Rough timeline

- Intro & motivation, getting started

- Foundations & Principles
  - Lambda Calculus, natural deduction
  - Higher Order Logic
  - Term rewriting

- Proof & Specification Techniques
  - Isar
  - Inductively defined sets, rule induction
  - Datatypes, recursion, induction
  - Calculational reasoning, mathematics style proofs
  - Hoare logic, proofs about programs

Content

slide 1

slide 2

slide 3

slide 4

Example:

\[
\text{datatype 'a list} = \text{Nil} \mid \text{Cons 'a 'a list}
\]

Properties:

\[-\]

- Constructors:
  \[
  \begin{align*}
  \text{Nil} & \quad :\quad \text{'a list} \\
  \text{Cons} & \quad :\quad \text{'a} \to \text{'a list} \to \text{'a list}
  \end{align*}
  \]

- Distinctness: \(\text{Nil} \neq \text{Cons x xs}\)

- Injectivity: \((\text{Cons x xs} = \text{Cons y ys}) = (x = y \land \text{xs} = \text{ys})\)
How is this Type Defined?

datatype `a list = Nil | Cons `a `a list

- internally defined using typedef
- hence: describes a set
- set = trees with constructors as nodes
- inductive definition to characterise which trees belong to datatype

More detail: HOL/Datatype.thy

Datatype Limitations

Must be definable as set.

- Infinitely branching ok.
- Mutually recursive ok.
- Strictly positive (right of function arrow) occurrence ok.

Not ok:

datatype `t = C (t ⇒ bool)
| D (bool ⇒ t ⇒ bool)
| E (t ⇒ bool ⇒ bool)

Because: Cantor’s theorem (α set is larger than α)

Case

Every datatype introduces a case construct, e.g.

(case xs of [] ⇒ ...) | (y #ys ⇒ ...) y ... ys ...)

In general: one case per constructor

- Nested patterns allowed: x #y #z
- Dummy and default patterns with
- Binds weakly, needs () in context

Cases

apply (case_tac t)

creates k subgoals

[t = C1 x1 ... xn] ⇒ ...

one for each constructor C1
Why nontermination can be harmful

How about \( f \ x = f \ x + 1 \)?

Subtract \( f \ x \) on both sides.

\[ \Rightarrow 0 = 1 \]

All functions in HOL must be total!

---

**Principle Recursion**

\[ \text{primrec guarantees termination structurally} \]

Example primrec def:

\[ \text{primrec app :: } \text{"a list } \rightarrow \text{"a list } \rightarrow \text{"a list}\]

\[ \text{where } \]

\[ \text{"app Nil ys = ys" } | \]

\[ \text{"app (Cons x xs) ys = Cons x (app xs ys)"} \]
The General Case

If $\tau$ is a datatype (with constructors $C_1, \ldots, C_k$) then $f : \tau \Rightarrow \tau'$ can be defined by primitive recursion:

\[
f(C_1 y_1 \ldots y_{m_1}) = r_1 \\
\vdots \\
f(C_k y_1 \ldots y_{m_k}) = r_k
\]

The recursive calls in $r$, must be structurally smaller
(of the form $f a_1 \ldots y_i, \ldots a_p$)

How does this Work?

primrec just fancy syntax for a recursion operator

Example:

\[
\begin{align*}
\text{list}	ext{rec} &:: \text{"b} \Rightarrow \text{"a list} \Rightarrow \text{"b} \Rightarrow \text{"b} \\ \text{list}	ext{rec} f_1 f_2 \text{ Nil} & = f_1 \\ \text{list}	ext{rec} f_1 f_2 \text{ (Cons } x \text{ xs)} & = f_2 x \text{ xs} (\text{list}	ext{rec} f_1 f_2 \text{ xs}) \\
\text{app} & = \text{list}	ext{rec} (\lambda y. \text{ys}) (\lambda x. \text{xs} \text{ xs'}, \lambda y. \text{Cons } (x y) \text{ ys}) \\
\text{primrec app} &:: \text{"a list} \Rightarrow \text{"a list} \Rightarrow \text{"a list} \\
\text{where} \\
\text{"app} \text{ Nil } y s = y s \\
\text{"app} \text{ (Cons } x \text{ xs) } y s = \text{Cons } x \text{ (app xs ys)}
\end{align*}
\]

Predefined Datatypes

list\_rec

Defined: automatically, first inductively (set), then by epsilon

\[
\begin{align*}
(\text{Nil, } f_1) &\in \text{list\_rel } f_1 f_2 \\
(\text{Cons } x \text{ xs, } f_2 x \text{ xs } xs' ) &\in \text{list\_rel } f_1 f_2 \\
\text{list\_rec } f_1 f_2 \text{ xs } &\equiv \text{SOME } y. \ (x y) \in \text{list\_rel } f_1 f_2 \\
\text{Automatic proof that set def indeed is total function} \\
\text{(the equations for list\_rec are lemmas!)}
\end{align*}
\]
nat is a datatype

```ml
datatype nat = 0 | Suc nat
```

Functions on nat definable by primrec!

```ml
primrec
  f 0     = ...
  f (Suc n) = ... f n ...
```

---

Option

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

Important application:

```ml
'b ⇒ 'a option ∼ partial function:
  None ∼ no result
  Some a ∼ result a
```

Example:

```ml
primrec lookup :: 'k ⇒ ('k × 'v) list ⇒ 'v option
where
  lookup k []     = None |
  lookup k (x #xs) = (if fst x = k then Some (snd x) else lookup k xs)
```
Structural induction

\( P \) holds for all lists \( x \) if
\( \Rightarrow P \) Nil
\( \Rightarrow \) and for arbitrary \( x \) and \( xs \), \( P xs \Rightarrow P (x \# xs) \)

Induction theorem list.induct:
\[ [ P \]; \forall a \text{ list. } P \text{ list } \Rightarrow P (a \# \text{list}) ] \Rightarrow P \text{ list } \]

\( \Rightarrow \) General proof method for induction: (induct \( x \))
- \( x \) must be a free variable in the first subgoal.
- type of \( x \) must be a datatype.

Basic heuristics

Theorems about recursive functions are proved by induction

Induction on argument number \( i \) of \( f \)
if \( f \) is defined by recursion on argument number \( i \)

Example

A tail recursive list reverse:
primrec itrev :: \( 'a \text{ list } \Rightarrow 'a \text{ list } \Rightarrow 'a \text{ list } \)
where
itrev [] \_ = itrev \_ []
itrev (x \# xs) \_ = itrev xs (x \# \_)

lemma itrev xs [] = rev xs

DEMO: PROOF ATTEMPT
Generalisation

Replace constants by variables

\[ \text{lemma } \text{itrev } \, x \, s \, y \, s = \text{rev } \, x \, s \, @ \, y \, s \]

Quantify free variables by \( \forall \)
(\( except \) the induction variable)

\[ \text{lemma } \forall y. \, \text{itrev } \, x \, s \, y = \text{rev } \, x \, s \, @ \, y \, s \]

Slide 25

We have seen today...

- Datatypes
- Primitive recursion
- Case distinction
- Structural Induction

Slide 26