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 ≡ (62, 103)"
defintion sq::"nat ⇒ nat" where
sq_def:  "sq n ≡ n * n"
```

The ASCII for ≡ is ==.
Definitions of form $f\ x_1 \ldots x_n \equiv t$ where t only uses $x_1 \ldots x_n$ and previously defined constants.
Creates theorem with default name `f_def`

Definition Restrictions

```
definition prime ::  "nat ⇒ bool" where
"prime p ≡ 1<p ∧ (m dvd p ⟶ m = 1 ∨ m = p)"
```

Not a definition: `m` free, but not on left

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

```
"prime p ≡ 1<p ∧ (∀ m.  m dvd p ⟶ m = 1 ∨ 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` ≠ `y` then ∀`y`. `f x` ≠ `y`. Then **`f x` ≠ `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
  ($\leadsto$ 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 $C_1, \ldots, C_k$, then $f :: \cdots \Rightarrow \tau \Rightarrow \tau'$ can be defined by *primitive recursion* by:

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

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

`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:

`... ⇒ 'a option` $\approx$ partial function:
`None` $\approx$ no result
`Some x` $\approx$ result of `x`

`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)
```

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:

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

Rewriting:

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

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: $\forall$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 variabes are fixed
  - schematic variables are instantiated by substitutions

From `x` to `?x`

State lemmas with free variables:

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

```
apply(induct_tac 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. ⟦P1; ...; Pm⟧ ⟹ C`

`apply (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` 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.

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

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

$$\begin{array}{rl} \texttt{(n < m) = (Sucn < Sucm)} & \textbf{NO} \\ \texttt{(n < m)} \Longrightarrow \texttt{(n < Sucm) = True} & \text{YES} \\ \texttt{Sucn < m} \Longrightarrow \texttt{(n < m) = True} & \textbf{NO} \end{array}$$

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 (`definition`)

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: $\mathtt{b} + \mathtt{a} \leadsto \mathtt{a} + \mathtt{b}$ but not $\mathtt{a} + \mathtt{b} \leadsto \mathtt{b} + \mathtt{a}$.
For types `nat`, `int`, etc., commutative, associative and distributive laws built in.
Example: `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:

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

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