Topics in Automated Deduction (CS 576)

Elsa L. Gunter

2112 Siebel Center

eguntercs.uiuc.edu

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

Structural Induction on Lists

P xs holds for all lists xs if

- P Nil, and
- for arbitrary a and list, P list implies

  P (Cons a list)

$$\frac{\texttt{P Nil} \qquad \begin{array}{c}\texttt{P ys}\\ \vdots \\ \texttt{P (Cons y ys)}\end{array}}{\texttt{P xs}}$$

In Isabelle:

```
[| ?P []; !!a list.  ?P list ==> ?P (a # list) |] ==> ?P ?list
```

Proof Method

- Structural Induction
  - **Syntax:** `(induct x)`

    `x` must be a free variable in the first subgoal

    The type of `x` must be a datatype

  - **Effect:** Generates 1 new subgoal per constructor
  - Type of `x` determines which induction principle to use

A Recursive Function: List Append

Declaration:

```
consts app :: "'a list ⇒ 'a list ⇒ 'a list
```

and definition by *primitive recursion*:

```
primrec
app Nil ys = ____
app (Cons x xs) ys = ____app xs ...____
```

One rule per constructor

Recursive calls only applied to constructor arguments

Guarantees termination (total function)

Demo: Append and Reverse

Introducing New Types

Keywords:

- `typedef`: Primitive for type definitions; Only real way of introducing a new type with new properties More on this later
- `typedecl`: Pure declaration; New type with no properties (expect that it is non-empty)

Introducing New Types

Keywords:

- `types`: Abbreviation - may be used in constant declarions
- `datatype`: Defines recursive data-types; solutions to free algebra specificaitons

  Basis for primitive recursive function definitions

`typedecl`

`typedecl` *name*

Introduces new "opaque" *name* without definition

Serves similar role for generic reasoning as polymorphism, but can't be specialized

Example:

`typedecl` **`addr`** — An abstract type of addresses

types

types $\langle$*tyvars*$\rangle$ *name* $= \tau$

Introduces an abbreviation $\langle$*tyvars*$\rangle$ *name* for type τ

Examples:

```
types
name = string
('a,'b)foo = "'a list * 'b"
```

Type abbreviations are expanded immediately after parsing

Not present in internal representation and Isabelle output

datatype: The Example

```
datatype 'a list = Nil | Cons 'a "'a list"
```

Properties:

- Type constructors: `Nil :: 'a list`
  `Cons :: 'a ⇒ 'a list ⇒ 'a list`
- Distinctness: `Nil ≠ Cons x xs`
- Injectivity:

  `(Cons x xs = Cons y ys) = (x = y ∧ xs = ys)`

`datatype`: The General Case

$$
\begin{array}{lcl}
\texttt{datatype}\ (\alpha_1, \ldots, \alpha_m)\tau & = & C_1\ \tau_{1,1} \ldots \tau_{1,n_1} \\
& | & \ldots \\
& | & C_k\ \tau_{k,1} \ldots \tau_{k,n_k}
\end{array}
$$

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

Distinctness and Injectivity are applied automatically

Induction must be applied explicitly

Definitions by Example

Declaration: **consts**

```
lot_size :: "nat * nat"
sq :  "nat ⇒ nat"
```

Definition: **defs**

```
"lot_size ≡ (62, 103)"
sq_def:  "sq n ≡ n * n"
```

Declarations
+ definitions: **constdefs**

```
lot_size :: "nat * nat"
lot_size_def:  "lot_size ≡ (62, 103)"
sq :  "nat ⇒ nat"
sq_def:  "sq n ≡ n * n"
```

Definition Restrictions

constdefs

`prime :: "nat ⇒ bool"`

`"prime p ≡ p<1 ∧ (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 ≡ p<1 ∧ (∀ m. m dvd p ⟶ m = 1 ∨ m = p)"`

Note: no recursive definitions with `defs` or `constdefs`

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 $\not{f}$ `x = y` then $\forall$`y. f x = y`. Then **$\not{f}$ x = 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:

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

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 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

```
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 [ ] ⇒... | y#ys ⇒ ...y ...ys ...)
```

In general: one case per constructor

Same number of cases as 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:

$$t = C_i\ x_1 \dots x_{n_i} \Longrightarrow \dots$$

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:

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

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: $\forall \mathtt{x}.\ \mathtt{x} = \mathtt{x}$
- 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. $[\![P_1; \ldots; P_m]\!] \Longrightarrow C$

`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