**Defining Things**

- **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

- **typedefc**: 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

---

**Introducing New Types**

- **datatype**: The General Case
  
  \[
  \text{datatype} (\alpha_1, \ldots, \alpha_m)\tau = C_1 \tau_{1,1} \ldots \tau_{1,n_1} \\
  \quad \ldots \\
  \quad | \\
  \quad C_k \tau_{k,1} \ldots \tau_{k,n_k}
  \]

  - Type constructors: list of one argument
  - Term constructors: Nil :: 'a list
    - Cons :: 'a => 'a list => 'a list
  - Distinctness: Nil \neq Cons x xs
  - Injectivity: (Cons x xs = Cons y ys) = (x = y \land xs = ys)

---

**Structural Induction on Lists**

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

  In Isabelle:
  
  \[
  \text{??P} \quad \text{??A}\text{ list} \quad \text{??P}\text{ list} \implies \text{??P}\ (\text{??a}\text{ list}) \implies \text{??P}\ \text{??list}
  \]
Proof Method

- **Syntax:** \((\text{induct\_tac } x)\)
  - \(x\) 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

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 (\(\Rightarrow\) 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:

- \(\text{consts app :: } 'a\ \text{list} \Rightarrow 'a\ \text{list} \Rightarrow 'a\ \text{list}\)
- and definition by recursion:

  - **fun**
    - \(\text{app Nil ys = ys}\)
    - \(\text{app (Cons x xs) ys = Cons x (app xs ys)}\)

  - Uses heuristics to find termination order
  - Guarantees termination (total function) if it succeeds

Demo: Another Datatype Example