Topics in Automated Deduction (CS 576)

Elsa L. Gunter

2112 Siebel Center

egunter@cs.uiuc.edu

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

HOL Functions are Total

Why nontermination can be harmful:

If $\texttt{f x}$ is undefined, is $\texttt{f x} = \texttt{f x}$?

Excluded Middle says it must be **True** or **False**

Reflexivity says it's **True**

How about $\texttt{f x} = 0$? $\texttt{f x} = 1$? $\texttt{f x} = \texttt{y}$? If $\texttt{f x} \neq \texttt{y}$ for arbitrary $\texttt{y}$, then $\forall \texttt{y}.\ \texttt{f x} \neq \texttt{y}$. Then $\mathbf{f\ x} \neq \mathbf{f\ x}$ $\#$

! 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` [ ] $\Rightarrow$...| y#ys $\Rightarrow$ ...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

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: Trees

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:

$$\begin{array}{rcll} 0 + n & = & n & (1) \\ (\texttt{Suc}\ m) + n & = & \texttt{Suc}(m + n) & (2) \\ (0 \leq m) & = & \texttt{True} & (3) \\ (\texttt{Suc}\ m \leq \texttt{Suc}\ n) & = & (m \leq n) & (4) \end{array}$$

Rewriting:

$$\begin{array}{rcll} 0 + \texttt{Suc}\ 0 & \leq & \texttt{Suc}\ 0 + x & \underline{\underline{(1)}} \\ \texttt{Suc}\ 0 & \leq & \texttt{Suc}\ 0 + x & \underline{\underline{(2)}} \\ \texttt{Suc}\ 0 & \leq & \texttt{Suc}(0 + x) & \underline{\underline{(4)}} \\ 0 & \leq & 0 + x & \underline{\underline{(3)}} \\ & \texttt{True} & & \end{array}$$

Rewriting: More Formally

substitution = mapping of variables to terms

- $l = r$ is *applicable* to term $t[s]$ if there is a substitution σ 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:

$$[\![P_1; \ldots; P_n]\!] \Longrightarrow l = r$$

is *applicable* to term $t[s]$ with substitution σ 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: $\forall \mathtt{x}.\ \mathtt{x} = \mathtt{x}$
- free: $\mathtt{x} = \mathtt{x}$
- *schematic*: $?\mathtt{x} = ?\mathtt{x}$

  ("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 variabes 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

```
rev(a @ [ ]) = rev a
```

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

Basic Simplification

Goal: `1.` $[\![\mathtt{P_1}; \ldots; \mathtt{P_m}]\!] \Longrightarrow \mathtt{C}$

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** . . . **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

Termination

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.

$$[\![P_1, \ldots P_n]\!] \Longrightarrow l = r$$

is only suitable as a `simp`-rule only if l is "bigger" than r and each P_i.

$$\begin{array}{rl} (\mathtt{n} < \mathtt{m}) = (\mathtt{Suc\,n} < \mathtt{Suc\,m}) & \textbf{NO} \\ (\mathtt{n} < \mathtt{m}) \Longrightarrow (\mathtt{n} < \mathtt{Suc\,m}) = \mathtt{True} & \text{YES} \\ \mathtt{Suc\,n} < \mathtt{m} \Longrightarrow (\mathtt{n} < \mathtt{m}) = \mathtt{True} & \textbf{NO} \end{array}$$

Assumptions and Simplification

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

- Simplify A_1 to A'_1
- Simplify $[\![A_2, \ldots, A_n]\!] \Longrightarrow B$ 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` attirbute.

They must be used explicitly:

`apply (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: $\texttt{b} + \texttt{a} \leadsto \texttt{a} + \texttt{b}$ but not $\texttt{a} + \texttt{b} \leadsto \texttt{b} + \texttt{a}$.

For types nat, int, etc., commutative, associative and distributive laws built in.

Example: apply simp yields:

$$((B + A) + ((2 :: nat) * C)) + (A + B) \leadsto \\ \ldots \leadsto 2 * A + (2 * B + 2 * C)$$

Preprocessing

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

$$\begin{array}{rcl} \neq A & \mapsto & A = \texttt{False} \\ A \longrightarrow B & \mapsto & A \Longrightarrow B \\ A \wedge B & \mapsto & A, B \\ \forall x. A(x) & \mapsto & A(?x) \\ A & \mapsto & A = \texttt{True} \end{array}$$

Example:

$$(\mathtt{p} \longrightarrow \mathtt{q} \wedge \neg \mathtt{r}) \wedge \mathtt{s} \mapsto \mathtt{p} \Longrightarrow \mathtt{q} = \mathtt{True}, \mathtt{r} = \mathtt{True}, \mathtt{s} = \mathtt{True}$$

Demo: Simplification through Rewriting