

CS576 Topics in Automated Deduction

Elsa L Gunter
2112 SC, UIUC
egunter@illinois.edu

<http://courses.engr.illinois.edu/cs576>

Slides based in part on slides by Tobias Nipkow

January 30, 2015

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: $\implies$ ($\Rightarrow$)

For separating premises and conclusion of theorems / rules

Equality: $\equiv$ ($=$)

For definitions

Universal Quantifier: $\wedge$ ($!!$)

Usually inserted and removed by Isabelle automatically

Do not use *inside* HOL formulae

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

Variables

Three kinds of variables in Isabelle:

- bound: $\forall x. x = x$ $\wedge x. x > 3 \implies x > 0$
- 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 variables 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

$$\text{rev}(a @ []) = \text{rev } a$$

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

Rule/Goal Notation

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

abbreviates

$$A_1 \Longrightarrow \dots \Longrightarrow A_n \Longrightarrow B$$

and means the rule (or potential rule):

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

; $\approx$ “and”

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

The Proof/Goal State

$$1. \wedge x_1 \dots x_m. \llbracket A_1; \dots; A_n \rrbracket \Longrightarrow B$$

$x_1 \dots x_m$ Local constants (fixed variables)

$A_1 \dots A_n$ 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:

- **typedec1**: 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.

typedefcl

`typedefcl` *name*

Introduces new “opaque” *name* without definition

Serves similar role for generic reasoning as polymorphism, but can't be specialized

Example:

`typedefcl` addr — An abstract type of addresses

type_synonym

`type_synonym` $\langle tyvars \rangle$ *name* = τ

Introduces an abbreviation $\langle tyvars \rangle$ *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  $\Rightarrow$  'a list  $\Rightarrow$  'a list`
- Distinctness: `Nil` $\neq$ `Cons x xs`
- Injectivity:
 $(\text{Cons } x \text{ xs} = \text{Cons } y \text{ ys}) = (x = y \wedge \text{xs} = \text{ys})$

Structural Induction on Lists

P xs holds for all lists xs if

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

$$\frac{\begin{array}{c} P\ ys \\ \vdots \\ P\ Nil \quad P\ (Cons\ y\ ys) \end{array}}{P\ xs}$$

In Isabelle:

$$\llbracket ?P[]; \wedge a\ list. ?P\ list \Longrightarrow ?P(a \# list) \rrbracket \Longrightarrow ?P\ ?list$$

datatype: The General Case

$$\begin{array}{lcl} \text{datatype } (\alpha_1, \dots, \alpha_m)\tau & = & C_1 \tau_{1,1} \dots \tau_{1,n_1} \\ & & | \dots \\ & & C_k \tau_{k,1} \dots \tau_{k,n_k} \end{array}$$

- Term Constructors:

$$C_i :: \tau_{i,1} \Rightarrow \dots \Rightarrow \tau_{i,n_i} \Rightarrow (\alpha_1, \dots, \alpha_m)\tau$$

- Distinctness: $C_i x_i \dots x_{i,n_i} \neq C_j y_j \dots y_{j,n_j}$ if $i \neq j$

- Injectivity: $(C_i x_1 \dots x_{n_i} = C_i y_1 \dots y_{n_i}) =$
 $(x_1 = y_1 \wedge \dots \wedge x_{n_i} = y_{n_i})$

Distinctness and Injectivity are applied by **simp**

Induction must be applied explicitly

- 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 [ ]  $\Rightarrow$  ... | y#ys  $\Rightarrow$  ...y ...ys ...)
```

In general:

```
case Arbitrarily nested pattern  $\Rightarrow$  Expression using pattern variables  
| Another pattern  $\Rightarrow$  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:

$$t = C_i x_1 \dots x_{n_i} \implies \dots$$

one for each constructor C_i

Demo: Another Datatype Example

Definitions by Example

Definition:

```
definition lot_size::"nat * nat" where  
  "lot_size  $\equiv$  (62, 103)"
```

```
definition sq::"nat  $\Rightarrow$  nat" where  
  sq_def: "sq n  $\equiv$  n * n"
```

The ASCII for $\equiv$ is ==.

Definitions of form $f\ x_1 \dots x_n \equiv t$ where t only uses $x_1 \dots x_n$ and previously defined constants.

Creates theorem with default name f_def

Definition Restrictions

```
definition prime :: "nat  $\Rightarrow$  bool" where  
  "prime p  $\equiv$  1 < p  $\wedge$  (m dvd p  $\longrightarrow$  m = 1  $\vee$  m = p)"
```

Not a definition: m free, but not on left

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

```
"prime p  $\equiv$  1 < p  $\wedge$  ( $\forall$  m. m dvd p  $\longrightarrow$  m = 1  $\vee$  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 $f\ x \neq y$ then $\forall y. f\ x \neq y$. Then $fx \neq fx$ #

! 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
($\rightsquigarrow$ later)
- Roll 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.