CS576 Topics in Automated Deduction

Elsa L Gunter
2112 SC, UIUC
egunter@illinois.edu
http://courses.grainger.illinois.edu/cs576

Slides based in part on slides by Tobias Nipkow

January 27, 2026

nat as a Recursive datatype

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

nat as a Recursive datatype

$$?P\ 0 \Rightarrow (\bigwedge nat. ?P\ nat \Rightarrow ?P\ (Suc\ nat)) \Rightarrow ?P\ ?nat$$

To show for `nat` that `P nat` holds

- `P 0` holds
- Pick a new (fresh) variable `n`, and
- Assuming `P n` holds, show `P (Suc n)` holds

A Recursive datatype

```
datatype 'a list = Nil    ("[ ]")
                 | Cons 'a "'a list"  (infixr "#'' 65)
```

`[ ]`: empty list

`x # xs`: list with head x::'a, tail xs::'a list

A toy list: False # (True # [ ])

Syntactic sugar: [False, True]

Concrete Syntax

When writing terms and types in `.thy` files

Types and terms need to be enclosed in `"..."`

Except for single identifiers, e.g. `'a`

`"  ..."` won't always be shown on slides

Structural Induction on Lists

P xs holds for all lists xs if

- P [ ]
- and for arbitrary y and ys, P ys implies P (y # ys)

$$\frac{\texttt{P []} \qquad \begin{array}{c}\texttt{P ys} \\ \vdots \\ \texttt{P (y \# ys)}\end{array}}{\texttt{P xs}}$$

A Recursive Function: List Append

Definition by *primitive recursion*:

```
primrec app ::  "'a list ⇒ 'a list ⇒ 'a list
where
app [] ys = ____
app (x # xs) ys = ____app xs ...____
```

One rule per constructor

Recursive calls only applied to constructor arguments

Guarantees termination (total function)

Demo: Append and Reverse

Proofs - Method 1

General schema:

```
lemma name: " ..."
apply (method)
  ⋮
done
```

If the lemma is suitable as a simplification rule:

```
lemma name[simp]: " ..."
```

Adds lemma *name* to future simplifications

Proof - Method 2

General schema:

```
lemma lemma_name:  " ..."
proof (method)
fix x y z
assume hyp1_name:  " ..."
from hyp1_name
show :  " ..."
 proof method
  ⋮
 qed
qed
```

Will try to use only Method 2 (Isar) in lectures in class

Top-down Proofs

`sorry`

- "completes" any proof (by giving up, and accepting it)
- Suitable for top-down development of theories:
- Assume lemmas first, prove them later.

Only allowed for interactive proof!

Isabelle Syntax

- Distinct from HOL syntax
- Contains HOL syntax within it
- Also the same as HOL - need to not confuse them

Theory = Module

Syntax:

```
theory MyTh
 imports ImpTh_1 ... ImpTh_n
begin

   declarations, definitions, theorems, proofs, ...

end
```

- *MyTh*: name of theory being built. Must live in file *MyTh*.thy.
- $ImpTh_i$: name of *imported* theories. Importing is transitive.

Meta-logic: Basic Constructs

Implication: $\Longrightarrow$ (`==>`)

For separating premises and conclusion of theorems / rules

Equality: $\equiv$ (`==`)

For definitions

Universal Quantifier: $\bigwedge$ (`!!`)

Usually inserted and removed by Isabelle automatically

Do not use *inside* HOL formulae

Rule/Goal Notation

$$[\,|\,A_1\,;\,\ldots\,;\,A_n\,|\,]\implies B$$

abbreviates

$$A_1 \implies \ldots \implies A_n \implies B$$

and means the rule (or potential rule):

$$\frac{A_1\,;\,\ldots\,;\,A_n}{B}$$

$;\ \approx$ "and"

Note: A theorem is a rule; a rule is a theorem.

The Proof/Goal State

1. $\Lambda x_1 \ldots x_m.\ [| A_1 ; \ldots ; A_n |] \Longrightarrow B$

$x_1 \ldots x_m$	Local constants (fixed variables)
$A_1 \ldots A_n$	Local assumptions
B	Actual (sub)goal

Proof Methods

- Simplification and a bit of logic

`auto` **Effect:** tries to solve as many subgoals as possible using simplification and basic logical reasoning

`simp` **Effect:** relatively intelligent rewriting with database of theorem, extra given theorems, and assumptions.

- More specialized tactics to come

Top-down Proofs

`sorry`

"completes" any proof (by giving up, and accepting it)
Suitable for top-down development of theories:
Assume lemmas first, prove them later.

Only allowed for interactive proof!