Topics in Automated Deduction (CS 576)

Elsa L. Gunter
2112 Siebel Center
egunter@illinois.edu
http://www.cs.illinois.edu/class/sp10/cs576/

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.

Basic 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

Isar Proofs

General schema:

```
lemma name:
fixes variables . . .
assumes name: "assumption"
shows "goal"
proof method
assume name: "assumption"
from name . . .
have name: "assumption"
proof
show "goal"
proof
qed
```


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):

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

; $\approx$ "and"

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

The Proof/Goal State

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

$x_1 \ldots x_m$	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 thoerems, 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:

- `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
- `typedecl`: Pure declaration; New type with no properties (expect that it is non-empty)

Introducing New Types

Keywords:

- `types`: Abbreviation - used only to make theory files more readable
- `datatype`: Defines recursive data-types; solutions to free algebra specificaitons
  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

types

`types` ⟨*tyvars*⟩ *name* = τ
Introduces an abbreviation ⟨*tyvars*⟩ *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` ≠ `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)`

$$\frac{\begin{array}{cc}\texttt{P Nil} & \texttt{P (Cons y ys)} \\ & \vdots \\ & \texttt{P ys}\end{array}}{\texttt{P xs}}$$

In Isabelle:

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


datatype: The General Case

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

- 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* ⇒ *Expresion using pattern variables* `|` ...
Patterns may be non-exhaustive, or overlapping
Order of clauses matters - early clause takes precedence.

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

Demo: Another Datatype Example

Definitions by Example

Definition:

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

defintion 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`:

```
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 `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
- 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)
- Role your own, via definition of the functions graph
  tedious method, but can work when other methods don't.

`primrec` Example

```
primrec app :: "'a list ⇒ 'a list ⇒ 'a list
where
"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:

$$\begin{array}{l} f\ x_1 \ldots (C_1\ y_{1,1} \ldots y_{1,n_1}) \ldots x_m = r_1 \ | \\ \qquad \ldots \\ f\ x_1 \ldots (C_k\ y_{k,1} \ldots y_{k,n_k}) \ldots x_m = r_k \end{array}$$

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::nat⇒ ... where
f 0 = ... |
f (Suc n) = ...f n ...
```


Type option

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

Important application:

... ⇒ 'a option ≈ partial function:
None ≈ no result
Some x ≈ result of x

option Example

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