Topics in Automated Deduction
(CS 576)

Elsa L. Gunter
2112 Siebel Center
eyunter@cs.uiuc.edu
http://www.cs.uiuc.edu/class/sp06/cs576/
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 \( f \ x \neq y \) for arbitrary \( y \), 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 `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
  (⇒ later)
primrec Example

primrec

"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:

\[
f \ x_1 \cdots (C_1 \ y_{1,1} \cdots y_{1, n_1}) \cdots x_m = r_1
\]

\[
\vdots
\]

\[
f \ x_1 \cdots (C_k \ y_{k,1} \cdots y_{k, n_k}) \cdots x_m = r_k
\]

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

\[ \ldots \Rightarrow \text{'a option} \approx \text{partial function:} \]
\[ \text{None} \approx \text{no result} \]
\[ \text{Some } x \approx \text{result of } x \]
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)
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**

— Cases in same order as constructors in **datatype**

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

Nested cases are allowed

Needs ( ) in context
apply (case_tac \( t \))

creates \( k \) subgoals:

\[
t = C_i \ x_1 \ldots x_{n_i} \Rightarrow \ldots
\]

one for each constructor \( C_i \)
Demo: Trees
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:

\[ 0 + n = n \] \hspace{1cm} (1)

\[ (\text{Suc } m) + n = \text{Suc}(m + n) \] \hspace{1cm} (2)

\[ (0 \leq m) = \text{True} \] \hspace{1cm} (3)

\[ (\text{Suc } m \leq \text{Suc } n) = (m \leq n) \] \hspace{1cm} (4)

Rewriting:

\[ 0 + \text{Suc } 0 \leq \text{Suc } 0 + x \] \hspace{1cm} (1)

\[ \text{Suc } 0 \leq \text{Suc } 0 + x \] \hspace{1cm} (2)

\[ \text{Suc } 0 \leq \text{Suc}(0 + x) \] \hspace{1cm} (4)

\[ 0 \leq 0 + x \] \hspace{1cm} (3)

\[ \text{True} \]
Rewriting: More Formally

\textit{substitution} = 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:

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

is \textit{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)
Variables

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 $\equiv$ bound at meta-level

- Operationally:
  - free variables are fixed
  - schematic variables are instantiated by substitutions
From $x$ to $?x$

State lemmas with free variables:

```isar
lemma app_Nil2 [simp]: "xs @ [ ] = xs"
```

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 $\sigma = \{ ?xs \mapsto a \}$
Basic Simplification

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

apply (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
**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
Termination

Simplification may not terminate.
Isabelle uses \texttt{simp}-rules (almost) blindly left to right.
Example: \( f(x) = g(x), \ g(x) = f(x) \) will not terminate.

\[
\begin{array}{c}
\{ P_1, \ldots P_n \} \implies l = r \\
\end{array}
\]

is only suitable as a \texttt{simp}-rule only if \( l \) is “bigger” than \( r \) and each \( P_i \).

\[
\begin{array}{c}
(n < m) = (\text{Suc}n < \text{Suc}m) \quad \text{NO} \\
(n < m) \implies (n < \text{Suc}m) = \text{True} \quad \text{YES} \\
\text{Suc}n < m \implies (n < m) = \text{True} \quad \text{NO}
\end{array}
\]
Assumptions and Simplification

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

- Simplify $A_1$ to $A'_1$
- Simplify $[A_2, \ldots, A_n] \Rightarrow B$ using $A'_1$
Ignoring Assumptions

Sometimes need to ignore assumptions; can introduce non-termination.

How to exclude assumptions from simp:

apply (simp (no_asm simp)...)  

Simplify only the conclusion, but use assumptions

apply (simp (no_asm_use)...)  

Simplify all, but do not use assumptions

apply (simp (no_asm)...)

Ignore assumptions completely
Rewriting with Definitions (**constdefs**)  

Definitions do not have the `simp` attribute.  

They must be used explicitly:  

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

Problem: \(?x + ?y = ?y + ?x\) does not terminate
Solution: Permutative \(\text{simp}\)-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 \(\text{nat}, \text{int}, \) etc., commutative, associative and distributive laws built in.
Example: \text{apply simp \yields:}
\[
((B + A) + ((2 :: \text{nat}) * C)) + (A + B) \leadsto \\
... \leadsto 2 * A + (2 * B + 2 * C)
\]
Preprocessing

simp-rules are preprocessed (recursively) for maximal simplification power:

\[ \not= A \mapsto A = \text{False} \]

\[ A \rightarrow B \mapsto A \implies B \]

\[ A \land B \mapsto A, B \]

\[ \forall x. A(x) \mapsto A(\text{?}x) \]

\[ A \mapsto A = \text{True} \]

Example:

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