Topics in Automated Deduction (CS 576)

Elsa L. Gunter

2112 Siebel Center

egunter@illinois.edu

http://www.cs.illinois.edu/class/sp12/cs576/

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. $[\![\mathrm{P}_1; \ldots; \mathrm{P}_\mathrm{m}]\!] \Longrightarrow \mathrm{C}$

`apply (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.

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

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

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

Assumptions and Simplification

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

- Simplify A_1 to A_1'
- Simplify $\llbracket A_2, \dots, A_n \rrbracket \Longrightarrow B$ using A_1'

Ignoring Assumptions

Sometimes need to ignore assumptions; can introduce non-termination.

How to exclude assumptions from simp:

```
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:

$$((B + A) + ((2 :: nat) * C)) + (A + B) \leadsto$$
$$\dots \leadsto 2 * A + (2 * B + 2 * C)$$

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

Proof Basics

- Isabelle uses *Natural Deduction* proofs
  - Uses *sequent* encoding
- Rule notation:

Rule	Sequent Encoding
$\dfrac{\mathtt{A_1 \ldots A_n}}{\mathtt{A}}$	$[\![\mathtt{A_1, \ldots, A_n}]\!] \Longrightarrow \mathtt{A}$
$\dfrac{\mathtt{A_1 \ldots} \overset{\mathtt{B}}{\underset{\mathtt{A_i}}{\vdots}} \mathtt{\ldots A_n}}{\mathtt{A}}$	$[\![\mathtt{A_1, \ldots, B \Longrightarrow A_i, \ldots, A_n}]\!] \Longrightarrow \mathtt{A}$

Natural Deduction

For each logical operator $\oplus$, have two kinds of rules:

Introduction: How can I prove $A \oplus B$?

$$\frac{?}{A \oplus B}$$

Elimination: What can I prove using $A \oplus B$?

$$\frac{\ldots A \oplus B \ldots}{?}$$

Operational Reading

$$\frac{A_1 \ldots A_n}{A}$$

Introduction rule:

To prove A it suffices to prove $A_1 \ldots A_n$.

Elimination rule:

If we know A_1 and we want to prove A

it suffices to prove $A_2 \ldots A_n$

More Rules

$$\frac{\mathtt{A \wedge B}}{\mathtt{A}}\ \texttt{conjunct1} \qquad \frac{\mathtt{A \wedge B}}{\mathtt{B}}\ \texttt{conjunct2}$$

$$\frac{\mathtt{A \longrightarrow B} \quad \mathtt{A}}{\mathtt{B}}\ \texttt{mp}$$

Compare to elimination rules:

$$\frac{\mathtt{A \wedge B} \quad [\![\mathtt{A; B}]\!] \Longrightarrow \mathtt{C}}{\mathtt{C}}\ \texttt{conjE} \qquad \frac{\mathtt{A \longrightarrow B} \quad \mathtt{A} \quad \mathtt{B} \Longrightarrow \mathtt{C}}{\mathtt{C}}\ \texttt{impE}$$

"Classical" Rules

$$\frac{\neg A \Longrightarrow \mathtt{False}}{A}\ \mathtt{ccontr} \qquad \frac{\neg A \Longrightarrow A}{A}\ \mathtt{classical}$$

- `ccontr` and `classical` are not derivable from the Natural Deduction rules.
- They make the logic *"classical"*, i.e. *"non-constructive* or *"non-intuitionistic"*.

Proof by Assumption

$$\frac{A_1 \dots A_i \dots A_n}{A_i}$$

Rule Application: The Rough Idea

Applying rule $[\![A_1; \dots; A_n]\!] \Longrightarrow A$ to subgoal C:

- Unify A and C
- Replace C with n new subgoals: $A'_1 \ \dots \ A'_n$

Backwards reduction, like in Prolog

Example: rule: $[\![?P; ?Q]\!] \Longrightarrow ?P \wedge ?Q$
subgoal: 1. $A \wedge B$

Result: 1. A
2. B

Rule Application: More Complete Idea

Applying rule $[\![A_1; \dots; A_n]\!] \Longrightarrow A$ to subgoal C:

- Unify A and C with (meta)-substitution σ
- Specialize goal to $\sigma(C)$
- Replace C with n new subgoals: $\sigma(A_1) \ \dots \ \sigma(A_n)$

Note: schematic variables in C treated as existential variables

Does there exist value for $?X$ in C that makes C true?

(Still not the whole story)

`rule` Application

Rule:	$[\![A_1; \dots; A_n]\!] \Longrightarrow A$
Subgoal:	1. $[\![B_1; \dots; B_m]\!] \Longrightarrow C$
Substitution:	$\sigma(A) \equiv \sigma(C)$
New subgoals:	1. $[\![\sigma(B_1); \dots; \sigma(B_m)]\!] \Longrightarrow \sigma(A_1)$ $\vdots$ n. $[\![\sigma(B_1); \dots; \sigma(B_m)]\!] \Longrightarrow \sigma(A_n)$
Proves:	$[\![\sigma(B_1); \dots; \sigma(B_m)]\!] \Longrightarrow \sigma(C)$
Command:	`apply (rule` *<rulename>*`)`

Proof by `assumption`

`apply assumption`

proves:

1. $[\![B_1; \dots; B_m]\!] \Longrightarrow C$

by unifying C with one of the B_i

Demo: Application of Introduction Rule

Applying Elimination Rules

`apply (erule` *<elim-rule>*`)`

Like `rule` but also

- unifies first premise of rule with an assumption
- eliminates that assumption instead of conclusion

Example

Rule:	$[\![?P\wedge?Q; [\![?P; ?Q]\!] \Longrightarrow ?R]\!] \Longrightarrow ?R$
Subgoal:	1. $[\![X; A \wedge B; Y]\!] \Longrightarrow Z$
Unification:	$?P\wedge?Q \equiv A \wedge B$ and $?R \equiv Z$
New subgoal:	1. $[\![X; Y]\!] \Longrightarrow [\![A; B]\!] \Longrightarrow Z$
Same as:	1.$[\![X; Y; A; B]\!] \Longrightarrow Z$

How to Prove in Natural Deduction

- *Intro* rules decompose formulae to the *right* of $\Longrightarrow$
  `apply (rule` *<intro-rule>*`)`
- *Elim* rules decompose formulae to the *left* of $\Longrightarrow$
  `apply (erule` *<elim-rule>*`)`

Demo: Examples

Safe and Unsafe Rules

Safe rules preserve provability:
`conjI`, `impI`, `notI`, `iffI`, `refl`, `ccontr`, `classical`, `conje`, `disjE`

Unsafe rules can reduce a provable goal to one that is not:
`disjI1`, `disjI2`, `impE`, `iffD1`, `iffD2`, `notE`

Try safe rules before unsafe ones

$\Longrightarrow$ vs $\longrightarrow$

- Theorems usually more useful written as
  $[\![A_1; \ldots ; A_n]\!] \Longrightarrow A$
  instead of $A_1 \wedge \ldots \wedge A_n \longrightarrow A$ (easier to apply)
- **Exception:** (in `apply`-style): induction variable must not occur in premises
- Example: For induction on x, transform:
  $[\![A; B(x)]\!] \Longrightarrow C(x) \leadsto A \Longrightarrow B(x) \longrightarrow C(x)$

  Reverse transformation (after proof):
  `lemma abc [rule_format]:` $A \Longrightarrow B(x) \longrightarrow C(x)$

Demo: Further Techniques

Parameters

Subgoal:

1. $\bigwedge x_1 \ldots x_n.\ Formula$

The x_i are called *parameters* of the subgoal
Intuition: local constants, i.e. arbitrary fixed values

Rules are automatically lifted passed $\bigwedge x_1 \ldots x_n$ and applied directly to $Formula$

Scope

- Scope of parameters: whole subgoal
- Scope of $\forall$, $\exists$, $\ldots$: ends with ; or $\Longrightarrow$, or enclosing)

$$\bigwedge xy.\ [\![\forall y.\ P\ y\ \longrightarrow Q\ z\ y;\ Q\ x\ y]\!] \Longrightarrow \exists x.\ Q\ x\ y$$

means

$$\bigwedge xy.\ [\![(\forall y_1.\ P\ y_1\ \longrightarrow Q\ z\ y_1);\ Q\ x\ y]\!] \Longrightarrow \exists x_1.\ Q\ x_1\ y$$

α-Conversion and Scope of Variables

- $\forall$x. P x: x can appear in P x.
  Example: $\forall$x. x = x is obtained by P $\mapsto$ λu. u = u
- $\forall$x. P: x cannot appear in P
  Example: P $\mapsto$ x = x yields $\forall$x$'$. x = x

Bound variables are renamed automatically to avoid name clashes with other variables.

Natural Deduction Rules for Quantifiers

$$\frac{\bigwedge x.\ P\ x}{\forall x. P\ x}\ \mathtt{allI} \qquad \frac{\forall x.\ P x \qquad P\ ?x \Longrightarrow R}{R}\ \mathtt{allE}$$

$$\frac{P ?x}{\exists x.\ P\ x}\ \mathtt{exI} \qquad \frac{\exists x.\ P x \qquad \bigwedge x.\ P\ x \Longrightarrow R}{R}\ \mathtt{exE}$$

- `allI` and `exE` introduce new parameters ($\bigwedge x$)
- `allE` and `exI` introduce new unknowns ($?x$)

Safe and Unsafe Rules

Safe: `allI`, `exE`

Unsafe: `allE`, `exI`

Create parameters first, unknowns later

Instantiating Variables in Rules

`apply (rule_tac` x `= "`*term*`" in` *rule*`)`

Like `rule`, but $?x$ in *rule* is instantiated with *term* before application.

$?x$ must be schematic variable occurring in statement of *rule*.

Similar: `erule_tac`

! x is in rule, not in goal !

Two Successful Proofs

1\. $\forall x.\ \exists y.\ x = y$
`apply (rule allI)`
1\. $\bigwedge x. \exists y.\ x = y$

Better practice:
`apply(rule_tac x = "x" in exI)`
1\. $\bigwedge x.\ x = x$
`apply (rule refl)`

simpler & cleaner

Exploration:
`apply (rule exI)`
1\. $\bigwedge x.\ x = ?y x$
`apply (rule refl)`
$?y \mapsto \lambda u.\ u$

shorter & trickier