Topics in Automated Deduction (CS 576)

Elsa L. Gunter

2112 Siebel Center

egunter@cs.uiuc.edu

http://www.cs.uiuc.edu/class/sp06/cs576/

Theory = Module

Syntax:

theory $MyTh$ **=** $ImpTh_1$**+**$\ldots$**+**$ImpTh_n$**:**

(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.

Contrete Syntax

When writing terms and types in `.thy` files (or an Isabelle shell):

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

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

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

Proofs

General schema:

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

If the lemma is suitable as a simplification rule:

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

Adds lemma *name* to future simplifications

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

$$[\!|\, A_1 ; \ldots ; A_n \,|\!] \Longrightarrow B$$

abbreviates

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

and means the rule (or potential rule):

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

$$; \quad \approx \quad \text{“and”}$$

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

The Proof/Goal State

1. $\bigwedge x_1 \dots x_m.\ [| A_1; \dots ; A_n |] \Longrightarrow B$

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

Proof Methods

- Simplification and a bit of logic
  - **Syntax:** `auto`
  - **Effect:** tries to solve as many subgoals as possible using simplification and basic logical reasoning
- 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!

A Recursive `datatype`

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

`Nil:` empty list

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

A toy list: Cons False (Cons True Nil)

Syntactic sugar: [False, True]

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 []; !!a list.  ?P list ==> ?P (a # list) |] ==> ?P ?list
```

A Recursive Function: List Append

Declaration:

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

and definition by *primitive recursion*:

```
primrec
app Nil ys = ____
app (Cons x xs) ys = ____app xs ...____
```

One rule per constructor

Recursive calls only applied to constructor arguments

Guarantees termination (total function)

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

Demo: Append and Reverse

Introducing New Types

Keywords:

- `typedef`: Primitive for type definitions; Only real way of introducing a new type with new properties More on this later
- `typedecl`: Pure declaration; New type with no properties (expect that it is non-empty)

Introducing New Types

Keywords:

- `types`: Abbreviation - may be used in constant declarions
- `datatype`: Defines recursive data-types; solutions to free algebra specificaitons

  Basis for primitive recursive function definitions

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

types

types $\langle$*tyvars*$\rangle$ *name* $= \tau$

Introduces an abbreviation $\langle$*tyvars*$\rangle$ *name* for type τ

Examples:

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

Type abbreviations are expanded immediately after parsing

Not present in internal representation and Isabelle output

datatype: The Example

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

Properties:

- Type 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 ∧ xs = ys)`

datatype: The General Case

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

- Type 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 automatically

Induction must be applied explicitly

Definitions by Example

Declaration: **consts**

```
lot_size :: "nat * nat"
sq : "nat ⇒ nat"
```

Definition: **defs**

```
"lot_size ≡ (62, 103)"
sq_def: "sq n ≡ n * n"
```

Declarations + definitions: **constdefs**

```
lot_size :: "nat * nat"
lot_size_def: "lot_size ≡ (62, 103)"
sq : "nat ⇒ nat"
sq_def: "sq n ≡ n * n"
```

Definition Restrictions

```
constdefs
prime :: "nat ⇒ bool"
"prime p ≡ p<1 ∧ (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 ≡ p<1 ∧ (∀ m. m dvd p ⟶ m = 1 ∨ m = p)"
```

Note: no recursive definitions with `defs` or `constdefs`

Using Definitions

Definitions are not used automatically

Unfolding of definition of `sq`:

```
apply (unfold 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 $\not{f}$ `x = y` then $\forall$`y. f x = y`. Then **$\not{f}\,x = f\,x$ #**

! All functions in HOL must be total **!**

Function Definition in Isabelle/HOL

- Non-recursive definitions with `defs`/`constdefs`

  No problem

- Primitive-recursive (over datatypes) with `primrec`

  Termination proved automatically internally

- Well-founded recursion with `recdef`

  User must (help to) prove termination

  ($\leadsto$ later)

`primrec` Example

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

`primrec`: The General Case

If τ is a `datatype` with constructors $C_1, \ldots, C_k$, then $f :: \cdots \Rightarrow \tau \Rightarrow \tau'$ can be defined by *primitive recursion* by:

$$f\ x_1 \ldots (C_1\ y_{1,1} \ldots y_{1,n_1}) \ldots x_m = r_1$$
$$\cdots$$
$$f\ x_1 \ldots (C_k\ y_{k,1} \ldots y_{k,n_k}) \ldots x_m = r_k$$

The recursive calls in r_i must be *structurally smaller*, i.e. of the form $f\ a_1 \ldots y_{i,j} \ldots a_m$.

nat is a datatype

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

Functions on `nat` are definable by `primrec`!

```
primrec
f 0 = ...
f (Suc n) = ...f n ...
```

Type `option`

```
datatype 'a option = None | Some 'a
```

Important application:

$$
\begin{aligned}
\ldots \Rightarrow \texttt{'a option} &\approx \text{partial function:} \\
\texttt{None} &\approx \text{no result} \\
\texttt{Some x} &\approx \text{result of } \texttt{x}
\end{aligned}
$$

option Example

```
consts lookup ::  'k ⇒ ('k×'v)list ⇒ 'v option
primrec
lookup k [ ] = None
lookup k (x#xs) =
(if fst x = k then Some(snd x) else lookup k xs)
```

case

Every datatype introduces a case construct, e.g.

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

In general: one case per constructor

Same number of cases as in datatype

No nested patterns (e.g. x# y# zs)

Nested cases are allowed

Needs ( ) in context

Case Distinctions

`apply (case_tac t)`

creates k subgoals:

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

one for each constructor C_i