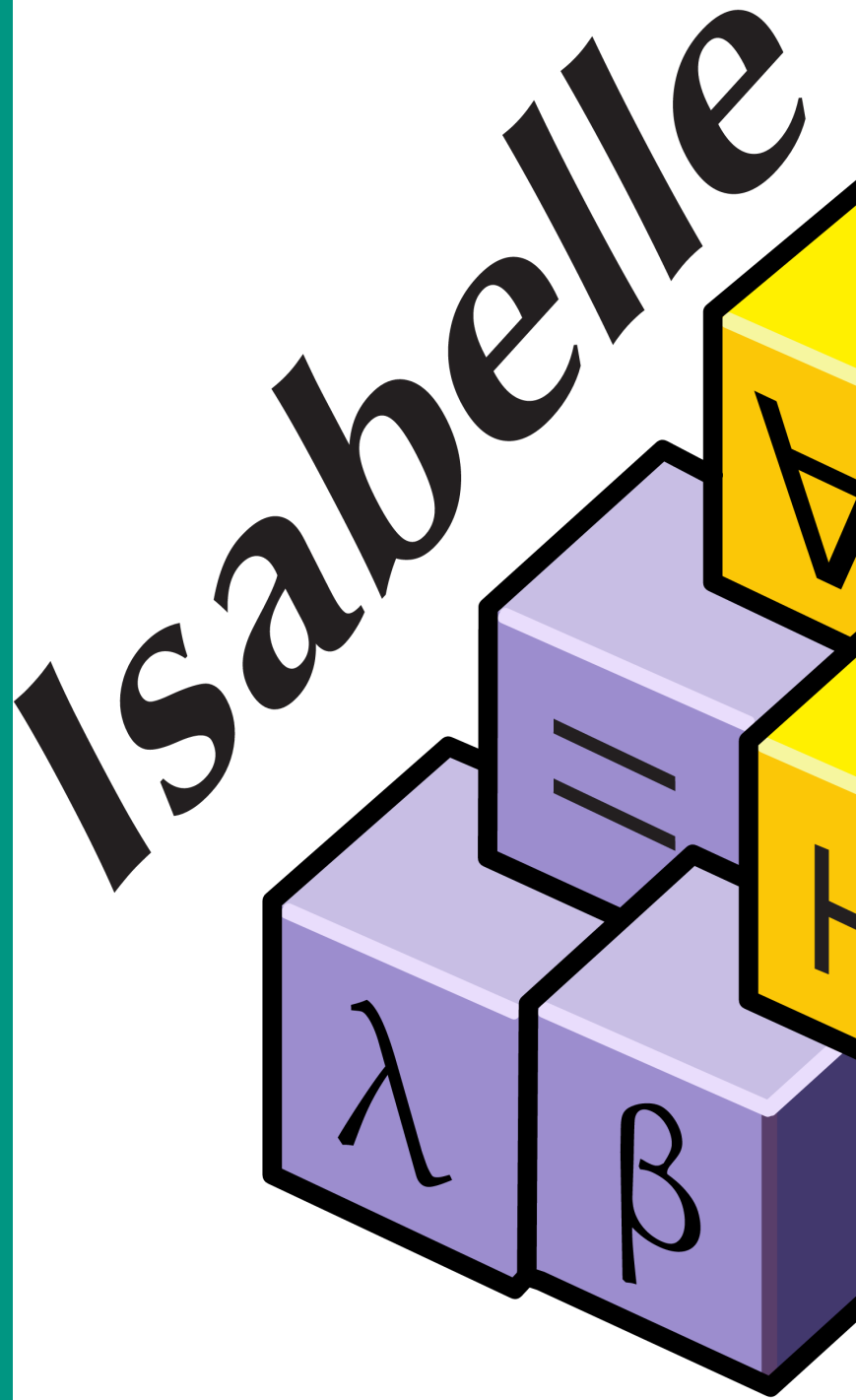


Theorem Prover Lab

Episode 5: Type Classes & Locales

Terru Stübinger | 2025-12-03



Today

Two kinds of ad-hoc polymorphism:

1. Type classes
2. Locales

Project Topics

Polymorphism

So far: `definition` & `fun` with concrete types

```
1 value "2 + 2 = 4"
```

Polymorphism

So far: `definition` & `fun` with concrete types

```
1 value "2 + 2 = 4" (* Cannot derive subsort relation numeral < equal *)
```

Polymorphism

So far: `definition` & `fun` with concrete types

```
1 value "2 + 2 = 4" (* Cannot derive subsort relation numeral < equal *)
```

what is the type of 2? (and where the definition of +?)

Polymorphism

So far: `definition` & `fun` with concrete types

```
1 value "2 + 2 = 4" (* Cannot derive subsort relation numeral < equal *)
```

what is the type of 2? (and where the definition of +?)

Why can we prove

```
1 lemma "2 + 2 = 4"
2   by simp
```

but not:

```
1 lemma "2 * 2 = 4"
2   by simp (* fails *)
```

Polymorphism

So far: `definition` & `fun` with concrete types

```
1 value "2 + 2 = 4" (* Cannot derive subsort relation numeral < equal *)
```

what is the type of 2? (and where the definition of +?)

Why can we prove

```
1 lemma "2 + 2 = 4"
2   by simp
```

but not:

```
1 lemma "2 * 2 = 4"
2   by simp (* fails *)
```

→ *ad-hoc polymorphism*: one symbol, different meanings for different types!

Polymorphism: Type classes

Isabelle has “Haskell-like” type classes:

```
1 class semigroup =  
2   fixes plus :: "'a ⇒ 'a ⇒ 'a" (infixl ⊕ 65)  
3   assumes semigroup_assoc: "(a ⊕ b) ⊕ c = a ⊕ (b ⊕ c)"  
4 begin  
5   :  
6 end
```

- can declare abstract parameters & specification
- concrete types can then instantiate the class
- anything inside the **begin** ... **end** can use the class assumptions

Class Instances

Instantiating a class for a type requires concrete operations & a proof:

```
1  instantiation nat :: semigroup_add begin
2    definition plus_nat_def: "a ⊕ b = (a :: nat) + b"
3    instance proof
4      fix a b c :: nat
5      show "a ⊕ b ⊕ c = a ⊕ (b ⊕ c)"
6        unfolding plus_nat_def by simp
7    qed
8  end
9
10 lemma "1 + (2 + 3) = (1 :: nat + 2) + 3"
11   using semigroup_assoc by simp
```

Subclasses

One class can extend another:

```
1 class monoid1 = semigroup +  
2   fixes neutral :: 'a (1)  
3   assumes neut1: "1 ⊕ a = a"  
4  
5 class monoid = monoid1 +  
6   assumes neutr: "a ⊕ 1 = a"
```

```
1 class group = monoid1 +  
2   fixes inverse :: "'a 'a"  
3   assumes inv1:  
4     "(inverse x) ⊕ x = 1"
```

Subclasses

One class can extend another:

```
1 class monoid1 = semigroup +  
2   fixes neutral :: 'a (1)  
3   assumes neut1: "1  $\oplus$  a = a"  
4  
5 class monoid = monoid1 +  
6   assumes neutr: "a  $\oplus$  1 = a"
```

```
1 class group = monoid1 +  
2   fixes inverse :: "'a 'a"  
3   assumes inv1:  
4     "(inverse x)  $\oplus$  x = 1"
```

But we know that any group is also a monoid!

Extending classes

At any point, we can extend a class:

```
1 lemma (in group) left_cancel:  
2   "x ⊕ y = x ⊕ z ↔ y = z"  
3   using neutl invl semigroup_assoc by metis
```

≈

```
1 lemma left_cancel:  
2   "x ⊕ y = x ⊕ z ↔ y = z"  
3   using neutl invl semigroup_assoc by metis
```

inside the **begin** ... **end** block of the class.

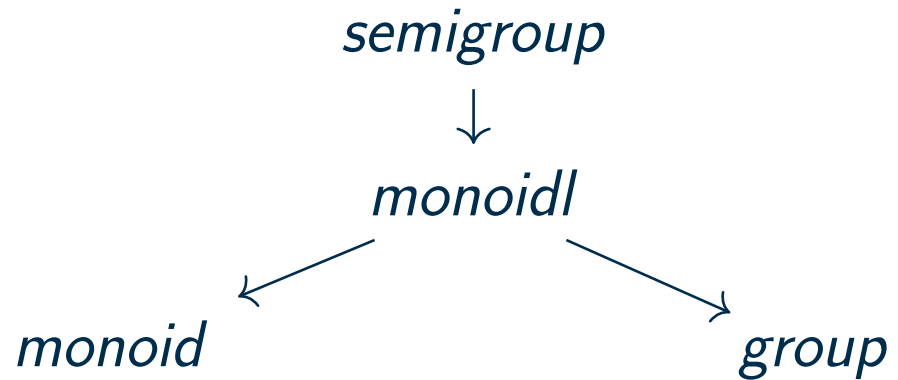
Managing subclass relations

We can also extend its subclass relations!

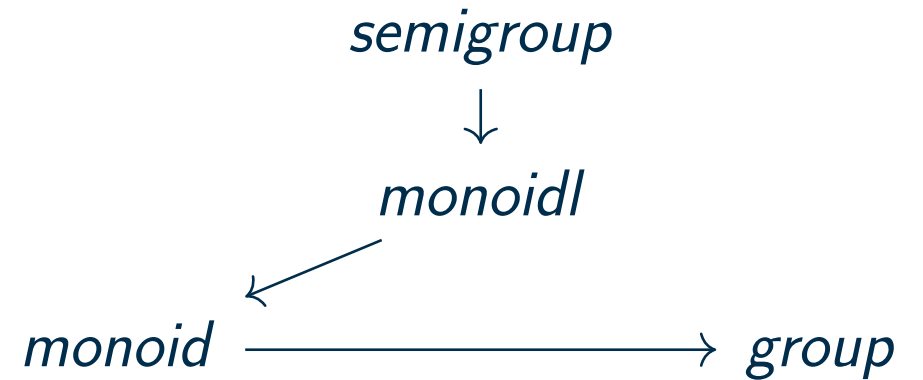
```
1 subclass (in group) monoid proof
2   fix x
3   from invl have inverse x  $\oplus$  x = 1 by simp
4   hence "inverse x  $\oplus$  (x  $\oplus$  1) = inverse x  $\oplus$  x"
5     using semigroup_assoc[symmetric] neutl invl by simp
6   with left_cancel show x  $\oplus$  1 = x by simp
7 qed
```

Class Hierarchy

The class hierarchy is a tree:



→



and `subclass` can manipulate it.

Classes.thy

Introduction
○○○

Type Classes
○○○○○○●○○○○○○

Locales
○○○○○○○○○

Project Topics
○○○



A Special Class: numeral

Some classes are “special” and used by Isabelle:

```
1 datatype num = One | Bit0 num | Bit1 num
2
3 class numeral = one + semigroup_add
4   fixes one    :: 'a    (<1>) and plus  :: "'a 'a  $\Rightarrow$  'a" (infixl <+> 65)
5   assumes add_assoc: "(a + b) + c = a + (b + c)" begin
6
7 fun numeral :: num 'a where
8   numeral_One: numeral One = 1
9   | numeral_Bit0: numeral (Bit0 n) = numeral n + numeral n
10  | numeral_Bit1: numeral (Bit1 n) = numeral n + numeral n + 1
```

Generic statements about numeral

```
1 lemma "2 + 2 = 4"  
2   by simp
```

follows from numeral's assumptions

```
1 lemma "2 * 2 = 4"  
2   (* unprovable *)
```

does not

→ If you use a generic operator, specify which type you mean!

```
1 lemma "2 * 2 = (4 :: int)"  
2   by simp
```

Sorts & sort inference

A sort is to a type as a type is to a value

Sorts & sort inference

A sort is to a type as a type is to a value

sorts = set of constraints which intersect on a type; each class is a sort

Sorts & sort inference

A *sort* is to a type as a type is to a value

sorts = set of constraints which intersect on a type; each class is a sort

```
1  typ "'a :: {plus,ord,numeral}"  
2  term "2 :: ('a :: numeral)"
```

Sorts & sort inference

A *sort* is to a type as a type is to a value

sorts = set of constraints which intersect on a type; each class is a sort

```
1  typ "'a :: {plus,ord,numeral}"  
2  term "2 :: ('a :: numeral)"
```

Isabelle will attempt to infer the most general sorts for term you write

Sorts & sort inference

A *sort* is to a type as a type is to a value

sorts = set of constraints which intersect on a type; each class is a sort

```
1  typ "'a :: {plus,ord,numeral}"  
2  term "2 :: ('a :: numeral)"
```

Isabelle will attempt to infer the most general sorts for term you write

→ might be *too* general

Sorts & sort inference

A *sort* is to a type as a type is to a value

sorts = set of constraints which intersect on a type; each class is a sort

```
1  typ "'a :: {plus,ord,numeral}"
2  term "2 :: ('a :: numeral)"
```

Isabelle will attempt to infer the most general sorts for term you write

→ might be *too* general

→ possible to be specific, e.g.

```
1  lemma "1 * 2 = (2 :: 'a :: {monoid_mult,numeral})"
2  by simp
```

Working with sorts

Sorts (and types) are usually hidden.

This can cause frustration if things don't unify as expected!

```
1  mult.left_neutral [of 2]
2    (* gives 1 * 2 = 2 *)
3  lemma "1 * 2 = 2"
4    using mult.left_neutral [of 2] by simp (* fails! *)
```

You can say:

- declare `[[show_types]]` to always print types
- declare `[[show_sorts]]` to always print sorts (and types)
- Can also put `[[show_types]] / [[show_sorts]]` in a proof to only do this locally

Working with sorts

Sorts (and types) are usually hidden.

This can cause frustration if things don't unify as expected!

```
1 mult.left_neutral [of 2]
2   (* gives (1 :: 'a :: {monoid_mult,numeral}) * 2 = 2 *)
3 lemma "(1 :: 'a :: {times,numeral}) * 2 = 2"
4   using mult.left_neutral [of 2] by simp (* fails! *)
```

You can say:

- declare `[[show_types]]` to always print types
- declare `[[show_sorts]]` to always print sorts (and types)
- Can also put `[[show_types]] / [[show_sorts]]` in a proof to only do this locally

Limitations of Type Classes

We could also define right monoids separately:

```
1 class monoid1 = semigroup +  
2   fixes neutral :: 'a (1)  
3   assumes neutr: "a  $\oplus$  1 = a"
```

But: parameters, once fixed, are fixed for the whole hierarchy!

Limitations of Type Classes

We could also define right monoids separately:

```
1 class monoidr = semigroup +  
2   fixes neutral :: 'a (1)  
3   assumes neutr: "a ⊕ 1 = a"
```

But: parameters, once fixed, are fixed for the whole hierarchy!

→ no way to now write `class monoid = monoidr + monoidl`

Type classes can also only take *one* type parameter (always called 'a)!

... which can only be instantiated in one way per type

Classes.thy

Introduction
○○○

Type Classes
○○○○○○○○○○○○●

Locales
○○○○○○○○○

Project Topics
○○○

Locales

Locales are Isabelle's general-purpose module system.

Mathematically, they are simply persistent contexts:

$$\bigwedge x_1 \dots x_n. \llbracket A_1; \dots; A_m \rrbracket \implies C$$

- Locales can have many operations, type variables, and assumptions
- provide these as fixed variables & theorems inside their context
- can have many (named) interpretations
- Type Classes can interact with underlying locales!

A Locale

We can restate classes as locales:

```
1 locale semigroup =  
2   fixes op :: "'a 'a 'a" (infixl 65)  
3   assumes semigroup_assoc: "(a b) c = a (b c)"  
4 begin ... end
```

A Locale

We can restate classes as locales:

```
1 locale semigroup =
2   fixes op :: "'a 'a 'a" (infixl 65)
3   assumes semigroup_assoc: "(a b) c = a (b c)"
4 begin ... end
```

But locales can also do more:

```
1 locale kvmap =
2   fixes get :: "'k 'a 'v" and set :: "'k 'v 'a 'a"
3   and empty :: 'a
4   assumes "get k (set k v m) = v"
5   and "set k (get k m) m = m"
```

Combining things

We can also write:

```
1 locale monoid' = monoidr + monoidl
```

Combining things

We can also write:

```
1 locale monoid' = monoidr op unit + monoidl op unit
2   for op (infixl <.> 70) and unit (<1>)
```

Combining things

We can also write:

```
1 locale monoid' = monoidr op unit + monoidl op unit  
2   for op (infixl <.> 70) and unit (<1>)
```

(each type class is also a locale!)

Combining things

We can also write:

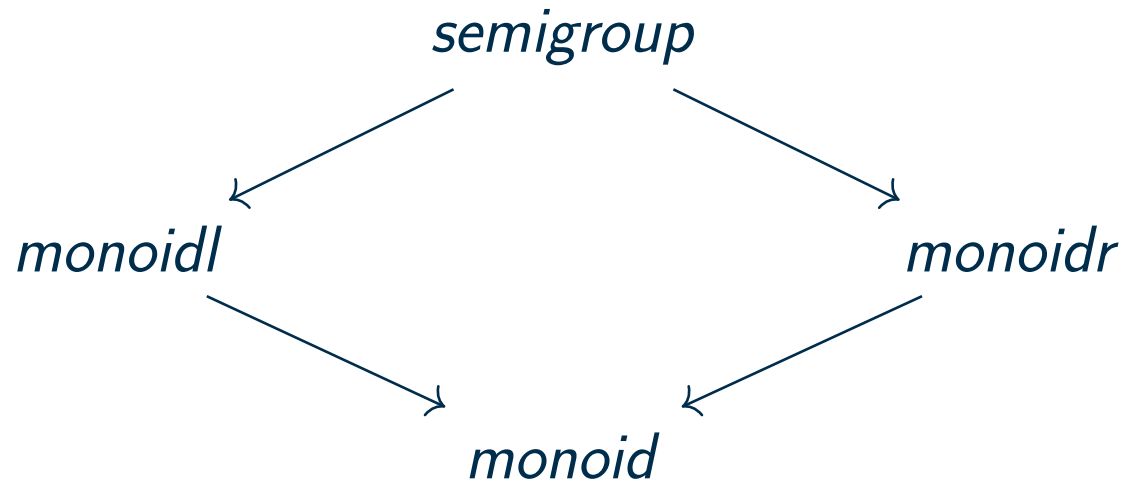
```
1 locale monoid' = monoidr op unit + monoidl op unit
2   for op (infixl <.> 70) and unit (<1>)
```

(each type class is also a locale!)

- locales can inherit from locales
- can (but don't have to) give names to those locales' parameters
- use **for** to “fix” names used for this (and give types/syntax/...)

Locale hierarchy

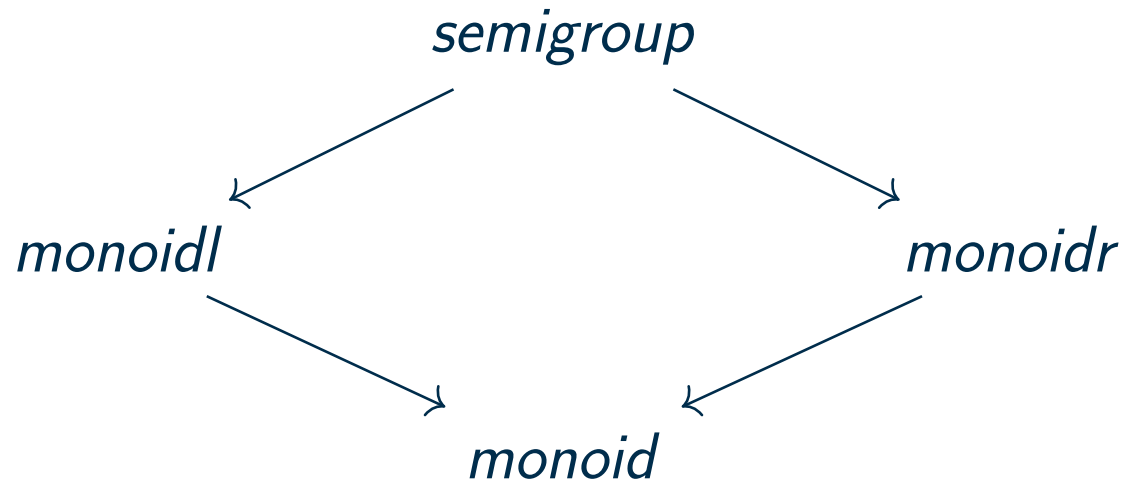
Unlike classes “inheritance diamonds” are supported



The **only** rule: assumptions of a one imply assumptions of the other

Locale hierarchy

Unlike classes “inheritance diamonds” are supported



The **only** rule: assumptions of a one imply assumptions of the other
(we can even have two equivalent locales be sublocales of each other!)

Interpreting a locale

Locales have **named** “global” interpretations:

```
1  interpretation lookup: kvmap "k f. f k" "k v f. f(k:= v)"
2    by standard auto
3
4  interpretation int_add: monoid' "(+)" "0::int"
5    by standard auto
6  interpretation int_mult: monoid' "(*)" "1::int"
7    by standard auto
```

Inherited properties can then be used outside the context by qualified names:

```
1  lemma "(1 + 2) + 3 = 1 + (2 + 3 :: int)"
2    using int_add.semigroup_assoc .
```

Local Interpretations

Can get an “instance” of a locale inside a proof, as necessary.

→ useful when working outside a locale’s context

→ or with multiple locales at once

```
1 lemma "(xs @ ys) @ zs = xs @ (ys @ zs)"
2 proof -
3   interpret concat: monoid' "@[]"
4   by standard auto
5   show ?thesis
6   using concat.semigroup_assoc .
7 qed
```

Nonsensical Locales

If you work with a locale, **make sure it has an instance!**

Else: easy to put lots of work into meaningless theories ...

E.g. this is a perfectly fine locale ...

```
1 locale nonsens = fixes a and b
2   assumes "a  $\implies$  b" "a" " $\neg$  b"
```

Theory Patterns: Set-based Locales

With locales, it's also common to use a *carrier set*:

```
1 locale semigroup =  
2   fixes M and op  
3   assumes [intro, simp]: "[ a ∈ M; b ∈ M ] ⇒ op a b ∈ M"  
4   and "[ a ∈ M; b ∈ M; c ∈ M ] ⇒ op a (op b c) = op (op a b) c"
```

Theory Patterns: Set-based Locales

With locales, it's also common to use a *carrier set*:

```
1 locale semigroup =  
2   fixes M and op  
3   assumes [intro, simp]: "[ a ∈ M; b ∈ M ] ⇒ op a b ∈ M"  
4   and "[ a ∈ M; b ∈ M; c ∈ M ] ⇒ op a (op b c) = op (op a b) c"
```

If you do “set-based” mathematics, locales are what you want

Theory Patterns: Combining Locales

It's easier to work inside locale contexts.

Theory Patterns: Combining Locales

It's easier to work inside locale contexts.

So if we talk about several locales, we can combine them:

Theory Patterns: Combining Locales

It's easier to work inside locale contexts.

So if we talk about several locales, we can combine them:

```
1 locale semigroup_morphism = source: semigroup M op + target:
  semigroup M' op'
2   for M :: "'a set" and M' :: "'b set" and op op'
3   + fixes bij_map :: "'a  $\Rightarrow$  'b" (< $\pi$ >)
4   assumes "bij_betw bij_map M M'"
5     and "[[ a  $\in$  M; b  $\in$  M ]  $\Longrightarrow$  (op a b) = op' ( $\pi$  a) ( $\pi$  b)"]
```

Theory Patterns: Combining Locales

It's easier to work inside locale contexts.

So if we talk about several locales, we can combine them:

```
1 locale semigroup_morphism = source: semigroup M op + target:
  semigroup M' op'
2   for M :: "'a set" and M' :: "'b set" and op op'
3   + fixes bij_map :: "'a  $\Rightarrow$  'b" (< $\pi$ >)
4   assumes "bij_betw bij_map M M'"
5     and "[[ a  $\in$  M; b  $\in$  M ]  $\Longrightarrow$  (op a b) = op' ( $\pi$  a) ( $\pi$  b)"]
```

- if a locale has several “instances” of another, give them names!
- names and types of parameters can be given using **for**
- (otherwise punning can cause problems ...)

Group Projects

Make sure you have topics fixed before the holidays start!

Introduction
○○○

Type Classes
○○○○○○○○○○○○○○

Locales
○○○○○○○○○

Project Topics
●○○

Group Projects

Make sure you have topics fixed before the holidays start!

You should answer these questions until then:

- At least: who's part of your group?
- What's your topic, and what do you intend to do?
- (optional: have you looked at previous work you could build on?)

Group Projects

Make sure you have topics fixed before the holidays start!

You should answer these questions until then:

- At least: who's part of your group?
- What's your topic, and what do you intend to do?
- (optional: have you looked at previous work you could build on?)

We will help you discuss topics/scope/etc.

Course Schedule

We are almost at the end of the “introduction” block

- Next week is one more regular slot
- December 17 we will have a guest talk by Michael Kirsten about Isabelle in research
- Then holidays & project work!
- (but we still have this slot, come by for questions etc.)
- Project presentations: probably last two slots (March 4 & 11)

Conclusion

You should be able to answer now:

- What type classes & locales are
- When to use either

Until next week:

- Download and work on the fifth (& final!) exercise sheet
- Submit your solution to ILIAS

See you next week! :)