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

Function Definition in Isabelle/HOL

- 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 `function`
  User must (help to) prove termination
  ($\rightsquigarrow$ 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 Example

```
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

$$\texttt{datatype}\ (\alpha_1, \dots, \alpha_m)\tau \ = \ C_1\ \tau_{1,1} \dots \tau_{1,n_1} \ \mid \ \dots \ \mid \ C_k\ \tau_{k,1} \dots \tau_{k,n_k}$$

- Term Constructors:
  $C_i\ ::\ \tau_{i,1} \Rightarrow \dots \Rightarrow \tau_{i,n_i} \Rightarrow (\alpha_1, \dots, \alpha_m)\tau$
- Distinctness: $C_i\ x_i \dots x_{i,n_i} \neq C_j\ y_j \dots y_{j,n_j}$ if $i \neq j$
- Injectivity: $(C_i\ x_1 \dots x_{n_i} = C_i\ y_1 \dots y_{n_i}) =$
  $(x_1 = y_1 \wedge \dots \wedge x_{n_i} = y_{n_i})$

Distinctness and Injectivity are applied by `simp`
Induction must be applied explicitly

`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:

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

The recursive calls in r_i must be *structurally smaller*, i.e. of the form $f\ a_1 \ldots y_{i,j} \ldots a_m$ where $y_{i,j}$ is a recursive subcomonent 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⇒ ... where
 f 0 = ... |
 f (Suc n) = ...f n ...
```

Type option

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

Important application:

$$
\begin{array}{rcl}
\ldots \Rightarrow \texttt{'a option} & \approx & \text{partial function:} \\
\texttt{None} & \approx & \text{no result} \\
\texttt{Some x} & \approx & \text{result of } \texttt{x}
\end{array}
$$

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)
```

Recursive Function Definition with `fun`

- Recursive definitions more generally may be defined by `fun`
- Same basic syntax as `primrec`
- May nest patterns arbitrarily
- There must exist an "obvious" measure where all recursive calls are done on (structurally) smaller values
- `fun` finds measure automatically using mpstly structural size, lexicagraphic orderings

fun Example

```
fun fib :: nat ⇒ 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:

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

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 $\sigma = \{\texttt{?xs} \mapsto \texttt{a}\}$

Basic Simplification

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

`proof (simp add:` eq_thm_1 ... eq_thm_n`)`

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

- lemmas with attribute `simp`
- rules from `primrec`, `fun` and `datatype`
- additional lemmas eq_thm_1 ... 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.

$$\llbracket P_1, \ldots P_n \rrbracket \Longrightarrow l = r$$

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

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

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`:

`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

Rewriting with Definitions (definition)

Definitions do not have the simp attirbute.

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: $\mathtt{b} + \mathtt{a} \leadsto \mathtt{a} + \mathtt{b}$ but not $\mathtt{a} + \mathtt{b} \leadsto \mathtt{b} + \mathtt{a}$.

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

Example: `proof 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:

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

Example:

$$
(\texttt{p} \longrightarrow \texttt{q} \wedge \neg\texttt{r}) \wedge \texttt{s} \quad \mapsto \quad
\begin{array}{l}
p \Longrightarrow q = \mathit{True}, \\
p \Longrightarrow r = \texttt{False}, \\
s = \mathit{True}
\end{array}
$$

Demo: Simplification through Rewriting