CS576 Topics in Automated Deduction

Elsa L Gunter
2112 SC, UIUC
egunter@illinois.edu
http://courses.grainger.illinois.edu/cs576

Slides based in part on slides by Tobias Nipkow

February 3, 2026

datatype: An Example

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

Properties:

- Type constructors: `list` of one argument
- Term constructors: `Nil  :: 'a list`
  `Cons :: 'a ⇒ 'a list ⇒ 'a list`
- Distinctness: `Nil` $\neq$ `Cons x xs`
- Injectivity:
  `(Cons x xs = Cons y ys) = (x = y ∧ xs = ys)`

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)`

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

In Isabelle:

$$[\![\ ?P[\,];\ \bigwedge a\ list.\ ?P\ list \Longrightarrow ?P(a\,\#\,list)\ ]\!] \Longrightarrow\ ?P\ ?list$$

`datatype`: The General Case

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

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

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

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

case

Every `datatype` introduces a `case` construct, e.g.

```
(case xs of [ ] ⇒...| y#ys ⇒ ...y ...ys ...)
```

In general:

`case` *Arbitrarily nested pattern* ⇒ *Expression using pattern variables*
| *Another pattern* ⇒ *Another expression*
| . . .

Patterns may be non-exhaustive, or overlapping
Order of clauses matters - early clause takes precedence.

Case Distinctions

$$\texttt{apply} \;/\; \texttt{proof}\ (\texttt{case\_tac}\ t)$$

creates k subgoals:

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

one for each constructor C_i

Demo: Another Datatype Example

Definitions by Example

Definition:

```
definition lot_size::"nat * nat" where
 "lot_size ≡ (62, 103)"
```

```
definition sq::"nat ⇒ nat" where
 sq_def:  "sq n ≡ n * n"
```

The ASCII for $\equiv$ 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`:

```
proof (unfold sq_def)
```

Rewriting definition of `sq` out of current goal:

```
proof (simp add:  sq_def)
```

HOL Functions are Total

Why nontermination can be harmful:

If `f x` is undefined, is $\texttt{f x} = \texttt{f x}$?

Excluded Middle says it must be `True` or `False`

Reflexivity says it's `True`

How about $\texttt{f x} = 0$? $\texttt{f x} = 1$? $\texttt{f x} = \texttt{y}$?

If $\texttt{f x} \neq \texttt{y}$ then $\forall \texttt{y}.\ \texttt{f x} \neq \texttt{y}$. Then $\texttt{fx} \neq \texttt{fx}$ #

! 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
  use of choose operator, and other tedious approaches, but can work when built-in 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:

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

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:

$\ldots \Rightarrow$ `'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:

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

Rewriting:

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

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

Basic Simplification

Goal: 1. $\llbracket P_1; \ldots ; P_m \rrbracket \Longrightarrow C$

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` 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

Basic Simplification

Goal: 1. $[\![\mathtt{P}_1;\ldots;\mathtt{P}_\mathtt{m}]\!] \Longrightarrow \mathtt{C}$

`proof (simp add:` $eq\_thm_1 \; \ldots \; eq\_thm_n$`)`

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

- lemmas with attribute `simp`
- rules from `primrec` and `datatype`
- additional lemmas $eq\_thm_1 \; \ldots \; eq\_thm_n$
- 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.

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

Assumptions and Simplification

Simplification of $\llbracket A_1, \ldots, A_n \rrbracket \Longrightarrow B$:

- Simplify A_1 to A_1'
- Simplify $\llbracket A_2, \ldots, A_n \rrbracket \Longrightarrow B$ 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: `b` $+$ `a` $\leadsto$ `a` $+$ `b` but not `a` $+$ `b` $\leadsto$ `b` $+$ `a`.

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

Example: `proof simp` yields:

$$\begin{array}{l} ((B + A) + ((2 :: nat) * C)) + (A + B) \leadsto \\ \ldots \leadsto 2 * A + (2 * B + 2 * C) \end{array}$$

Preprocessing

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

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

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