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 17, 2026

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

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

Example:

$$(\texttt{p} \longrightarrow \texttt{q} \wedge \neg \texttt{r}) \wedge \texttt{s} \quad \mapsto \quad \begin{array}{l} p \Longrightarrow q = \mathit{True}, \\ p \Longrightarrow r = \texttt{False}, \\ s = \mathit{True} \end{array}$$

Demo: Simplification through Rewriting

General Isar Proof Format

```
proof (method)
    fix x
    assume A0: formula_0
    from A0
    have A1: formula_1
    by (method)
    from A0 and A1
      ...
      show formula_n
      proof (method)
        ...
      qed
qed
```

Proves $formula_0 \Longrightarrow formula_n$

Basic Isar Syntax

proof = `proof` [*method*] *statement** `qed`
| `by` *method*

method = `(simp ...)`|`(auto ...)`|`(blast ...)`|`(rule ...)`| ...

statement = `fix` *variable*$^+$ ($\bigwedge$)
| `assume` *proposition* ($\Longrightarrow$)
| [`from` *name*$^+$] *objective proof*
| `next` (starts next subgoal)

objective = `show` *proposition* (next proof step)
| `have` *proposition* (local claim)
| `obtain` *variable*$^+$ `where` *proposition*$^+$

proposition = [*name*:] *formula*

Proof Basics

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

Rule	Sequent Encoding
$\dfrac{A_1 \ldots A_n}{A}$	$[\![A_1, \ldots, A_n]\!] \Longrightarrow A$
$\dfrac{\begin{array}{c}B \\ \vdots \\ A_1 \ldots\ A_i\ \ldots A_n\end{array}}{A}$	$[\![A_1, \ldots, B \Longrightarrow A_i, \ldots, A_n]\!] \Longrightarrow A$

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 $A_2 \ldots A_n$

Natural Deduction for Propositional Logic

$$\frac{A \quad B}{A \wedge B}\ \mathtt{conjI}$$

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

$$\frac{A}{A \vee B} \qquad \frac{B}{A \vee B}\ \mathtt{disjI1/2}$$

$$\frac{A \vee B \quad A \Longrightarrow C \quad B \Longrightarrow C}{c}\ \mathtt{disjE}$$

$$\frac{A \Longrightarrow B}{A \longrightarrow B}\ \mathtt{impI}$$

$$\frac{A \longrightarrow B \quad A \quad B \Longrightarrow C}{C}\ \mathtt{impE}$$

Natural Deduction for Propositional Logic

$$\frac{A \Longrightarrow B \quad B \Longrightarrow A}{A = B}\ \texttt{iffI} \qquad \frac{A = B \quad A}{B}\ \texttt{iffD1}$$

$$\frac{A = B \quad B}{A}\ \texttt{iffD2}$$

$$\frac{A \Longrightarrow \texttt{False}}{\neg A}\ \texttt{notI} \qquad \frac{\neg A \quad A}{B}\ \texttt{notE}$$

Equality

$$\frac{}{t = t}\,\texttt{refl} \qquad \frac{s = t}{t = s}\,\texttt{sym} \qquad \frac{r = s \quad s = t}{r = t}\,\texttt{trans}$$

$$\frac{s = t \quad A(s)}{A(t)}\,\texttt{subst}$$

`subst` rarely needed explicitly – used implicitly by `simp`

More Rules

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

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

Compare to elimination rules:

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

"Classical" Rules

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

- `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

In classical Natural Deduction,

$$\frac{\mathtt{A_1} \ldots \mathtt{A_i} \ldots \mathtt{A_n}}{\mathtt{A_i}}$$

If we know a bunch of things, including A_i, then we know A_i

In Isabelle

$$[\![\mathtt{A}]\!] \Longrightarrow \mathtt{A}$$

Rule Application: The Rough Idea

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

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

Backwards reduction, like in Prolog

Example: rule: $[\![?\mathtt{P}; ?\mathtt{Q}]\!] \Longrightarrow ?\mathtt{P} \wedge ?\mathtt{Q}$

subgoal: `1. A` $\wedge$ `B`

Result:
```
1.  A
2.  B
```

Rule Application: More Complete Idea

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

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

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: $[\![A_1; \ldots; A_n]\!] \Longrightarrow A$

Subgoal: 1. $[\![B_1; \ldots; B_m]\!] \Longrightarrow C$

Substitution: $\sigma(A) \equiv \sigma(C)$

New subgoals:
1. $[\![\sigma(B_1); \ldots; \sigma(B_m)]\!] \Longrightarrow \sigma(A_1)$
⋮
n. $[\![\sigma(B_1); \ldots; \sigma(B_m)]\!] \Longrightarrow \sigma(A_n)$

Proves: $[\![\sigma(B_1); \ldots; \sigma(B_m)]\!] \Longrightarrow \sigma(C)$

Command: `apply (rule` <*rulename*>`)`

Proof by assumption

```
apply assumption
```

proves:

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

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

- `proof (rule ...)` generally does the work of `erule` in Isar.

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>)
proof (rule <intro-rule>)
```

- *Elim* rules decompose formulae to the *left* of $\Longrightarrow$

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

Demo: Examples

Safe and Unsafe Rules - Basic Logical Connectives

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:
  $[\![A; B(x)]\!] \Longrightarrow C(x) \rightsquigarrow A \Longrightarrow B(x) \longrightarrow C(x)$

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

Demo: Further Techniques

Parameters

Subgoal:

1. $\Lambda x_1 \ldots x_n.$ *Formula*

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

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

Scope

- Scope of parameters: whole subgoal
- Scope of $\forall$, $\exists$, ...: 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

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

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

Natural Deduction Rules for Quantifiers

$$\frac{\Lambda x.\ P\ x}{\forall x.P\ x}\ \texttt{allI} \qquad \frac{\forall x.\ Px \quad P\ ?x \Longrightarrow R}{R}\ \texttt{allE}$$

$$\frac{P\ ?x}{\exists x.\ P\ x}\ \texttt{exI} \qquad \frac{\exists x.\ Px \quad \Lambda x.\ P\ x \Longrightarrow R}{R}\ \texttt{exE}$$

- allI and exE introduce new parameters (Λx)
- allE and exI introduce new unknowns ($?x$)

Safe and Unsafe Rules - Quantifiers

Safe: `allI`, `exE`

Unsafe: `allE`, `exI`

Create parameters first, unknowns later

Instantiating Variables in Rules

`proof (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 !