interpretation of the step_deterministic locale.

```
interpretation det: Step_Deterministic small_step final
  apply (unfold_locales)
  using deterministic by blast
```

An intermediatory Small-step semantics?

Rather than jumping straight from our very abstract Step locale to a concrete implementation, we often want to *refine gradually*.

For example, our arithmetic and boolean expressions have no impact on any of our results, so could be formalised much more abstractly.

```
locale IMPLang =
  fixes aval :: "'aexp ⇒ state ⇒ val"
  and bval :: "'bexp ⇒ state ⇒ bool"
  and Not :: "'bexp ⇒ 'bexp"
  assumes bval_Not[simp]: "⋀ s t. bval (Not t) s = (¬ bval t s)"
```

An intermediatory Small-step semantics?

As our "more concrete" operational semantics are now expressed as a locale, we can use sublocales (instead of an interpretation) to set up a *persistent* inheritance relation across the locales:

```
sublocale IMPLang < Step where
  small_step = small_step' and final = final' .
```

Note in this sublocale declaration, the small_step and final on the left-hand side represent the locale parameters of Step, and the ones on the right-hand side are the definitions in the more concrete IMPLang locale we want to instantiate them with.

Isabelle Demo

More Locales in Action

Locales can come in very handy when doing any kind of refinement, e.g. specifying an abstract object/definition and proving properties on it, before later *interpreting* a more concrete implementation.

Other example use cases of locales include:

- Specifying an abstract data structure, and reasoning on different interpretations.
- Modelling information flow leaks ("Relative Security")
- Managing large mathematical hierarchies
- Algorithm verification

Conclusions

Wrapping things up

What have we covered:

- Small step operational semantics (theory and Isabelle)
- Program Logics: Hoare Logic and Rely Guarantee Logic
  - Hoare logic proof systems and hoare triple validity
  - Rely-guarantee logic proof systems
  - Different ways of reasoning on rely-guarantee clause validity
  - Soundness of program logics
  - All of the above in Isabelle!
- An introduction to coinduction (as a dual of induction), and using it in Isabelle
- Locales and modularity in Isabelle

Main takeaways

- Proving something is correct is only one part of using a proof assistant.
- You can use a proof assistant as part of the research process when developing new ideas, not just after the fact.
- Program logics are numerous and powerful tools for reasoning about programs.
- Proof engineering is important! It can help us find hidden patterns, develop modular libraries which are reusable, make libraries much more maintainable, and save a lot of time in the long run.
- Coinduction can provide a natural and elegant way of reasoning about programs.

These can be carried over to working in any interactive proof assistant.

What's next?

If we've managed to get you interested in interactive theorem proving. . .

- Try some more Isabelle yourself!
  - Prog-prove tutorial
  - Concrete Semantics textbook
  - Explore the Isabelle "Archive of Formal Proofs" for topics you're interested in: `https://www.isa-afp.org/`
- Try out a different proof assistant: Lean, Rocq, HOL4/HOL Light etc. Most have good tutorials (or even online games!) to start with.

Warning... proof assistants can be addictive!

What's next?

There is a lot of active research in program verification. If you're new to the field, publication venues you might want to look at to keep an eye on current research include:

- Conferences on interactive theorem proving: ITP, CPP (or JAR for journal).
- Programming language conferences (publications backed by formal proofs are common): e.g. POPL, PLDI, OOPSLA/SPLASH, ICFP
- Formal methods conferences: e.g. FM, ESOP
- More domain specific venues: e.g. CSF (Security)

What's next?

- Course resources will remain on the website - and expect solutions for all lectures to be posted within the next week.
- If the RG Coinductive work was of interest - expect paper and full AFP entries to become available early next year
- If you have any questions, feedback, or ideas, feel free to get in touch!