

Topics in Automated Deduction (CS 576)

Elsa L. Gunter

2112 Siebel Center

egunter@illinois.edu

<http://www.cs.illinois.edu/class/sp10/cs576/>

Definitions by Example

Definition:

```
defintion lot_size::"nat * nat" where  
"lot_size  $\equiv$  (62, 103)"
```

```
defintion sq::"nat  $\Rightarrow$  nat" where  
sq_def: "sq n  $\equiv$  n * n"
```

The ASCII for \equiv is ==.

Definitions of form $f\ x_1 \dots x_n \equiv t$ where t only uses $x_1 \dots x_n$ and previously defined constants.

Creates theorem with default name f_def

Definition Restrictions

```
definition prime :: "nat  $\Rightarrow$  bool" where  
"prime p  $\equiv$  1 < p  $\wedge$  (m dvd p  $\longrightarrow$  m = 1  $\vee$  m = p)"
```

Not a definition: m free, but not on left

! Every free variable on rhs must occur as argument
on lhs !

```
"prime p  $\equiv$  1 < p  $\wedge$  ( $\forall$  m. m dvd p  $\longrightarrow$  m = 1  $\vee$  m = p)"
```

Note: no recursive definitions with `definition`

Using Definitions

Definitions are not used automatically

Unfolding of definition of `sq`:

```
apply (unfold sq_def)
```

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 $f\ x \neq y$

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 `definition`
No problem
- Primitive-recursive (over datatypes) with `primrec`
Termination proved automatically internally
- 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
tedious method, but can work when other methods
don't.

primrec Example

```
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 τ is a **datatype** with constructors C_1, \dots, C_k , then $f :: \dots \Rightarrow \tau \Rightarrow \tau'$ can be defined by *primitive recursion* by:

$$\begin{aligned} f \ x_1 \dots (C_1 \ y_{1,1} \dots y_{1,n_1}) \dots x_m &= r_1 \quad | \\ &\dots \\ f \ x_1 \dots (C_k \ y_{k,1} \dots y_{k,n_k}) \dots x_m &= r_k \end{aligned}$$

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

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:

... \Rightarrow 'a option \approx partial function:
None \approx no result
Some x \approx result of x

option Example

`primrec lookup :: 'k \Rightarrow ('k \times 'v)list \Rightarrow '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:

$$0 + \text{Suc } 0 \leq \text{Suc } 0 + x \quad \underline{\underline{(1)}}$$
$$\text{Suc } 0 \leq \text{Suc } 0 + x \quad \underline{\underline{(2)}}$$
$$\text{Suc } 0 \leq \text{Suc}(0 + x) \quad \underline{\underline{(4)}}$$
$$0 \leq 0 + x \quad \underline{\underline{(3)}}$$
$$\text{True}$$

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:

$$\llbracket P_1; \dots; P_n \rrbracket \Longrightarrow l = r$$

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

- $\sigma(l) = s$ and
- $\sigma(P_1), \dots, \sigma(P_n)$ 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 = bound at meta-level
- Operationally:
 - free variables 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

$$\text{rev}(a @ []) = \text{rev } a$$

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

Basic Simplification

Goal: 1. $\llbracket P_1; \dots; P_m \rrbracket \implies C$

`apply (simp add: eq_thm1 ... eq_thmn)`

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

- lemmas with attribute `simp`
- rules from `primrec` and `datatype`
- additional lemmas `eq_thm1 ... eq_thmn`
- assumptions $P_1; \dots; P_m$

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.

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

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

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

Assumptions and Simplification

Simplification of $\llbracket A_1, \dots, A_n \rrbracket \implies B$:

- Simplify A_1 to A'_1
- Simplify $\llbracket A_2, \dots, A_n \rrbracket \implies B$ 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 (`definition`)

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 `simp`-rules are used only if the term becomes lexicographically smaller.

Example: $b + a \rightsquigarrow a + b$ but not $a + b \rightsquigarrow b + a$.

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

Example: `apply simp` yields:

$$\begin{aligned} & ((B + A) + ((2 :: nat) * C)) + (A + B) \rightsquigarrow \\ & \dots \rightsquigarrow 2 * A + (2 * B + 2 * C) \end{aligned}$$

Preprocessing

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

$$\neq A \mapsto A = \text{False}$$

$$A \longrightarrow B \mapsto A \implies B$$

$$A \wedge B \mapsto A, B$$

$$\forall x.A(x) \mapsto A(?x)$$

$$A \mapsto A = \text{True}$$

Example:

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

Demo: Simplification through Rewriting