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 29, 2026

Theory = Module

Syntax:

```
theory MyTh
 imports ImpTh1 ... ImpThn
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.

Isabelle Syntax

- Distinct from HOL syntax
- Contains HOL syntax within it
- Mirrors HOL syntax - need to not confuse them

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

Isabelle	HOL	Meaning
$\Longrightarrow$	$\longrightarrow$	Implies
$\equiv$	$=$	Equality
$\bigwedge$	$\forall$	Universal Quantification, For All

Variables

Three kinds of variables in Isabelle:

- bound: $\forall x.\ x = x \qquad \bigwedge x.\ x > 3 \Longrightarrow x > 0$
- free: $x = x$ (only in HOL terms)
- *schematic*: $?x = ?x$
  ("unknown", a.k.a. *meta-variables*)

Can be mixed in term or formula: $\forall b.\ \exists y.\ f\ ?a\ y = b$

Variables

- Logically: free = bound at meta-meta-level
- Operationally:
  - free variabes are fixed
  - schematic variables are instantiated by substitutions

From x to ?x

State lemmas with free variables:

```
lemma app_Nil2 [simp]:  "xs @ [ ] = xs"
  ⋮
done
```

After the proof: Isabelle changes xs to ?xs (internally):

```
?xs @ [ ] = ?xs
```

Now usable with arbitrary values for ?xs

Example: rewriting

```
rev(a @ [ ]) = rev a
```

using app_Nil2 with $\sigma = \{?\mathtt{xs} \mapsto \mathtt{a}\}$

Rule/Goal Notation

$$\llbracket A_1; \ldots; A_n \rrbracket \Longrightarrow B$$

abbreviates

$$A_1 \Longrightarrow \ldots \Longrightarrow A_n \Longrightarrow B$$

and means the rule (or potential rule):

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

; ≈ "and"

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

The Proof/Goal State

1. $\bigwedge x_1 \ldots x_m.\ \llbracket A_1; \ldots; A_n \rrbracket \Longrightarrow B$

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

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

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!

Defining Things

Introducing New Types

Keywords:

- `typedecl`: Pure declaration; New type with no properties (except that it is non-empty)
- `typedef`: Primitive for type definitions; Only real way of introducing a new type with new properties
  Must build a model and prove it nonempty
  More on this later
- `type_synonym`: Abbreviation - used only to make theory files more readable
- `datatype`: Defines recursive data-types; solutions to free algebra specifications
  Basis for primitive recursive function definitions
- `record:` introduces a record type scheme, introducing its fields. To be covered later.

typedecl

`typedecl` *name*
Introduces new "opaque" *name* without definition
Serves similar role for generic reasoning as polymorphism, but can't be specialized

Example:

`typedecl addr` — An abstract type of addresses

type_synonym

`type_synonym` ⟨*tyvars*⟩ *name* = τ
Introduces an abbreviation ⟨*tyvars*⟩ *name* for type τ

Examples:

```
type_synonym  name = string
type_synonym  ('a,'b)foo = "'a list * 'b"
```

Type abbreviations are expanded immediately after parsing

Not present in internal representation and Isabelle output

datatype: An Example

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

Properties:

- 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

`P xs` holds for all lists `xs` if

- `P Nil`, and
- for arbitrary `a` and `list`, `P list` implies `P (Cons a list)`

$$\dfrac{\texttt{P Nil} \qquad \begin{array}{c}\texttt{P ys}\\ \vdots \\ \texttt{P (Cons y ys)}\end{array}}{\texttt{P xs}}$$

In Isabelle:

$[\![?P[];\ \bigwedge a\ list.\ ?P\ list \Longrightarrow ?P(a \# list)\ ]\!] \Longrightarrow ?P\ ?list$

datatype: The General Case

$\texttt{datatype}\ \ (\alpha_1, \ldots, \alpha_m)\tau \ = \ C_1\ \tau_{1,1} \ldots \tau_{1,n_1}$
$\quad | \ \ldots$
$\quad | \ C_k\ \tau_{k,1} \ldots \tau_{k,n_k}$

- Term Constructors: $C_i :: \tau_{i,1} \Rightarrow \ldots \Rightarrow \tau_{i,n_i} \Rightarrow (\alpha_1, \ldots, \alpha_m)\tau$
- Distinctness: $C_i\ x_i \ldots x_{i,n_i} \neq C_j\ y_j \ldots y_{j,n_j}$ if $i \neq j$
- Injectivity: $(C_i\ x_1 \ldots x_{n_i} = C_i\ y_1 \ldots y_{n_i}) = (x_1 = y_1 \wedge \ldots \wedge x_{n_i} = y_{n_i})$

Distinctness and Injectivity are applied by `simp`
Induction must be applied explicitly

Proof Method

- Structural Induction
  - **Syntax:** `(induct 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

case

Every `datatype` introduces a `case` construct, e.g.

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

In general:

`case` *Arbitrarily nested pattern* ⇒ *Expression using pattern variables*
| *Another pattern* ⇒ *Another expression*
| ...

Patterns may be non-exhaustive, or overlapping
Order of clauses matters - early clause takes precedence.

Case Distinctions

`apply` / `proof` (`case_tac` *t*)

creates k subgoals:

$$t = C_i\ x_1 \ldots x_{n_i} \Longrightarrow \ldots$$

one for each constructor C_i

Demo: Another Datatype Example

Definitions by Example

Definition:

```
definition lot_size::"nat * nat" where
  "lot_size ≡ (62, 103)"

definition sq::"nat ⇒ nat" where
  sq_def:  "sq n ≡ n * n"
```

The ASCII for ≡ is ==.
Definitions of form $f\ x_1 \ldots x_n \equiv t$ where t only uses $x_1 \ldots x_n$ and previously defined constants.
Creates theorem with default name *f*_def

Definition Restrictions

```
definition prime ::  "nat ⇒ bool" where
  "prime p ≡ 1<p ∧ (m dvd p ⟶ m = 1 ∨ m = p)"
```

Not a definition: m free, but not on left

! Every free variable on rhs must occur as argument on lhs !

```
  "prime p ≡ 1<p ∧ (∀ m.  m dvd p ⟶ m = 1 ∨ m = p)"
```

Note: no recursive definitions with `definition`

Using Definitions

Definitions are not used automatically

Unfolding of definition of sq:

`proof (unfold sq_def)`

Rewriting definition of sq out of current goal:

`proof (simp add:  sq_def)`

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 ≠ y then ∀y. f x ≠ y. Then fx ≠ fx #

! All functions in HOL must be total !

Function Definition in Isabelle/HOL

- Non-recursive definitions with `definition`
  No problem
- Primitive-recursive (over datatypes) with `primrec`
  Termination proved automatically internally
- Well-founded recursion with `fun`
  Proved automatically, but user must take care that recursive calls are on "obviously" smaller arguments

Function Definition in Isabelle/HOL

- 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.