Defining Things
Introducing New Types

- **typedef**: Primitive for type definitions; Only real way of introducing a new type with new properties
  - Must build a model and prove it nonempty
  - Probably won’t use in this course

- **typedef**: Pure declaration; New type with no properties (except that it is non-empty)

- **type_synonym**: Abbreviation - used only to make theory files more readable

- **datatype**: Defines recursive data-types; solutions to free algebra specifications
Datatypes: An Example

```plaintext
datatype 'a list = Nil | Cons 'a ("'a list")
```

- **Type constructors**: list of one argument
- **Term constructors**: `Nil :: 'a list`  
  `Cons :: 'a ⇒ 'a list ⇒ 'a list`
- **Distinctness**: `Nil ≠ Cons x xs`
- **Injectivity**:  
  `(Cons x xs = Cons y ys) = (x = y ∧ xs = ys)`
Structural Induction on Lists

- To show \( P \) holds of every list
  - show \( P \text{ Nil} \), and
  - for arbitrary \( a \) and \( \text{list} \), show \( P \text{ list} \) implies \( P (\text{Cons } a \text{ list}) \)

\[
P \text{ list} \\
:\quad P \text{ Nil} \quad P (\text{Cons } a \text{ list}) \\
\hline \\P \text{ xs}
\]

In Isabelle:

\[
[\textbf{?P [ ]; } \Lambda a \text{ list. } ?P \text{ list } \rightarrow ?P (a \# \text{list}) ][] \rightarrow ?P ?\text{list}
\]
**datatype: The General Case**

\[
\text{datatype } (\alpha_1, \ldots, \alpha_m)\tau = \begin{array}{c}
C_1 \tau_{1,1} \ldots \tau_{1,n_1} \\
\vdots \\
C_k \tau_{k,1} \ldots \tau_{k,n_k}
\end{array}
\]

- **Term Constructors:**
  \[C_i :: \tau_{i,1} \Rightarrow \ldots \Rightarrow \tau_{i,n_i} \Rightarrow (\alpha_1, \ldots, \alpha_m)\tau\]
- **Distinctness:** \[C_i x_{i,1} \ldots x_{i,n_i} \neq C_j y_{j,1} \ldots y_{j,n_j} \text{ if } i \neq j\]
- **Injectivity:** \[(C_i x_1 \ldots x_{n_i} = C_i y_1 \ldots y_{n_i}) = (x_1 = y_1 \land \ldots \land x_{n_i} = y_{n_i})\]

Distinctness and Injectivity are applied by **simp**

Induction must be applied explicitly
Proof Method

- **Syntax:** \texttt{(induct_tac x)}
  
  \( x \) must be a free variable in the first subgoal
  
  The type of \( x \) must be a datatype

- **Effect:** Generates 1 new subgoal per constructor

- Type of \( x \) determines which induction principle to use
Every **datatype** introduces a **case** construct, e.g.

\[
\text{(case xs of } [ ] \Rightarrow \ldots | \ y#ys \Rightarrow \ldots y \ldots ys \ldots )
\]

In general: **case** *Arbitrarily nested pattern* \(\Rightarrow\) *Expression using pattern variables* | \ldots

Patterns may be non-exhaustive, or overlapping

Order of clauses matters - early clause takes precedence.
HOL Functions are Total

Why nontermination can be harmful:

- If $f(x)$ is undefined, is $f(x) = f(x)$?
- Excluded Middle says it must be True or False
- Reflexivity says it’s True
- How about $f(x) = 0$? $f(x) = 1$? $f(x) = y$?
- If $f(x) \neq y$ then $\forall y. f(x) \neq y$.
- Then $f(x) \neq f(x)$

! All functions in HOL must be total !
Function Definition in Isabelle/HOL

- Non-recursive definitions with `definition`
  No problem

- Well-founded recursion with `fun`
  Proved automatically, but user must take care that recursive calls are on “obviously” smaller arguments

- Well-founded recursion with `function`
  User must (help to) prove termination
  (⇝ later)

- Role your own, via definition of the functions graph
  use of choose operator, and other tedious approaches, but can work when built-in methods don’t.

- Shouldn’t need last two in this class
A Recursive Function: List Append

Declaration:

```haskell
consts app :: 'a list ⇒ 'a list ⇒ 'a list

and definition by recursion:

fun

app Nil ys = ys
app (Cons x xs) ys = Cons x (app xs ys)
```

Uses heuristics to find termination order
Guarantees termination (total function) if it succeeds
Demo: Another Datatype Example