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

February 4, 2015
datatype: An Example

```haskell
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

\( P \ \text{xs} \) holds for all lists \( \text{xs} \) if

- \( P \ \text{Nil} \), and

- for arbitrary \( a \) and \( \text{list} \), \( P \ \text{list} \) implies \( P \ (\text{Cons} \ a \ \text{list}) \)

\[
\begin{align*}
P \ \text{Nil} & \\
P \ \text{Cons} \ y \ \text{ys} & \\
P \ \text{xs} & \\
\end{align*}
\]

In Isabelle:

\[
\begin{align*}
\text{\textbf{\{} \ ?P[]; \ \lambda \ a. \ ?P \ \text{list} \ \rightarrow \ ?P(a \ \# \ \text{list}) \ \text{\textbf{\}}} & \ \rightarrow \ ?P \ ?\text{list}
\end{align*}
\]
### datatype: The General Case

\[
\text{datatype } (\alpha_1, \ldots, \alpha_m)\tau = \begin{cases} 
C_1 \tau_{1,1} \ldots \tau_{1,n_1} \\
\vdots \\
C_k \tau_{k,1} \ldots \tau_{k,n_k}
\end{cases}
\]

- **Term Constructors:**
  
  \[C_i :: \tau_{i,1} \Rightarrow \ldots \Rightarrow \tau_{i,n_i} \Rightarrow (\alpha_1, \ldots, \alpha_m)\tau\]

- **Distinctness:**
  \[C_i x_{i,1} \ldots x_{i,n_i} \neq C_j y_{j,1} \ldots y_{j,n_j} \text{ if } i \neq j\]

- **Injectivity:**
  \[(C_i x_{1,1} \ldots x_{n_i} = C_i y_{1,1} \ldots y_{n_i}) = (x_1 = y_1 \land \ldots \land x_{n_i} = y_{n_i})\]

Distinctness and Injectivity are applied by **simp**

Induction must be applied explicitly
Proof Method

- **Structural Induction**

  - **Syntax:** \( \text{induct } x \)
    - \( x \) 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
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.
apply / proof (case_tac \( t \))

creates \( k \) subgoals:

\[
t = C_i \, x_1 \ldots x_n_i \implies \ldots
\]

one for each constructor \( C_i \)
Demo: Another Datatype Example
Definitions by Example

Definition:

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

```plaintext
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 \( f \_ \text{def} \)
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
Definitions are not used automatically

Unfolding of definition of \textit{sq}:

\begin{verbatim}
proof (unfold sq_def)
\end{verbatim}

Rewriting definition of \textit{sq} out of current goal:

\begin{verbatim}
proof (simp add: sq_def)
\end{verbatim}
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 \( f \ x = 0? \quad f \ x = 1? \quad f \ x = y? \)

If \( f \ x \not= y \) then \( \forall y. f \ x \not= y \). Then \( f x \not= fx \) #

! All functions in HOL must be total !
Non-recursive definitions with \texttt{definition}
No problem

Primitive-recursive (over datatypes) with \texttt{primrec}
Termination proved automatically internally

Well-founded recursion with \texttt{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.
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 $\tau$ is a datatype with constructors $C_1, \ldots, C_k$, then $f :: \cdots \Rightarrow \tau \Rightarrow \tau'$ can be defined by primitive recursion by:

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

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

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

Important application:

```
... ⇒ 'a option ≅ partial function:
  None ≅ no result
  Some x ≅ result of x
```
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)
Term Rewriting

Term rewriting means . . .

Terminology: equation becomes *rewrite rule*

Using a set of equations $l = r$ from left to right

As long as possible (possibly forever!)
Example

Equations:

\[ 0 + n = n \quad (1) \]
\[ (\text{Suc } m) + n = \text{Suc}(m + n) \quad (2) \]
\[ (0 \leq m) = \text{True} \quad (3) \]
\[ (\text{Suc } m \leq \text{Suc } n) = (m \leq n) \quad (4) \]

Rewriting:

\[ 0 + \text{Suc } 0 \leq \text{Suc } 0 + x \quad (1) \]
\[ \text{Suc } 0 \leq \text{Suc } 0 + x \quad (2) \]
\[ \text{Suc } 0 \leq \text{Suc}(0 + x) \quad (4) \]
\[ 0 \leq 0 + x \quad (3) \]

True
Rewriting: More Formally

\[ \text{substitution} = \text{mapping of variables to terms} \]

- \( l = r \) is \textit{applicable} to term \( t[s] \) if there is a substitution \( \sigma \) such that \( \sigma(l) = s \)
  - \( s \) is an instance of \( l \)
- Result: \( t[\sigma(r)] \)
- Also have theorem: \( t[s] = t[\sigma(r)] \)
Example

- Equation: $0 + n = n$
- Term: $a + (0 + (b + c))$
- Substitution: $\sigma = \{n \mapsto b + c\}$
- Result: $a + (b + c)$
- Theorem: $a + (0 + (b + c)) = a + (b + c)$
Conditional Rewriting

Rewrite rules can be conditional:

\[ [P_1; \ldots; P_n] \iff l = r \]

is *applicable* to term $t[s]$ with substitution $\sigma$ if:

- $\sigma(l) = s$ and
- $\sigma(P_1), \ldots, \sigma(P_n)$ are provable (possibly again by rewriting)
Three kinds of variables in Isabelle:

- **bound**: $\forall x. x = x$
- **free**: $x = x$
- **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-level
- Operationally:
  - free variables are fixed
  - schematic variables are instantiated by substitutions
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 \(\text{app}\_\text{Nil2}\) with \(\sigma = \{?xs \mapsto a\}\)
Basic Simplification

Goal: \[ [P_1; \ldots ; P_m] \Rightarrow C \]

proof (simp add: eq_thm_1 \ldots eq_thm_n)

Simplify (mostly rewrite) \( P_1; \ldots ; P_m \) and \( C \) using

- lemmas with attribute simp
- rules from primrec and datatype
- additional lemmas eq_thm_1 \ldots eq_thm_n
- assumptions \( P_1; \ldots ; P_m \)

Variations:

- (simp \ldots del: \ldots) removes simp-lemmas
- add and del are optional
Basic Simplification

Goal: 1. $[P_1; \ldots; P_m] \implies C$

proof (simp add: eq_thm_1 \ldots eq_thm_n)

Simplify (mostly rewrite) $P_1; \ldots; P_m$ and $C$ using

- lemmas with attribute simp
- rules from primrec and datatype
- additional lemmas eq_thm_1 \ldots eq_thm_n
- assumptions $P_1; \ldots; P_m$

Variations:

- (simp ... del: ...) removes simp-lemmas
- add and del are optional
auto versus simp

- **auto** acts on all subgoals
- **simp** acts only on subgoal 1
- **auto** applies **simp** and more
  - **simp** concentrates on rewriting
  - **auto** combines rewriting with resolution
Simplification may not terminate. Isabelle uses simp-rules (almost) blindly left to right. Example: $f(x) = g(x)$, $g(x) = f(x)$ will not terminate.

$$[P_1, \ldots, P_n] \Rightarrow l = r$$

is only suitable as a simp-rule only if $l$ is “bigger” than $r$ and each $P_i$.

$(n < m) = (\text{Suc} n < \text{Suc} m)$ NO
$(n < m) \Rightarrow (n < \text{Suc} m) = \text{True}$ YES
$\text{Suc} n < m \Rightarrow (n < m) = \text{True}$ NO
Assumptions and Simplification

Simplification of $[A_1, \ldots, A_n] \implies B$:

- Simplify $A_1$ to $A_1'$
- Simplify $[A_2, \ldots, A_n] \implies B$ using $A_1'$
Ignoring Assumptions

Sometimes need to ignore assumptions; can introduce non-termination. How to exclude assumptions from simp:

proof (simp (no_asm_simp)...)  
   Simplify only the conclusion, but use assumptions

proof (simp (no_asm_use)...)  
   Simplify all, but do not use assumptions

proof (simp (no_asm)...)
   Ignore assumptions completely
Definitions do not have the `simp` attribute. They must be used explicitly:

```
proof (simp add: f_def ...)
```
Ordered Rewriting

Problem: $x + y = y + x$ does not terminate
Solution: Permutative simp-rules are used only if the term becomes lexicographically smaller.
Example: $b + a \leadsto a + b$ but not $a + b \leadsto b + a$.
For types nat, int, etc., commutative, associative and distributive laws built in.
Example: proof simp yields:

\[
((B + A) + ((2 :: nat) \times C)) + (A + B) \leadsto \\
\ldots \leadsto 2 \times A + (2 \times B + 2 \times C)
\]
simp-rules are preprocessed (recursively) for maximal simplification power:

\[ \neq A \iff A = \text{False} \]
\[ A \rightarrow B \iff A \Rightarrow B \]
\[ A \land B \iff A, B \]
\[ \forall x. A(x) \iff A(?x) \]
\[ A \iff A = \text{True} \]

Example:

\[ (p \rightarrow q \land \neg r) \land s \rightarrow p \Rightarrow q = \text{True}, r = \text{True}, s = \text{True} \]
Demo: Simplification through Rewriting