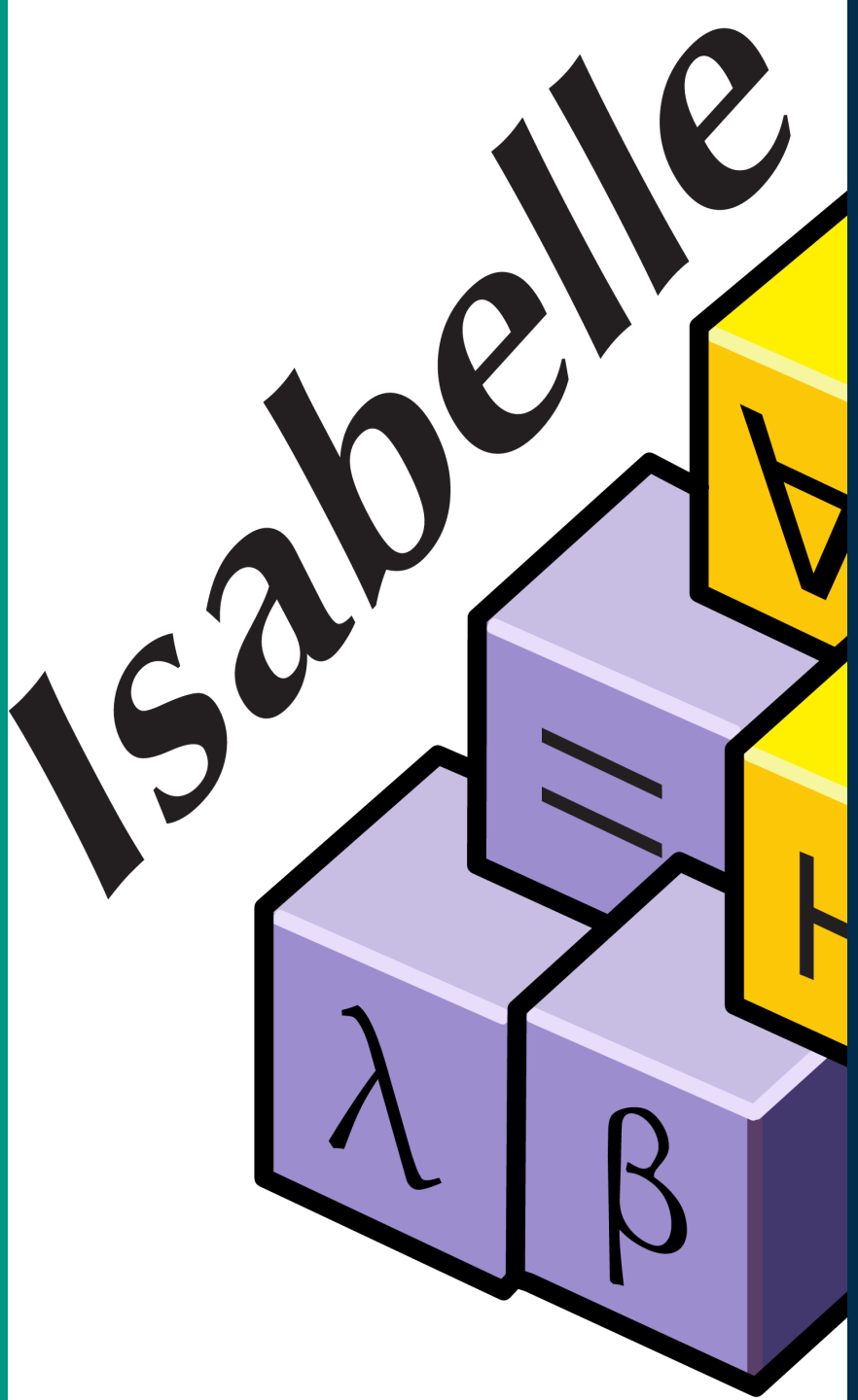
# Theorem Prover Lab




**Episode 2: Recursion & Induction**

Terru Stübinger | 2025-11-05

# Today

1. Homework
2. More on induction
3. About *simp* and *auto*

We're following Sections 3 & 4 of Tobias Nipkow's *Concrete Semantics* lecture (`http://concrete-semantics.org/`)

# Recap: Structural Induction

Last time, we had simple induction over datatypes:

```
1  datatype nat = Zero | Suc nat
2
3  lemma add_zero: "add m Zero = m"
4  proof (induction m)
5    case Zero
6    then show ?case by simp
7  next
8    case (Suc m)
9    then show ?case by simp
10 qed
```

- datatype has multiple cases
- proof follows *structure* of the type
- (and all values are finite)

$$\frac{P(0) \quad \bigwedge n.\ P(n) \longrightarrow P(\text{Suc } n)}{P(n)}$$

# Exercise1.thy

# Recap: Recursive Functions

We've also seen how to define functions:

```
fun add :: "nat ⇒ nat ⇒ nat" where
"add Zero n = n" |
"add (Suc m) n = Suc (add m n)"
```

# Recap: Recursive Functions

We've also seen how to define functions:

```
fun add :: "nat ⇒ nat ⇒ nat" where
"add Zero n = n" |
"add (Suc m) n = Suc (add m n)"
```

May have many equations:

```
fun ack :: "nat ⇒ nat ⇒ nat" where
  "ack 0 n = Suc n"
| "ack (Suc m) 0 = ack m (Suc 0)"
| "ack (Suc m) (Suc n) = ack m (ack (Suc m) n)"
```

Must prove termination! (if the automatic proof by size argument fails, you can use *function* instead and give it manually)

# Motto: Theorems about recursive functions are proved by induction

# Reversing lists

How to reverse a list?

```
1 fun rev :: "'a list ⇒ 'a list" where
2 "rev [] = []" |
3 "rev (x # xs) = rev xs @ [x]"
```

# Reversing lists

How to reverse a list?

```
fun rev :: "'a list ⇒ 'a list" where
"rev [] = []" |
"rev (x # xs) = rev xs @ [x]"
```

We can also do a tail-recursive version:

```
fun itrev :: "'a list ⇒ 'a list ⇒ 'a list" where
"itrev [] ys = ys" |
"itrev (x#xs) ys = itrev xs (x#ys)"
```

Now prove that *itrev xs [] = rev xs*!

# Induction_Demo.thy

# Stuck? → Generalise!

```
1  lemma "itrev xs [] = rev xs"
2  proof (induction xs)
3    case Nil
4    then show ?case by simp
5  next
6    case (Cons a xs)
7    have "itrev (a#xs) [] = rev (a#xs)"
8      by (* ? *)
9    then show ?case by simp
10 qed
```

# Stuck? → Generalise!

```
lemma rev_append: "itrev xs ys = rev xs @ ys"
proof (induction xs arbitrary: ys)
  case Nil
  then show ?case by simp
next
  case (Cons a xs)
  thus "itrev (a#xs) ys = rev (a#xs) @ ys"
    by simp
qed

lemma "itrev xs [] = rev xs"
  using rev_append[of xs "[]"] by simp
```

# Induction Rules (Non-Structural Induction)

So far, all proofs used *structural induction*,

# Induction Rules (Non-Structural Induction)

So far, all proofs used *structural induction*,

because all functions were *primitive-recursive*

# Induction Rules (Non-Structural Induction)

So far, all proofs used *structural induction*,

because all functions were *primitive-recursive*

```
1 fun sep :: "'a  'a list  'a list" where
2 "sep a [] = []" |
3 "sep a [x] = [x]" |
4 "sep a (x#y#zs) = x # a # sep a (y#zs)"
```

Gives *sep.induct*:

$$\frac{P\ a\ []\qquad \bigwedge a\ x.\ P\ a\ [x]\qquad \bigwedge a\ x\ y\ zs.\ P\ a\ (y\#zs) \longrightarrow P\ a\ (x\#y\#zs)}{P\ a\ xs}$$

# Induction rules follow the computation

For `f :: τ ⇒ τ'` we get an *induction schema* to prove $P(x)$ for all `x :: τ`

For each defining equation

$$f(e) = \ldots f(x_1) \ldots f(x_2) \ldots$$

$$\text{prove } P(e) \text{ assuming } P(x_1), P(x_2), \ldots$$

Generally: properties of $f$ are best proved using `f.induct`!

(note: `fun` proves termination, otherwise would be ill-founded)

# Using an induction rule

Heuristic: For an occurrence `f a b c ...` of `f` applied to parameters in a goal, we want to use

```
1  apply(induction a b c ... rule: f.induct)
```

Ideally, `a`, `b`, `c` ... are variables

# Induct_Demo.thy

# Part II: Simplication

*simp* applies rewriting rules from **left to right, as long as possible**

Rules are equations $l = r$ marked with *[simp]:* as "simp rules"

Depending on rules, the simplifier might not terminate!

You can turn on tracing: *using [[simp_trace]] apply simp*

# Conditional Rewriting

Rules may also have preconditions:

$$P_1 \Longrightarrow P_2 \Longrightarrow \cdots \Longrightarrow l = r$$

Example:

$$f\ 0 = True$$
$$f\ x \Longrightarrow g\ x = True$$

Lets us derive $g\ 0$.

# (Non-)Termination

What happens if we have a simp rule `f x = f x`?

Rules are applied **eagerly**, so might get stuck even if there's an easy solution!

Heuristic for good rules: left side should be "bigger" than right side & all preconditions

$$\text{Good: } n < m \implies \mathit{Suc}\ n \leq m = \mathit{True}$$

$$\text{Not: } \mathit{Suc}\ n \leq m \implies n < m = \mathit{True}$$

In practice, rarely an issue ...

# Specifying simp rules

On goal $P_1 \Longrightarrow P_2 \Longrightarrow \ldots \Longrightarrow C$

```
1     apply (simp add: eq1 eq2 ...)
```

will simplify $C$ and all $P_i$ using:

- Given facts *eq1 eq2 ...*
- The assumptions $P_i$
- Definitions of *fun* and *datatype*;
- *definition*s must be given explicitly as *f_def* (or marked as *[simp]*)
- Any lemma marked with *[simp]*
- (also: can ignore simp rules using *del:*)

# simp & auto

Convention: *simp* is supposed to terminate, *auto* doesn't have to

- *auto* is often more powerful
- *auto* operates on all subgoals
- *auto* can split cases
- *auto* can sometimes prove (basic) things about quantifiers (but not a lot)
- *auto* takes the same arguments, but prefixed with *simp* (so *simp add:* instead of *add:* ...)

# Simp_Demo.thy

# Conclusion

You should be able to answer now:

- How to use structural induction
- How to use induction rules
- When to use which one
- How does the simplifier work

Until next week:

- Download and work on the second exercise sheet
- Submit your solution to ILIAS

**See you next week! :)**