LECTURE 2: LOCALES, TYPE CLASSES & MODULARITY

MODULAR PROOFS IN ISABELLE HOL

CHELSEA EDMONDS | c.l.edmonds@sheffield.ac.uk

Midlands Graduate School 2025 |

University of Sheffield

1

COURSE OVERVIEW

A practical course on effective use of the Isabelle/HOL proof assistant in mathematics and programming languages

Lectures:

- Introduction to Proof Assistants
- Formalising the basics in Isabelle/HOL
- **Introduction to Isar, more types, Locales and Type classes**
- Case studies:
  - Formalising Mathematics: Combinatorics & advanced locale reasoning patterns
  - Program Verification: Formalising semantics, program properties, and introducing modularity/abstraction.

Example Classes:

- Isabelle exercises based on the previous lecture
- Will be drawing from the existing Isabelle tutorials/Nipkow's Concrete Semantic Book, as well as custom exercises (e.g. for locales).

2

LECTURE 2 OVERVIEW

Modular proofs = an engineering-like approach to formalisation.

Yesterday: Introduction to proof assistants, and a tour of Isabelle/HOL.

TODAY:

- Finishing off Isabelle introduction
  - A little more on types in Isabelle
- The role of modularity in formalisation
- Intro to Locales and Type-classes

3

ISAR: A STRUCTURED PROOF LANGUAGE

4

STRUCTURED PROOFS

- The Isar proof language allows us to do *structured human-readable proofs*
- It is also very easy to use! Pick almost any AFP entry, and you'll see elements of Isar style proofs
- Useful for breaking down a theorem into smaller goals, which may not be useful as their own lemmas.
- Useful keywords for calculations: (`have, also have, finally`) and (`have, moreover have, ultimately`)
- Proofs can also be nested

```
lemma ex3_isar:
  assumes "(P ∧ Q) ⟶ R"
  shows " P ⟶ (Q ⟶ R)"
proof (rule impI)+
  assume P Q
  then have "P ∧ Q" by (intro conjI)
  then show R using assms by (elim mp)
qed
```

```
lemma dvd_trans:
  fixes a :: nat
  assumes ab: "a dvd b" and bc: "b dvd c"
  shows "a dvd c"
proof -
  obtain v where "b = a * v"
    using dvdE ab by blast
  moreover obtain w where "c = b * w"
    using dvdE bc by blast
  ultimately have "c = a * v * w"
    by blast
  then show ?thesis by simp
qed
```


SOME MORE ON TYPES

BASIC TYPES

- Yesterday we introduced datatypes as an example of a user defined type in Isabelle
- *Today:*
  - More datatypes
  - Type declarations
  - Type Synonyms
  - Pairs
  - Record types
  - And finally … type classes.

7

DATATYPES

- One common use case of datatypes is an option datatype

```
datatype 'a option = None | Some 'a
```

- Datatypes can be parameterised by multiple types:

```
datatype ('a, 'b, 'c) three = Three 'a 'b 'c
```

- Datatypes can also be annotated:

```
datatype (set: 'a) mylist =
    null: mNil  (‹[]›)
  | mCons (mhd: 'a) (mtl: "'a mylist")  (infixr ‹##› 65)
```

Discriminator

Infix Notation

Constructors

Selectors

- The datatypes (and co-datatypes) tutorial has significantly more information.

8

TYPE SYNONYMS, DECLARATIONS, AND DEFINITIONS

- A type synonym can be useful to make a formalisation more readable/descriptive. E.g.

```
type_synonym 'a edge = "'a set"
```

  - declares a parameterised edge type which is the same as a set
- A type declaration declares a new type without defining it

```
typedecl Test
```

- A type definition allows you to define a new type

```
typedef three = "{0:: nat, 1, 2}"
  apply (intro exI[of _ 0]) (* Goal must show RHS is non-empty *)
  by simp
```

  - You must prove the type is not empty
  - Introduces Rep and Abs properties to convert between reasoning on base type and new type (then you need to establish useful properties)…
  - Or in this case just use a datatype which does the setup for you!

9

PAIRS

- While functions are usually curried, it is also possible to work with a pair type in Isabelle.
- For example, below is a type synonym which represents a graph that uses a pair

```
type_synonym 'a graph = "'a set × 'a edge set"
```

- Built in definitions to access the elements:

```
lemma "(λ(x,y).x) p = fst p"
  by(simp add: split_def)

lemma "(λ(x,y).y) p = snd p"
  by (simp split: prod.split)
```

10

RECORD TYPES

- Records are essentially an n-tuple, with labels, a familiar programming language construct
- Each field has a type (which may be polymorphic), field names are part of the record type, and the order of the fields is important.

```
record point =
  Xcoord :: int
  Ycoord :: int
```

```
definition pt1 :: point where
"pt1 ≡ (| Xcoord = 999, Ycoord = 23 |)"
```

- Record types support basic extensions.

```
datatype colour = Red | Green | Blue

record cpoint = point +
  col :: colour
```

11

DEMONSTRATION

RECORDS AND TYPES

12

TYPE CLASSES

13

INTRODUCTION

- Type classes introduce polymorphism and overloading into the Isabelle/HOL 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
  - Link to the Isabelle module system (we'll get to this later!)

Custom Notation

Parameters

Specification

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

- For more info see the type class tutorial and hierarchy documentation for examples: https://isabelle.in.tum.de/library/Doc/Typeclass_Hierarchy/typeclass_hierarchy.pdf

*Taken from the Isabelle Type Class Tutorial

14

TYPE CLASS INSTANCE

- To instantiate a type class by a particular type an instance proof is required:

Local def of param

Instance Proof

```
instantiation int :: semigroup
begin

definition mult_int_def : "i ⊗ j = i + (j::int)"

instance proof
  fix i j k :: int have "(i + j) + k = i + (j + k)" by simp
  then show " (i ⊗ j) ⊗ k = i ⊗ (j ⊗ k)" unfolding mult_int_def .
qed
end

lemma "(1 + 2) + (3 ::int) = 1 + (2 + 3)"
  using assoc by simp (* directly use *)
```

Can now use type class assumptions outside class context

15

SUBCLASS

Direct Inheritance

- Build directly off an existing class by adding new parameters and/or assumptions

```
class monoidl = semigroup +
  fixes neutral :: 'a ("1")
  assumes neutl: "1 ⊗ x = x"
```

```
class monoid = monoidl +
  assumes neutr: "x ⊗ 1 = x"
```

Indirect Inheritance

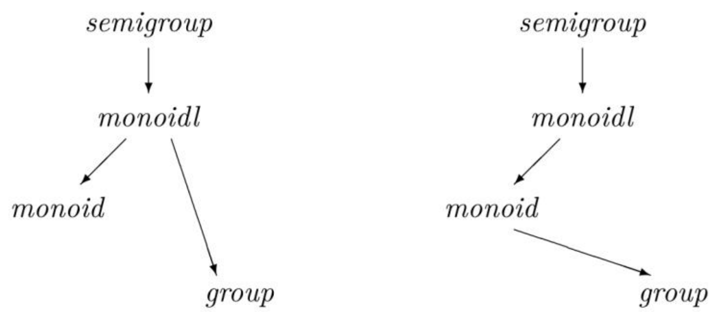
- We can use `subclass` to introduce indirect inheritance (with a proof)

```
class group = monoidl +
  fixes inverse :: "'a ⇒ 'a"
  assumes invl: "(inverse x) ⊗ x = 1"
```

16

SUBCLASS INHERITANCE HIERARCHY

- The impact of using subclass to manipulate the inheritance hierarchy.



17

DEMO

18

LIMITATIONS?

- 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.

```
class monoidr = semigroup +
 fixes neutral :: 'a ("1")
 assumes neutr: "x ⊗ 1 = x"

class monoid = monoidl + monoidr
```

Proof state Auto hovering Auto update Update Search:

```
Duplicate parameter(s) in superclasses: "neutral"
```

19

SO WHAT'S THE ALTERNATIVE?

20

LOCALES

21

LOCALE BASICS

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

$$\bigwedge x_1 \ldots x_n .\, [\![A_1; \ldots ; A_m]\!] \Rightarrow C.$$

- Provides fixed type and term variables and contextual assumptions within a local context.
- Type classes use and can interact with the underlying locale infrastructure.

Class

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

Same params/assumptions as before

Locale inheritance

```
class semigroup_orig_add = plus +
  assumes add_assoc: "(a + b) + c = a + (b + c)"
begin

sublocale add: semigroup_orig plus
  by standard (fact add_assoc)

end
```

22

LOCALE BASICS

- Locales allow us to work explicitly with "carrier sets" (if we want to)

Carrier set

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

- Think of locales as more of a set-based rather than type-based approach.

23

INTERPRETING A LOCALE

- Global theory interpretation:

Label interpretation

Locale being interpreted

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

Terms to "instantiate" locale parameters with

locale tactic

- Can also now use inherited locale properties outside locale context

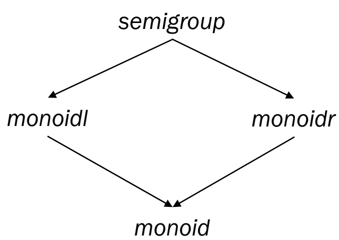
```
lemma "(1 + 2) + (3 ::int) = 1 + (2 + 3)"
  using ints.assoc by simp
```

Must reference named interpretation

24

DIAMONDS & MANIPULATING THE INHERITANCE HIERARCHY

- Locales support “inheritance diamonds” basically automatically




```
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"

locale monoid = monoidl + monoidr
```

25

MORE LOCALE KEYWORDS AND CONTEXTS

- When “inheriting” a locale it is possible to pass in the parameter names/syntax you want to use
- The **for** keyword can be useful for listing even more details (including type names etc, specifying parameter order etc).
- Proofs inside the locale context use parameters/assumptions naturally

for declaration

```
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"
begin

lemma sub [intro, simp]:
  "a ∈ N ⟹ a ∈ M"
  using subset by blast

end
```

Locale context

26

LOCALE CONTEXTS CONTINUED

- It is possible to "reopen" the locale context at any time (i.e. you can continue to add to a locale after its definition, and even in separate theories etc).

context "reopens" locale

```
context submonoid
begin
lemma sub [intro, simp]:
  "a ∈ N ⟹ a ∈ M"
  using subset by blast
end
```

- A lemma can also be stated "outside" a locale context, but added via the **in** keyword

```
lemma (in submonoid) sub [intro, simp]:
  "a ∈ N ⟹ a ∈ M"
  using subset by blast
```

27

LOCAL LOCALE INTERPRETATION

- Locally interpreting a locale is the most common type of interpretation.
- It gives you an "instance" of a locale to work with in your proof context.
- Locale proof tactics inside the proof also consider local interpretations in the hierarchy
- 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
```

Two local labelled interpretations

28

LOCALE PROOF TACTICS

- There are two main tactics for locale proofs: **unfold_locales**, and **intro_locales**
- The first 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.
- The second unfolds only one layer of the locale hierarchy.
- Using these *before* trying sledgehammer will make your life easier!!!

```
interpretation ints: semigroup ℤ plus
  apply unfold_locales
```

Proof state Auto hovering Auto update Update Search:

```
proof (prove)
goal (2 subgoals):
 1. ⋀a b. a ∈ ℤ ⟹ b ∈ ℤ ⟹ a + b ∈ ℤ
 2. ⋀a b c. a ∈ ℤ ⟹ b ∈ ℤ ⟹ c ∈ ℤ ⟹ a + b + c = a + (b + c)
```

29

DEMO

30

MODULAR PROOFS

31

THE CHALLENGE

- In mathematics/theoretical CS, we often deal with large hierarchies of structures. So:
  - How do formalise these/keep track of relationships?
  - How do we deal with the same structure occurring in different forms with different notation?
  - How can we minimise the need to redo work?

- In program verification there can be added challenges:
  - Sometimes, abstractions are hard (e.g. low-level hardware modelling).
  - More complex structures
  - Less consistency/less pretty!

32

THE SOLUTION

A software engineering-like approach to formalisation

- Type classes and locales (and similar ideas in other proof assistants) are essential as one part of this approach
  - Basically, we need a powerful, but flexible inheritance system.
- Just *using* these isn't enough though – we need to use them smartly.
- How do communities manage this?

33

NEXT TIME…

- Exercises:
  - Types, type classes, and locales.
  - Gain familiarity with defining locales/classes and basic proof techniques.
- Formalisation of Mathematics
  - More advanced locale reasoning patterns in Isabelle
  - Introduction to the field of formalisation of mathematics
  - Combinatorial case studies
- To come… semantics and refinement examples!

34