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 6, 2015
Non-recursive definitions with `definition` No problem

Primitive-recursive (over datatypes) with `primrec` Termination proved automatically internally. Definition syntactically restricted to only allow recursive subcalls on immediate recursive subcomponents.

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 \texttt{function}
  User must (help to) prove termination
  \(\leadsto\) 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.
datatype 'a list = Nil | Cons 'a "'a list"

primrec app :: "'a list ⇒ 'a list ⇒ 'a list
where
  "app Nil ys = ys" | 
  "app (Cons x xs) ys = Cons x (app xs ys)"
datatype: The General Case

$\text{datatype } (\alpha_1, \ldots, \alpha_m) \tau = \begin{array}{c} C_1 \tau_{1,1} \ldots \tau_{1,n_1} \\
\vdots \\
C_k \tau_{k,1} \ldots \tau_{k,n_k} \end{array}$

- 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: $(C_i x_1 \ldots x_{n_i} = C_i y_1 \ldots y_{n_i}) = (x_1 = y_1 \land \ldots \land x_{n_i} = y_{n_i})$

Distinctness and Injectivity are applied by \texttt{simp}

Induction must be applied explicitly
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:

\[
\begin{align*}
  f \ x_1 \ldots (C_1 \ y_{1,1} \ldots y_{1,n_1}) \ldots x_m &= r_1 \\
  &\quad \vdots \\
  f \ x_1 \ldots (C_k \ y_{k,1} \ldots y_{k,n_k}) \ldots x_m &= r_k
\end{align*}
\]

The recursive calls in $r_i$ must be *structurally smaller*, i.e. of the form $f \ a_1 \ldots y_{i,j} \ldots a_m$ where $y_{i,j}$ is a recursive subcomponent of $(C_i \ y_{i,1} \ldots y_{i,n_i})$. 
nat is a datatype

datatype nat = 0 | Suc nat

Functions on nat are definable by primrec!

primrec f :: nat \Rightarrow ... 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
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)
Recursive Function Definition with \texttt{fun}

- Recursive definitions more generally may be defined by \texttt{fun}
- Same basic syntax as \texttt{primrec}
- May nest patterns arbitrarily
- There must exist an "obvious" measure where all recursive calls are done on (structurally) smaller values
- \texttt{fun} finds measure automatically using mostly structural size, lexicographic orderings
fun fib :: nat \Rightarrow nat where
  fib 0 = Suc 0 |
  fib (Suc 0) = Suc 0 |
  fib (Suc (Suc n)) = fib n + fib (Suc n)
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:

\[ 0 + n = n \]  \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:

\[
\begin{array}{c}
\begin{array}{c}
[ P_1; \ldots; P_n ] \\
\Rightarrow \\
l = r
\end{array}
\end{array}
\]

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)
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
From \( x \) to \(?x\)

State lemmas with free variables:

```plaintext
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

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

using \( app\_Nil2 \) with \( \sigma = \{ ?xs \mapsto a \} \)
Basic Simplification

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

proof (simp add: eq_thm\_1 \ldots eq_thm\_n)

Simplify (mostly rewrite) \(P_1; \ldots; P_m\) and \(C\) using
- lemmas with attribute \texttt{simp}
- rules from \texttt{primrec}, \texttt{fun} and \texttt{datatype}
- additional lemmas \(eq\_thm\_1 \ldots eq\_thm\_n\)
- assumptions \(P_1; \ldots; P_m\)

Variations:
- \((simp \ldots del: \ldots)\) removes \texttt{simp}-lemmas
- \texttt{add} and \texttt{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 \( P_i \).

\[(n < m) = (\text{Suc } n < \text{Suc } m) \quad \text{NO}
\]
\[(n < m) \Longrightarrow (n < \text{Suc } m) = \text{True} \quad \text{YES}
\]
\[\text{Suc } n < m \Longrightarrow (n < m) = \text{True} \quad \text{NO}\]
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:

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 \texttt{simp} attribute.

They must be used explicitly:

\begin{verbatim}
proof (simp add: f_def ...)
\end{verbatim}
Ordered Rewriting

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

\[
((B + A) + ((2 :: \text{nat}) \times C)) + (A + B) \sim \\
... \sim 2 \times A + (2 \times B + 2 \times C)
\]
simp-rules are preprocessed (recursively) for maximal simplification power:

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

Example:

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