Isabelle Tutorial and User's Manual

Lawrence C. Paulson & Tobias Nipkow
Computer Laboratory
University of Cambridge

15 January 1990

Abstract

This manual describes how to use the theorem prover *Isabelle*. For beginners, it explains how to perform simple single-step proofs in the built-in logics. These include first-order logic, a classical sequent calculus, ZF set theory, Constructive Type Theory, and higher-order logic. Each of these logics is described. The manual then explains how to develop advanced tactics and tacticals and how to derive rules. Finally, it describes how to define new logics within Isabelle.

Acknowledgements. Isabelle uses Dave Matthews's Standard ML compiler, Poly/ML. Philippe de Groote wrote the first version of the logic LK. Funding and equipment were provided by SERC/Alvey grant GR/E0355.7 and ESPRIT BRA grant 3245. Thanks also to Philippe Noël, Brian Monahan, Martin Coen, and Annette Schumann.

Contents

You can only find truth with logic
if you have already found truth without it.

G.K. Chesterton, *The Man who was Orthodox*

Chapter 1

Basic Features of Isabelle

Although the theorem prover Isabelle is still far from finished, there are users enough to justify writing a manual. The manual describes pure Isabelle and several object-logics — natural deduction first-order logic (constructive and classical versions), Constructive Type Theory, a classical sequent calculus, ZF set theory, and higher-order logic — with their syntax, rules, and proof procedures. The theory and ideas behind Isabelle are described elsewhere [7, 9, 10].

Fortunately, beginners need not understand exactly how Isabelle works. Nor need they know about all the meta-level rules, proof tactics, and interactive proof commands. This manual starts at an introductory level, and leaves the worst details for the Appendices.

To understand this report you will need some knowledge of the Standard ML language (Wikström [15] is a simple introduction). When studying advanced material, you may want to have Isabelle's sources at hand. Advanced Isabelle theorem proving can involve writing functions in ML.

Isabelle was first distributed in 1986. The 1987 version was the first with a higher-order meta-logic and $\bigwedge$-lifting for quantifiers. The 1988 version added limited polymorphism and $\Longrightarrow$-lifting for natural deduction. The current version includes a pretty printer, an automatic parser generator, and an object-level higher-order logic. Isabelle is still under development and will continue to change.

The present syntax is more concise than before — and *not upwards-compatible* with previous versions! Existing Isabelle users can convert their files using a tool provided on the distribution tape.

The manual is organized around the different ways you can work as you become more experienced.

- Chapter 1 introduces Pure Isabelle, the things common to all logics. These include theories and rules, tactics, and subgoal commands. Several simple proofs are demonstrated.

- Chapter 2 introduces the versions of first-order logic provided by Isabelle. The easiest way to get started with Isabelle is to work in one of these. Each logic has a large collection of examples, with proofs, that you can try. Some automatic tactics are available. As you gain confidence with the standard examples, you can develop your own proofs and tactics.

- Chapter 3 describes Isabelle's more advanced logics, namely set theory, Constructive Type Theory, and higher-order logic. It is possible to plunge into these knowing only about single step proofs, but you might want to skip this chapter on a first reading.

- Chapter 4 describes in detail how tactics work. It also introduces *tacticals* — the building-blocks for tactics — and describes how to use them to define search procedures. The chapter also describes how to make simple extensions to a logic by defining new constants.

- Chapter 5 (written by Tobias Nipkow) describes how to build an object-logic: syntax definitions, the role of signatures, how theories are combined. Defining your own object-logic is a major undertaking. You must have a thorough understanding of the logic to have any chance of successfully representing it in Isabelle.

- The Appendices document the internal workings of Isabelle. This information is for experienced users.

1.1 Overview of Isabelle

Isabelle is a theorem prover that can cope with a large class of logics. You need only specify the logic's syntax and rules. To go beyond proof checking, you can implement search procedures using built-in tools.

1.1.1 The representation of logics

Object-logics are formalized within Isabelle's meta-logic, which is intuitionistic higher-order logic with implication, universal quantifiers, and equality. The implication $\phi \Longrightarrow \psi$ means 'ϕ implies ψ', and expresses logical entailment. The quantification $\bigwedge x.\phi$ means 'ϕ is true for all x', and expresses generality in rules and axiom schemes. The equality $a \equiv b$ means 'a equals b', and allows new symbols to be defined as abbreviations. For instance, Isabelle represents the inference rule

$$\frac{P \quad Q}{P \,\&\, Q}$$

by the following axiom in the meta-logic:

$$\bigwedge P.\ \bigwedge Q.\ P \Longrightarrow (Q \Longrightarrow P \,\&\, Q)$$

The structure of rules generalizes PROLOG's Horn clauses; proof procedures can exploit logic programming techniques.

Isabelle borrows ideas from LCF [8]. Formulae are manipulated through the meta-language Standard ML; proofs can be developed in the backwards direction via tactics and tacticals. The key difference is that LCF represents rules by functions,

not by axioms. In LCF, the above rule is a function that maps the theorems P and Q to the theorem $P \,\&\, Q$.

Higher-order logic uses the typed λ-calculus, including its formalization of quantifiers. So $\forall x.P$ can be represented by $All(\lambda x.P)$, where All is a new constant and P is a formula containing x. Viewed semantically, $\forall x.F(x)$ is represented by $All(F)$, where the variable F denotes a truth-valued function. Isabelle represents the rule

$$\frac{P}{\forall x.P}$$

by the axiom

$$\bigwedge F\,.\,(\bigwedge x.\,F(x)) \Longrightarrow All(F)$$

The introduction rule is subject to the proviso that x is not free in the assumptions. Any use of the axiom involves proving $F(x)$ for arbitrary x, enforcing the proviso [9]. Similar techniques handle existential quantifiers, the Π and Σ operators of Type Theory, the indexed union operator of set theory, and so forth. Isabelle easily handles induction rules and axiom schemes (like set theory's Axiom of Separation) that involve arbitrary formulae.

1.1.2 Theorem proving with Isabelle

Proof trees are derived rules, and are built by joining rules together. This comprises both forwards and backwards proof. Backwards proof works by matching a goal with the conclusion of a rule; the premises become the subgoals. Forwards proof works by matching theorems to the premises of a rule, making a new theorem.

Rules are joined by *higher-order unification*, which involves solving equations in the typed λ-calculus with respect to α, β, and η-conversion. Unifying $f(x)$ with the constant A gives the two unifiers $\{f = \lambda y.A\}$ and $\{f = \lambda y.y,\ x = A\}$. Multiple unifiers indicate ambiguity: the four unifiers of $f(0)$ with $P(0,0)$ reflect the four different ways that $P(0,0)$ can be regarded as depending upon 0.

To demonstrate the implementation of logics, several examples are provided. Many proofs have been performed in these logics. For first-order logic, an automatic procedure can prove many theorems involving quantifiers. Constructive Type Theory examples include the derivation of a choice principle and simple number theory. The set theory examples include properties of union, intersection, and Cartesian products.

1.1.3 Fundamental concepts

Isabelle comprises a tree of object-logics. The branching can be deep as well as broad, for one logic can be based on another. The root of the tree, Pure Isabelle, implements the meta-logic. Pure Isabelle provides the concepts and operations common to all the object-logics: types and terms; syntax and signatures; theorems and theories; tactics and proof commands; a functor for building simplifiers.

The *types* (denoted by Greek letters σ, τ, and υ) include basic types, which correspond to syntactic categories in the object-logic. There are also function types of the form $\sigma \to \tau$. Types have the ML type `typ`.

The *terms* (denoted by a, b, and c) are the usual terms of the typed λ-calculus. They can encode the syntax of object-logics. The encoding of object-formulae into the meta-logic usually has an obvious semantic reading as well. Isabelle implements many operations on terms, of which the most complex is higher-order unification. Terms have the ML type `term`.

An automatic package (written by Tobias Nipkow) performs parsing and display of terms. You can specify the syntax of a logic as a collection of mixfix operators, including directives for Isabelle's pretty printer.

The theorems of the meta-logic have the ML type `thm`. And since meta-theorems represent the theorems and inference rules of object-logics, those object-theorems and rules also have type `thm`. The meta-level inference rules are implemented in LCF style: as functions from theorems to theorems.

Theories have the ML type `theory`. Each object-logic has its own theory. Extending a logic with new constants and axioms creates a new theory. This is a basic step in developing proofs, and fortunately is much easier than creating an entire new logic.

Proofs are constructed using *tactics*. The simplest tactics apply an (object-level) inference rule to a subgoal, producing some new subgoals. Another simple tactic solves a goal by assumption under Isabelle's framework for natural deduction. Complex tactics typically apply other tactics repeatedly to certain goals, possibly using depth-first search or like strategies. Such tactics permit proofs to be performed at a higher level, with fewer top-level steps. The novice does not need to write new tactics, however: deriving new rules can also lead to shorter proofs, and is easier way than writing new tactics. Tactics have the ML type `tactic`.

1.1.4 How to get started

In order to conduct simple proofs you need to know some details about Isabelle. The following sections introduce theories, theorems, subgoal module commands, and tactics — including ML identifiers with their types. Although these concepts apply to all the object-logics, they are demonstrated within first-order logic. Ideally, you should have a terminal nearby where you can run Isabelle with this logic. If necessary, see the installation instructions for advice on setting things up.

What about the user interface? Isabelle's top level is simply the Standard ML system. If you are using a workstations that provides a window system, it is easy to make a menu: put common commands in a window where you can pick them up and insert them into an Isabelle session. This may seem uninviting, but once you get started on a serious project, you will see that the main problems are logical.

1.2 Theorems, rules, and theories

The theorems and rules of an object-logic are represented by theorems in the meta-logic. Each logic is defined by a theory. Isabelle provides many operations (as ML functions) that involve theorems, and some that involve theories. Chapters 4 and 5 present examples of theory construction. For now, we consider built-in theories.

1.2.1 Notation for theorems and rules

The keyboard rendering of the symbols of the meta-logic is summarized below. The precise syntax is described in Sections 5.3 and 5.4.4.

a `==` b	$a \equiv b$	meta-equality
ϕ `==>` ψ	$\phi \Longrightarrow \psi$	meta-implication
`[|` ϕ_1`;` $\ldots$ `;` ϕ_n `|] ==>` ψ	$\phi_1 \Longrightarrow (\cdots \phi_n \Longrightarrow \psi \cdots)$	nested implication
`!`xyz`.`ϕ	$\bigwedge xyz.\phi$	meta-quantification
`%`xyz`.`ϕ	$\lambda xyz.\phi$	meta-abstraction
`?P ?Q4 ?Ga12`	$?P \quad ?Q_4 \quad ?Ga_{12}$	scheme variables

Meta-abstraction is normally invisible, as in quantification. It comes into play when new binding operators are introduced: for example, an operator for defining primitive recursive functions.

Symbols of object-logics also must be rendered into keyboard characters. These typically are as follows:

P `&` Q	$P \& Q$	conjunction
P `|` Q	$P \vee Q$	disjunction
`~` P	$\neg P$	negation
P `-->` Q	$P \supset Q$	implication
P `<->` Q	$P \leftrightarrow Q$	bi-implication
`ALL` xyz`.`P	$\forall xyz.P$	for all
`EX` xyz`.`P	$\exists xyz.P$	there exists

To illustrate this notation, let us consider how first-order logic is formalized. Figure 1.1 presents the natural deduction system for intuitionistic first-order logic as implemented by Isabelle.

The rule &I is expressed by the meta-axiom

$$\bigwedge PQ.\, P \Longrightarrow (Q \Longrightarrow P \,\&\, Q)$$

This translates literally into keyboard characters as

```
!P Q. P ==> (Q ==> P&Q)
```

Leading universal quantifiers are usually dropped in favour of scheme variables:

```
?P ==> (?Q ==> ?P & ?Q)
```

	introduction (I)	*elimination* (E)
Conjunction	$\dfrac{A \quad B}{A \,\&\, B}$	$\dfrac{A \,\&\, B}{A}$ $\quad$ $\dfrac{A \,\&\, B}{B}$
Disjunction	$\dfrac{A}{A \vee B}$ $\quad$ $\dfrac{B}{A \vee B}$	$\dfrac{A \vee B \quad \overset{[A]}{C} \quad \overset{[B]}{C}}{C}$
Implication	$\dfrac{\overset{[A]}{B}}{A \supset B}$	$\dfrac{A \supset B \quad A}{B}$
Contradiction		$\dfrac{\bot}{A}$
Universal quantifier	$\dfrac{A}{\forall x.A}*$	$\dfrac{\forall x.A}{A[t/x]}$
Existential quantifier	$\dfrac{A[t/x]}{\exists x.A}$	$\dfrac{\exists x.A \quad \overset{[A]}{B}}{B}*$

**Eigenvariable conditions*:
$\forall$I: provided x not free in the assumptions
$\exists$E: provided x not free in B or in any assumption save A

Figure 1.1: Intuitionistic first-order logic

The parentheses are optional because `==>` groups to the right. We can also use the `[|...|]` shorthand:

```
[| ?P; ?Q |] ==> ?P & ?Q
```

Scheme variables are logically equivalent to ordinary variables, but may be instantiated during unification, while ordinary variables remain fixed. They have subscripts so that they can be renamed easily prior to resolution.

The definition of rules in an Isabelle theory usually involves ordinary variables:

```
[| P; Q |] ==> P & Q
```

Isabelle converts these into scheme variables so that the rule will work with unification. This convention for theories avoids cluttering the rules with question marks. When stating a goal, scheme variables are used only for answer extraction. Free variables in the goal will remain fixed throughout the proof, and will be converted to scheme variables afterwards.

A few more examples should make the syntax clearer. The rule $\vee$I is represented by the axiom

$$\bigwedge PQR\,.\,P \vee Q \Longrightarrow (P \Longrightarrow R) \Longrightarrow (Q \Longrightarrow R) \Longrightarrow R$$

In the Isabelle theory definition it is

```
[| P|Q; P==>R; Q==>R |] ==> R
```

The axiom representing $\forall$I has a nested meta-quantifier:

$$\bigwedge P\,.\,(\bigwedge x\,.\,P(x)) \Longrightarrow \forall x.P(x)$$

In the Isabelle theory definition it is

```
(!y. P(y)) ==> ALL x.P(x)
```

The 'Foundations' paper [9] and earlier versions of Isabelle enclose all object-formulae in special brackets, as in

$$[\![P]\!] \Longrightarrow ([\![Q]\!] \Longrightarrow [\![P \,\&\, Q]\!])$$

Although this notation clearly distinguishes the object and meta-levels, it is verbose and has therefore been dropped. Note that $P \Longrightarrow Q$ is ambiguous: are P and Q meta or object-formulae? Isabelle always assumes that object-formulae are intended. This is a matter of types: meta-formulae have type *prop* while object-formulae typically have type *form*. You can force a variable to have type *prop* by a type constraint, as in

```
?psi$$prop ==> ?theta$$prop
```

The occasions for doing this are few.

1.2.2 The type of theorems and its operations

The inference rules, theorems, and axioms of a logic have type `thm`. Theorems and axioms can be regarded as rules with no premises, so let us speak just of rules. Each proof is constructed by a series of rule applications, usually through subgoal module commands.

The type `thm` is an abstract type. Since ML will not print values of abstract types, you have to use the `prth` command. The rules of a theory are normally bound to ML identifiers. Suppose we are running an Isabelle session with natural deduction first-order logic, which is described in Chapter 2. Then we can print `disj_intr1`, which is one of the $\vee$I rules:

```
> prth disj_intr1;
?P ==> ?P | ?Q
val it = () : unit
```

User's input is preceded by a `>` character, which is the prompt of the Poly/ML system. The prompt character for input lines after the first is `#`. The response `val it = ()` will henceforth be omitted!

The command `prths` prints a list of theorems. Here are the definitional axioms of first-order logic:

```
> prths [True_def, not_def, iff_def];
True == False --> False

~?P == ?P --> False

?P <-> ?Q == (?P --> ?Q) & (?Q --> ?P)
```

Summary of the theorem printing commands:

```
prth: thm -> unit
prths: thm list -> unit
```

1.2.3 The type of theories

Each logic is an ML object of type `theory`. For natural deduction first-order logic, the identifiers are `Int_Rule.thy` (for intuitionistic logic) and `cla_thy` (for classical logic). A theory includes information about types, constants, and syntax. In particular, each theory includes information about how to parse a string into a logical formula. Unlike LCF, an Isabelle session is not restricted to a single theory. A theory is stated at the start of each proof: the theory of the initial goal. Most tactics work within this theory, while certain Isabelle functions take a theory as argument.

Type `theory` is also an abstract type, so theory values cannot be printed. There is no way of opening up a theory value to see what is inside.

1.3 The subgoal module

Most Isabelle proofs are conducted through the subgoal module, which maintains a proof state and manages the proof construction. Isabelle is largely a functional program, but this kind of interaction is imperative. From the ML top-level you can invoke commands (which are ML functions, with side effects) to establish a goal, apply a tactic to the proof state, undo a proof step, and finally obtain the theorem that has been proved.

Tactics, which are operations on proof states, are described in the following section. We peek at them here, however, during the demonstration of the subgoal commands.

1.3.1 Basic commands

To start a new proof, type

`goal` *theory* *formula* `;`

where the *formula* is written as an ML string.

To apply a tactic to the proof state, type

`by` *tactic* `;`

A tactic can do anything to a proof state — even replace it by a completely unrelated state — but most tactics apply a rule or rules to a numbered subgoal.

At the end of the proof, call `result()` to get the theorem you have just proved.

If ever you are dissatisfied with a previous step, type `undo()` to cancel it. The undo operation can be repeated.

To demonstrate these commands, consider the following elementary proof. We enter the goal $P \supset P \vee Q$ in classical first-order logic.

```
> goal cla_thy "P --> P | Q";
Level 0
P --> P | Q
 1. P --> P | Q
val it = [] : thm list
```

Isabelle responds by printing the proof state, which has the same formula (namely $P \supset P \vee Q$) as the main goal and as the only subgoal — as always for the initial proof state. Note that `goal` has returned an empty theorem list; we can ignore this unless we are deriving a rule. The level number of the state is the number of tactics that have been applied to it, so we begin at Level 0.

The first step is 'by' `resolve_tac` (described in the next section), which applies the rule `imp_intr` ($\supset$I) to subgoal 1:

```
> by (resolve_tac [imp_intr] 1);
Level 1
P --> P | Q
 1. P ==> P | Q
```

In the new proof state, subgoal 1 is $P \vee Q$ under the assumption P. (The meta-implication `==>` indicates assumptions.) We now apply the rule `disj_intr1` to that subgoal:

```
> by (resolve_tac [disj_intr1] 1);
Level 2
P --> P | Q
 1. P ==> P
```

At Level 2 the one subgoal is 'prove P assuming P'. That is easy, by the tactic `assume_tac`.

```
> by (assume_tac 1);
Level 3
P --> P | Q
No subgoals!
```

Isabelle tells us that there are no longer any subgoals: the proof is complete. Once finished, call `result()` to get the theorem you have just proved.

```
> val mythm = result();
val mythm = ? : thm
```

ML will not print theorems unless we force it to:

```
> prth mythm;
?P --> ?P | ?Q
```

Note that `P` and `Q` have changed from free variables into the scheme variables `?P` and `?Q`. As free variables, they remain fixed throughout the proof; as scheme variables, the theorem `mythm` can be applied with any formulae in place of P and Q. Here we go back to Level 2:

```
> undo();
Level 2
P --> P | Q
 1. P ==> P
```

You can `undo()` and `undo()` right back to the beginning. But `undo()` is irreversible. Incidentally, do not omit the parentheses:

```
> undo;
val it = fn : unit -> unit
```

This is just a function value!

1.3.2 More advanced commands

The following commands are mainly of importance to experienced users, so feel free to skip this section on the first reading.

The subgoal package stores the current proof state and many previous states; commands can produce new states or return to previous ones. The *state list* at level n is a list of pairs

$$[(\psi_n, \Psi_n),\ (\psi_{n-1}, \Psi_{n-1}),\ \ldots,\ (\psi_0, [])]$$

where ψ_n is the current proof state, ψ_{n-1} is the previous one, ..., and ψ_0 is the initial proof state. The Ψ_i are sequences of proof states, storing branch points where a tactic returned a sequence longer than one.

Chopping elements from the state list reverts to previous proof states. Besides this, the `undo` command uses a list of previous states of the package itself.

To print the current proof state, type `pr()`. Calling `prlev n` prints the proof state at level `n`. The variable `goals_limit`, initially 10, holds the upper bound for the number of subgoals to print.

Starting at the top level, `back` looks down the state list for an alternative state. The first one found becomes the current proof state. The previous state is discarded and the level is reset to that where the alternative was found.

Calling `chop()` deletes the top level of the state list, cancelling the effect of the last `by` command. It provides a limited undo facility, and the `undo()` command can cancel its effect. Note that `undo()` cannot undo itself. Calling `choplev n` truncates the state list to level `n`. This is quicker than typing `chop()` or `undo()` several times.

Calling `topthm()` returns the top level proof state, which is a theorem. This is not the best way to extract the theorem you have proved: try `result()` or `uresult()`.

Calling `result()` returns the final theorem. It tidies this theorem, generalizing its free variables and discharging its assumptions, and it raises an exception unless the proof state has zero subgoals and the theorem proved is the same as the one stated in the `goal` command. They could differ if the proof involved answer extraction, for example. In that case you should use `uresult()`, which omits the comparison of the initial goal with the final theorem.

Calling `getgoal i` returns subgoal `i` as a term. When debugging a tactic you might employ this function.

In the middle of a proof you may discover that a lemma needs to be proved first. Isabelle provides commands to let you put aside the current proof, prove the lemma, and finally resume the previous proof. Call `getstate()` to return the entire state of the subgoal package. (This object belongs to the abstract type `gstack`.) Bind this state to an ML identifier, say `save`. To resume the proof, call `setstate(save)`.

Summary of these subgoal module commands:

```
back: unit -> unit
by: tactic -> unit
chop: unit -> unit
choplev: int -> unit
getgoal: int -> term
getstate: unit -> gstack
goal: theory -> string -> thm list
goals_limit: int ref
pr: unit -> unit
prlev: int -> unit
result: unit -> thm
setstate: gstack -> unit
topthm: unit -> thm
undo: unit -> unit
uresult: unit -> thm
```

1.4 Tactics

Tactics are operations on the proof state, such as, 'apply the following rules to these subgoals'. For the time being, you may want to regard them as part of the syntax of the `by` command. They have a separate existence because they can be combined — using operators called *tacticals* — into more powerful tactics. Those tactics can be similarly combined, and so on.

Tacticals are not discussed until Chapter 4. Here we consider only the most basic tactics. Fancier tactics are provided in the built-in logics, so you should still be able to do substantial proofs.

Applying a tactic changes the proof state to a new proof state. A tactic may produce multiple outcomes, permitting backtracking and search. For now, let us pretend that a tactic can produce at most one next state. When a tactic produces no next state, it is said to *fail*.

The tactics given below each act on a subgoal designated by a number, starting from 1. They fail if the subgoal number is out of range.

To understand tactics, you will need to have read 'The Foundation of a Generic Theorem Prover' [9] — particularly the discussion of backwards proof. We shall perform some proofs from that paper in Isabelle.

The basic resolution tactic, used for most proof steps, is

`resolve_tac` *thms* *i*

The *thms* represent object-rules. The rules in this list are tried against subgoal i of the proof state. For a given rule, resolution can form the next state by unifying the conclusion with the subgoal, replacing it by the instantiated premises. There can be many outcomes: many of the rules may be unifiable, and for each there can be

many (higher-order) unifiers. The tactic fails if none of the rules can be applied to the subgoal.

In a natural deduction proof, a subgoal's assumptions are represented by meta-implication. Resolution lifts object-rules over any assumptions: in effect, the assumptions are copied to the new subgoals. Eigenvariables in a subgoal are represented by meta-quantifiers; resolution also lifts object-rules over these.

The tactic to solve subgoal i by assumption is

`assume_tac` i

Isabelle can recognize when a subgoal holds by assumption, but you must tell it to by applying this tactic. They are not simply erased. Proving a subgoal by assumption can involve unification, instantiating variables shared with other subgoals — and possibly making them false. The tactic fails if subgoal i cannot be solved by assumption.

The elimination resolution tactic is

`eresolve_tac` *thms* i

Like `resolve_tac` *thms* i followed by `assume_tac` i, it applies an object-rule and then solves its first premise by assumption. But `eresolve_tac` does one thing more: it deletes that assumption from the other subgoals resulting from the resolution. The assumption is used once then discarded. The tactic is appropriate for typical elimination rules, where applying the rule generates new assumptions that are stronger than the old. Also, it does two steps in one.

The following tactics are less important. They permit reasoning about definitions and deriving rules, and are demonstrated in Chapter 4.

Three rewriting tactics are

`rewrite_goals_tac` *thms*
`rewrite_tac` *thms*
`fold_tac` *thms*

For each, *thms* is a list of equational theorems of the form $t \equiv u$. These must be theorems rather than rules: they must have no premises. Both `rewrite_goals_tac` and `rewrite_tac` apply these as left-to-right rewrite rules. However `rewrite_tac` rewrites the entire proof state, including the main goal, while `rewrite_goals_tac` rewrites the subgoals only, which is normally preferable. Calling `fold_tac` applies the theorems as *right-to-left* rewrite rules in the proof state. Typically `rewrite_goals_tac` is used to expand definitions in subgoals, while `fold_tac` inverts this operation.

Calling `cut_facts_tac` *thms* i inserts the *thms* as assumptions in subgoal i. This allows `eresolve_tac` or `rewrite_goals_tac` to operate on the *thms*. Only those rules that are outright theorems, with no premises, are inserted; `eresolve_tac` cannot cope with general rules as assumptions. In many cases the *thms* are in fact the premises of a rule being derived, as illustrated in Chapter 4.

Summary of these tactics:

```
assume_tac: int -> tactic
cut_facts_tac: thm list -> int -> tactic
eresolve_tac: thm list -> int -> tactic
fold_tac: thm list -> tactic
resolve_tac: thm list -> int -> tactic
rewrite_goals_tac: thm list -> tactic
rewrite_tac: thm list -> tactic
```

The tactics `resolve_tac`, `assume_tac`, and `eresolve_tac` suffice for most single-step proofs. The examples below demonstrate how the subgoal commands and tactics are used in practice. Although `eresolve_tac` is not strictly necessary, it simplies proofs that involve elimination rules.

1.4.1 A first example

Let us do the first example proof from 'Foundations' [9]. We enter the goal:

```
> goal Int_Rule.thy "P&Q  --> (R-->P&R)";
Level 0
P & Q --> R --> P & R
 1. P & Q --> R --> P & R
```

There is one subgoal; we apply $\supset$I to it:

```
> by (resolve_tac [imp_intr] 1);
Level 1
P & Q --> R --> P & R
 1. P & Q ==> R --> P & R
```

The one subgoal has an assumption, $P \,\&\, Q$, but its outer form is still an implication. We apply the same rule again.

```
> by (resolve_tac [imp_intr] 1);
Level 2
P & Q --> R --> P & R
 1. [| P & Q; R |] ==> P & R
```

There are two assumptions ($P \,\&\, Q$ and R), and the outer form is conjunctive. So apply the rule &I:

```
> by (resolve_tac [conj_intr] 1);
Level 3
P & Q --> R --> P & R
 1. [| P & Q; R |] ==> P
 2. [| P & Q; R |] ==> R
```

Now there are two subgoals, with the same assumptions as before. Subgoal 2 holds trivially by assumption:

```
> by (assume_tac 2);
Level 4
P & Q --> R --> P & R
 1. [| P & Q; R |] ==> P
```

Noting the assumption $P \,\&\, Q$, we work backwards from P by applying a version of &E:

```
> by (resolve_tac [conjunct1] 1);
Level 5
P & Q --> R --> P & R
 1. [| P & Q; R |] ==> P & ?Q3
```

The subgoal contains a scheme variable, `?Q3`, which can be instantiated to any formula. Therefore the subgoal is provable by assumption.

```
> by (assume_tac 1);
Level 6
P & Q --> R --> P & R
No subgoals!
```

We bind our theorem to an ML variable and inspect it.

```
> val example1 = result();
val example1 = ? : thm
> prth example1;
?P & ?Q --> ?R --> ?P & ?R
```

1.4.2 An example with elimination rules

The proof that disjunction is commutative requires use of $\vee$E.

We enter $(P \vee Q) \supset (Q \vee P)$ to Isabelle and apply $\supset$I:

```
> goal Int_Rule.thy "P|Q --> Q|P";
Level 0
P | Q --> Q | P
 1. P | Q --> Q | P
> by (resolve_tac [imp_intr] 1);
Level 1
P | Q --> Q | P
 1. P | Q ==> Q | P
```

The assumption $P \vee Q$ being available, we apply $\vee$E, here using `resolve_tac`.

```
> by (resolve_tac [disj_elim] 1);
Level 2
P | Q --> Q | P
 1. P | Q ==> ?P1 | ?Q1
 2. [| P | Q; ?P1 |] ==> Q | P
 3. [| P | Q; ?Q1 |] ==> Q | P
```

This elimination rule has three premises, of which the first is any disjunction: hence the subgoal `?P1|?Q1`. Proving this subgoal will instantiate `?P1` and `?Q1` in the other subgoals. We prove subgoal 1 by `assume_tac`, instantiating `?P1` to `P` and `?Q1` to `Q`.

```
> by (assume_tac 1);
Level 3
P | Q --> Q | P
 1. [| P | Q; P |] ==> Q | P
 2. [| P | Q; Q |] ==> Q | P
```

The old subgoal 1 disappears, and subgoals 2 and 3 slide down to fill the gap. Both of these are provable thanks to their new assumptions. Since $P \vee Q$ follows from P and also from Q, that assumption is now redundant in both subgoals. In the following example we shall apply $\vee$E using `eresolve_tac`, which will delete this assumption.

For now, let us prove subgoal 1 by $\vee$I and assumption.

```
> by (resolve_tac [disj_intr2] 1);
Level 4
P | Q --> Q | P
 1. [| P | Q; P |] ==> P
 2. [| P | Q; Q |] ==> Q | P
> by (assume_tac 1);
Level 5
P | Q --> Q | P
 1. [| P | Q; Q |] ==> Q | P
```

The remaining subgoal is proved similarly.

```
> by (resolve_tac [disj_intr1] 1);
Level 6
P | Q --> Q | P
 1. [| P | Q; Q |] ==> Q
> by (assume_tac 1);
Level 7
P | Q --> Q | P
No subgoals!
```

Now `result()` should be called to return the theorem that has just been proved. Once a new goal is entered, this theorem will be lost.

1.4.3 An example of `eresolve_tac`

Using `eresolve_tac` instead of `resolve_tac` in the above proof makes it three steps shorter, and perhaps clearer. The first use of `eresolve_tac` is the most important, for it involves an elimination rule ($\vee$E) and the deletion of an assumption.

Let us again enter $(P \vee Q) \supset (Q \vee P)$ and apply $\supset$I. (If you have done the previous example on a terminal, type `undo()` six times to get back to Level 1.)

```
> goal Int_Rule.thy "P|Q --> Q|P";
Level 0
P | Q --> Q | P
 1. P | Q --> Q | P
> by (resolve_tac [imp_intr] 1);
Level 1
P | Q --> Q | P
 1. P | Q ==> Q | P
```

The first premise of $\vee$E is the formula being eliminated: the disjunction. The tactic `eresolve_tac` searches among the assumptions for one that unifies with the first premise, simultaneously unifying the conclusion of this rule with the subgoal. (The conclusion of $\vee$E unifies with any formula, for it is simply R.) It uses the selected assumption to prove the first premise, and deletes that assumption from the resulting subgoals. In short, the assumption is eliminated.

```
> by (eresolve_tac [disj_elim] 1);
Level 2
P | Q --> Q | P
 1. P ==> Q | P
 2. Q ==> Q | P
```

Although subgoals 1 and 2 now have only one assumption (compared with two in the previous proof), they can be proved exactly as before. But `eresolve_tac` simplifies these steps also, for they consist of application of a rule followed by proof by assumption.

```
> by (eresolve_tac [disj_intr2] 1);
Level 3
P | Q --> Q | P
 1. Q ==> Q | P
```

The same thing again ...

```
> by (eresolve_tac [disj_intr1] 1);
Level 4
P | Q --> Q | P
No subgoals!
```

The importance of `eresolve_tac` is clearer in larger proofs. It prevents assumptions from accumulating and getting reused. The eliminated assumption is redundant with most elimination rules save $\forall$E. Their deletion is especially important in automatic tactics.

1.5 Proofs involving quantifiers

One of the most important aspects of Isabelle is the treatment of quantifier reasoning. We can illustrate this by comparing a proof of $\forall x\,.\,\exists y\,.\,x = y$ with an attempted proof of $\exists y\,.\,\forall x\,.\,x = y$ (which happens to be false). The one proof succeeds and the other fails because of the scope of quantified variables. These proofs are also discussed in 'Foundations' [9].

Unification helps even in these trivial proofs. In $\forall x\,.\,\exists y\,.\,x = y$ the y that 'exists' is simply x, but we need never say so. This choice is forced by the reflexive law for equality, and it happens automatically. The proof forces the correct instantiation of variables. Of course, if the instantiation is complicated, it may not be found in a reasonable amount of time!

1.5.1 A successful quantifier proof

The theorem $\forall x\,.\,\exists y\,.\,x = y$ is one of the simplest to contain both quantifiers. Its proof illustrates the use of the introduction rules $\forall$I and $\exists$I.

To begin, we enter the goal:

```
> goal Int_Rule.thy "ALL x. EX y. x=y";
Level 0
ALL x. EX y. x = y
 1. ALL x. EX y. x = y
```

The only applicable rule is $\forall$I:

```
> by (resolve_tac [all_intr] 1);
Level 1
ALL x. EX y. x = y
 1. !ka. EX y. ka = y
```

The `!ka` introduces an eigenvariable in the subgoal. The exclamation mark is the character for meta-forall ($\bigwedge$). The subgoal must be proved for all possible values of `ka`. Isabelle chooses the names of eigenvariables; they are always `ka`, `kb`, `kc`, ..., in that order.

The only applicable rule is $\exists$I:

```
> by (resolve_tac [exists_intr] 1);
Level 2
ALL x. EX y. x = y
 1. !ka. ka = ?a1(ka)
```

Note that the bound variable `y` has changed into `?a1(ka)`, where `?a1` is a scheme variable. It is also a function, and is applied to `ka`. Instantiating `?a1` will change `?a1(ka)` into a term that may — or may not — contain `ka`. In particular, if

?a1 is instantiated to the identity function, ?a1(ka) changes into simply ka. This corresponds to proof by the reflexivity of equality.

```
> by (resolve_tac [refl] 1);
Level 3
ALL x. EX y. x = y
No subgoals!
```

The proof is finished. Unfortunately we cannot observe the instantiation of ?a1 because it appears nowhere else.

1.5.2 An unsuccessful quantifier proof

The formula $\exists y \,.\, \forall x \,.\, x = y$ is not a theorem. Let us hope that Isabelle cannot prove it!

We enter the goal:

```
> goal Int_Rule.thy "EX y. ALL x. x=y";
Level 0
EX y. ALL x. x = y
 1. EX y. ALL x. x = y
```

The only rule that can be considered is ∃I:

```
> by (resolve_tac [exists_intr] 1);
Level 1
EX y. ALL x. x = y
 1. ALL x. x = ?a
```

The scheme variable ?a may be replaced by any term to complete the proof. Problem is, no term is equal to all x. We now must apply ∀I:

```
> by (resolve_tac [all_intr] 1);
Level 2
EX y. ALL x. x = y
 1. !ka. ka = ?a
```

Compare our position with the previous Level 2. Where before was ?a1(ka) there is now ?a. In both cases the scheme variable (whether ?a1 or ?a) can only be instantiated by a term that is free in the entire proof state. But so doing can change ?a1(ka) into something that depends upon ka. In our present position we can do nothing. The reflexivity axiom does not unify with subgoal 1 because ka is a bound variable. Here is what happens if we try:

```
> by (resolve_tac [refl] 1);
by: tactic returned no results
Exception- ERROR raised
```

You do not have to think about the β-reduction that changes ?a1(ka) into ka. Instead, regard ?a1(ka) as any term possibly containing ka.

1.5.3 Nested quantifiers

Multiple quantification produces complicated terms. Consider this contrived example. Without more information about P we cannot prove anything, but observe how the eigenvariables and scheme variables develop.

```
> goal Int_Rule.thy "EX u.ALL x.EX v.ALL y.EX w. P(u,x,v,y,w)";
Level 0
EX u. ALL x. EX v. ALL y. EX w. P(u,x,v,y,w)
 1. EX u. ALL x. EX v. ALL y. EX w. P(u,x,v,y,w)
> by (resolve_tac [exists_intr, all_intr] 1);
Level 1
EX u. ALL x. EX v. ALL y. EX w. P(u,x,v,y,w)
 1. ALL x. EX v. ALL y. EX w. P(?a,x,v,y,w)
```

The scheme variable `?a` has appeared.

Note that `resolve_tac`, if given a list of rules, will choose a rule that applies. Here the only rules worth considering are ∀I and ∃I.

```
> by (resolve_tac [exists_intr, all_intr] 1);
Level 2
EX u. ALL x. EX v. ALL y. EX w. P(u,x,v,y,w)
 1. !ka. EX v. ALL y. EX w. P(?a,ka,v,y,w)
> by (resolve_tac [exists_intr, all_intr] 1);
Level 3
EX u. ALL x. EX v. ALL y. EX w. P(u,x,v,y,w)
 1. !ka. ALL y. EX w. P(?a,ka,?a2(ka),y,w)
```

The bound variable `ka` and scheme variable `?a2` have appeared. Note that `?a2` is applied to the bound variables existing at the time of its introduction — but not, of course, to bound variables introduced later.

```
> by (resolve_tac [exists_intr, all_intr] 1);
Level 4
EX u. ALL x. EX v. ALL y. EX w. P(u,x,v,y,w)
 1. !ka kb. EX w. P(?a,ka,?a2(ka),kb,w)
> by (resolve_tac [exists_intr, all_intr] 1);
Level 5
EX u. ALL x. EX v. ALL y. EX w. P(u,x,v,y,w)
 1. !ka kb. P(?a,ka,?a2(ka),kb,?a4(ka,kb))
```

In the final state, `?a` cannot become any term containing `ka` or `kb`, while `?a2(ka)` can become a term containing `ka`, and `?a4(ka,kb)` can become a term containing both bound variables. This example is discussed in 'Foundations' [9].

1.6 Priority Grammars

In the remainder of this manual we shall frequently define the precise syntax of some logic by means of context-free grammars. These grammars obey the following

conventions: identifiers denote nonterminals, `typewriter` fount denotes terminals, constructs enclosed in {.} can be repeated 0 or more times (Kleene star), and alternatives are separated by |. The predefined categories of alphanumeric identifiers and of scheme variables are denoted by *identifier* and *variable* respectively (see Section 5.4.2).

In order to simplify the description of mathematical languages, we introduce an extended format which permits *priorities* or *precedences*. This scheme generalizes precedence declarations in ML and PROLOG. In this extended grammar format, nonterminals are decorated by integers, their priority. In the sequel, priorities are shown as subscripts. A nonterminal A_p on the right-hand side of a production may only be rewritten using a production $A_q = \gamma$ where q is not less than p.

Formally, a set of context free productions G induces a derivation relation $\longrightarrow_G$ on strings as follows:

$$\alpha A_p \beta \longrightarrow_G \alpha\gamma\beta \quad \textit{iff} \quad \exists q \geq p.\ (A_q{=}\gamma) \in G$$

Any extended grammar of this kind can be translated into a normal context free grammar. However, this translation may require the introduction of a large number of new nonterminals and productions.

Example 1.1 The following simple grammar for arithmetic expressions demonstrates how binding power and associativity of operators can be enforced by priorities.

$$\begin{array}{lcl} A_9 & = & \texttt{0} \\ A_9 & = & \texttt{(}\ A_0\ \texttt{)} \\ A_0 & = & A_0\ \texttt{+}\ A_1 \\ A_2 & = & A_3\ \texttt{*}\ A_2 \\ A_3 & = & \texttt{-}\ A_3 \end{array}$$

The choice of priorities determines that "`-`" binds tighter than "`*`" which binds tighter than "`+`", and that "`+`" and "`*`" associate to the left and right, respectively.

To minimize the number of subscripts, we adopt the following conventions:

- all priorities p must be in the range $0 \leq p \leq m$ for some fixed m.
- priority 0 on the right-hand side and priority m on the left-hand side may be omitted.

In addition, we write the production $A_p = \alpha$ as $A = \alpha\ (p)$.

Using these conventions and assuming $m = 9$, the grammar in Example 1.1 becomes

$$\begin{array}{lcll} A & = & \texttt{0} & \\ & | & \texttt{(}\ A\ \texttt{)} & \\ & | & A\ \texttt{+}\ A_1 & (0) \\ & | & A_3\ \texttt{*}\ A_2 & (2) \\ & | & \texttt{-}\ A_3 & (3) \end{array}$$

Priority grammars are not just used to describe logics in this manual. They are also supported by Isabelle's syntax definition facility (see Chapter 5).

Chapter 2

Isabelle's First-Order Logics

Although Isabelle has been developed to be a generic theorem prover — one that you can customize to your own logics — several logics come with it. They reside in various subdirectories and can be built using the installation instructions. You can use Isabelle simply as an implementation of one of these. This chapter describes the three versions of first-order logic provided with Isabelle. The following chapter describes set and type theories.

First-order logic with natural deduction comes in both constructive and classical versions. First-order logic is also available as the classical sequent calculus LK. Each sequent has the form $A_1, \ldots, A_m \vdash B_1, \ldots, B_n$. This formulation is equivalent to deductive tableaux.

Each object-logic comes with simple proof procedures. These are reasonably powerful (at least for interactive use), though neither complete nor amazingly scientific. You can use them as they are or take them as examples of tactical programming. You can perform single-step proofs using `resolve_tac` and `assume_tac`, referring to the inference rules of the logic by ML identifiers of type `thm`.

Call a rule *safe* if when applied to a provable goal the resulting subgoals will also be provable. If a rule is safe then it can be applied automatically to a goal without destroying our chances of finding a proof. For instance, all the rules of the classical sequent calculus LK are safe. Intuitionistic logic includes some unsafe rules, like disjunction introduction ($P \vee Q$ can be true when P is false) and existential introduction ($\exists x \,.\, P(x)$ can be true when $P(a)$ is false for certain a). Universal elimination is unsafe if the formula $\forall x \,.\, P(x)$ is deleted after use, which is necessary for termination.

Proof procedures use safe rules whenever possible, delaying the application of unsafe rules. Those safe rules are preferred that generate the fewest subgoals. Safe rules are (by definition) deterministic, while the unsafe rules require search. The design of a suitable set of rules can be as important as the strategy for applying them.

Many of the proof procedures use backtracking. Typically they attempt to solve subgoal i by repeatedly applying a certain tactic to it. This tactic, which is known as a *step tactic*, resolves a selection of rules with subgoal i. This may replace one subgoal by many; but the search persists until there are fewer subgoals in total than at the start. Backtracking happens when the search reaches a dead end: when the step tactic fails. Alternative outcomes are then searched by a depth-first or best-first

strategy. Techniques for writing such tactics are discussed in the next chapter.

Each logic is distributed with many sample proofs, some of which are described below. Though future Isabelle users will doubtless find better proofs and tactics than mine, the examples already show that Isabelle can prove interesting theorems in various logics.

2.1 First-order logic with natural deduction

The directory `FOL` contains theories for first-order logic based on Gentzen's natural deduction systems (which he called NJ and NK). Intuitionistic logic is first defined, then classical logic is obtained by adding the double negation rule.

Natural deduction typically involves a combination of forwards and backwards reasoning, particularly with the rules &E, $\supset$E, and $\forall$E. Isabelle's backwards style handles these rules badly, so alternative rules are derived to eliminate conjunctions, implications, and universal quantifiers. The resulting system is similar to a cut-free sequent calculus.

Basic proof procedures are provided. The intuitionistic prover works with derived rules to simplify uses of implication in the assumptions. Far from complete, it can still prove many complex theorems automatically. The classical prover works like a straightforward implementation of LK, which is like a deductive tableaux prover. It is not complete either, though less flagrantly so.

A serious theorem prover for classical logic would exploit sophisticated methods for efficiency and completeness. Most known methods, unfortunately, work only for certain fixed inference systems. With Isabelle you often work in an evolving inference system, deriving rules as you go. Classical resolution theorem provers are extremely powerful, of course, but do not support interactive proof.

The type of expressions is *term* while the type of formulae is *form*. The ML names for these types are `Aterm` and `Aform` respectively. The infixes are the equals sign and the connectives. Note that `-->` has two meanings: in ML it is a constructor of the type `typ`, while in `FOL` it is the implication sign. Figure 2.1 gives the syntax, including translations for the quantifiers.

2.1.1 Intuitionistic logic

The intuitionistic theory has the ML identifier `Int_Rule.thy`. Figure 2.2 shows the inference rules with their ML names. The connective $\leftrightarrow$ is defined through & and $\supset$; introduction and elimination rules are derived for it. Derived rules are shown in Figure 2.3, again with their ML names.

The hardest rule for search is implication elimination (whether expressed by `mp` or `imp_elim`). Given $P \supset Q$ we may assume Q provided we can prove P. In classical logic the proof of P can assume $\neg P$, but the intuitionistic proof of P may require repeated use of $P \supset Q$. If the proof of P fails then the whole branch of the proof must be abandoned. Thus intuitionistic propositional logic requires backtracking. For an elementary example, consider the intuitionistic proof of Q from $P \supset Q$ and

symbol	*meta-type*	*precedence*	*description*
`=`	$[term, term] \to form$	Left 50	equality ($=$)
`&`	$[form, form] \to form$	Right 35	conjunction ($\&$)
`|`	$[form, form] \to form$	Right 30	disjunction ($\vee$)
`-->`	$[form, form] \to form$	Right 25	implication ($\supset$)
`<->`	$[form, form] \to form$	Right 25	if and only if ($\leftrightarrow$)

Infixes

symbol	*meta-type*	*description*
`Trueprop`	$form \to prop$	meta-predicate of truth
`~`	$form \to form$	negation ($\neg$)
`Forall`	$(term \to form) \to form$	universal quantifier ($\forall$)
`Exists`	$(term \to form) \to form$	existential quantifier ($\exists$)
`True`	$form$	tautologous formula ($\top$)
`False`	$form$	absurd formula ($\bot$)

Constants

external	*internal*	*standard notation*
`ALL` x`.` P	`Forall(`$\lambda x.P$`)`	$\forall x.P$
`EX` x`.` P	`Exists(`$\lambda x.P$`)`	$\exists x.P$

Translations

$$
\begin{array}{rcll}
form & = & \texttt{ALL}\ identifier\ \{\ identifier\ \}\ .\ form & (10) \\
& | & \texttt{EX}\ \ identifier\ \{\ identifier\ \}\ .\ form & (10) \\
& | & \texttt{\~{}}\ form_{40} & (40) \\
& | & others\ldots &
\end{array}
$$

Grammar

Figure 2.1: Syntax of `FOL`

```
refl        a=a
sym         a=b ==> b=a
trans       [| a=b;  b=c |] ==> a=c
```

EQUALITY

```
conj_intr   [| P;  Q |] ==> P&Q
conjunct1   P&Q ==> P
conjunct2   P&Q ==> Q

disj_intr1  P ==> P|Q
disj_intr2  Q ==> P|Q
disj_elim   [| P|Q;  P ==> R;  Q ==> R |] ==> R

imp_intr    (P ==> Q) ==> P-->Q
mp          [| P-->Q;  P |] ==> Q

False_elim  False ==> P

True_def    True == False-->False
not_def     ~P == P-->False
iff_def     P<->Q == (P-->Q) & (Q-->P)
```

PROPOSITIONAL RULES

```
all_intr        (!y. P(y)) ==> ALL x.P(x)
spec            ALL x.P(x) ==> P(a)

exists_intr     P(a) ==> EX x.P(x)
exists_elim     [| EX x.P(x);  !y.P(y) ==> R |] ==> R
```

QUANTIFIER RULES

Figure 2.2: Meta-axioms for intuitionistic FOL

```
conj_elim  [| P&Q; [| P; Q |] ==> R |] ==> R
imp_elim   [| P-->Q;  P;  Q ==> R |] ==> R
all_elim   [| ALL x.P(x); P(a) ==> R |] ==> R
```

Sequent-style elimination rules

```
contr      [| ~P;  P |] ==> False
not_intr   (P ==> False) ==> ~P
not_elim   [| ~P;  P |] ==> R

iff_intr    [| P ==> Q;  Q ==> P |] ==> P<->Q
iff_elim    [| P <-> Q;  [| P-->Q; Q-->P |] ==> R |] ==> R
```

Derived rules for ¬ and ↔

```
conj_imp_elim   [| (P&Q)-->S;  P-->(Q-->S) ==> R |] ==> R
disj_imp_elim   [| (P|Q)-->S;  [| P-->S; Q-->S |] ==> R |] ==> R
imp_imp_elim    [| (P-->Q)-->S;  [| P; Q-->S |] ==> Q;
                   S ==> R |] ==> R
all_imp_elim    [| (ALL x.P(x))-->S;  !x.P(x);  S ==> R |] ==> R
exists_imp_elim [| (EX x.P(x))-->S;  P(a)-->S ==> R |] ==> R
```

Intuitionistic simplification of implication

```
disj_cintr    (~Q ==> P) ==> P|Q
exists_cintr  (ALL x. ~P(x) ==> P(a)) ==> EX x.P(x)
ex_middle     ~P | P
imp_celim     [| P-->Q; ~P ==> R; Q ==> R |] ==> R
iff_celim     [| P<->Q;  [| P; Q |] ==> R;
                 [| ~P; ~Q |] ==> R |] ==> R
swap          ~P ==> (~Q ==> P) ==> Q
```

Derived rules for classical logic

Figure 2.3: Derived rules for FOL

$(P \supset Q) \supset P$. The implication $P \supset Q$ is needed twice.

$$\dfrac{P \supset Q \qquad \dfrac{(P \supset Q) \supset P \qquad P \supset Q}{P}}{Q}$$

The theorem prover for intuitionistic logic avoids `imp_elim`, trying to simplify implication with derived rules (Figure 2.3). The idea is to reduce the antecedents of implications to atoms and then use Modus Ponens: from $P \supset Q$ and P deduce Q. Some of the derived rules are still unsafe, and so the method is incomplete.

The following belong to the structure `Int_Prover`. This structure is open, but using the full identifier will avoid name clashes, for instance between `Int_Prover.step_tac` and `Pc.step_tac`.

The tactic `mp_tac` performs Modus Ponens among the assumptions. Calling `mp_tac i` searches for assumptions of the form $P \supset Q$ and P in subgoal i. It replaces that subgoal by a new one where $P \supset Q$ has been replaced by Q. Unification may take place, selecting any implication whose antecedent is unifiable with another assumption. If more than one pair of assumptions satisfies these conditions, the tactic will produce multiple outcomes.

The tactic `safestep_tac` performs one safe step. Calling `safestep_tac i` tries to solve subgoal i completely by assumption or absurdity, then tries `mp_tac`, then tries other safe rules. It is badly named: due to unification, it is not really safe. If `mp_tac` instantiates some variables, for example, then the resulting subgoals could be unprovable. It may produce multiple outcomes.

Calling `safe_tac i` tries to solve subgoal i by backtracking, with `safestep_tac i` as the step tactic. This tactic is useful for demonstrations and debugging. It solves the easy parts of the proof while leaving the hard parts.

The tactic `step_tac` performs one step of the basic strategy. Calling `step_tac i` tries to reduce subgoal i by `safestep_tac i`, then tries unsafe rules. It may produce multiple outcomes.

The main theorem-proving tactic is `pc_tac`. Calling `pc_tac i` tries to solve subgoal i by backtracking, with `step_tac i` as the step tactic.

The following are some of the many theorems that `pc_tac` proves automatically. The latter three are from *Principia Mathematica* (*11.53, *11.55, *11.61) [14].

```
(~ ~ P) & ~ ~ (P --> Q) --> (~ ~ Q)

(ALL x y. P(x) --> Q(y)) <-> ((EX x. P(x)) --> (ALL y. Q(y)))

(EX x y. P(x) & Q(x,y)) <-> (EX x. P(x) & (EX y. Q(x,y)))

(EX y. ALL x. P(x) --> Q(x,y)) --> (ALL x. P(x) --> (EX y. Q(x,y)))
```

Summary of the tactics (which belong to structure `Int_Prover`):

```
mp_tac: int -> tactic
pc_tac: int -> tactic
safestep_tac: int -> tactic
safe_tac: int -> tactic
step_tac: int -> tactic
```

2.1.2 Classical logic

The classical theory has the ML identifier `cla_thy`. It consists of intuitionistic logic plus the rule

$$\frac{\begin{array}{c}[\neg P]\\ P\end{array}}{P}$$

Natural deduction in classical logic is not really all that natural. Derived rules such as the following help [8, pages 46–49].

$$\frac{\begin{array}{c}[\neg Q]\\ P\end{array}}{P \vee Q} \qquad \frac{P \supset Q \qquad \begin{array}{c}[\neg P]\\ R\end{array} \qquad \begin{array}{c}[Q]\\ R\end{array}}{R} \qquad \frac{\neg P \qquad \begin{array}{c}[\neg Q]\\ P\end{array}}{Q}$$

The first two exploit the classical equivalence of $P \supset Q$ and $\neg P \vee Q$. The third, or swap rule, is typically applied to an assumption $\neg P$. If P is a complex formula then the resulting subgoal is P, which can be broken up using introduction rules. The classical proof procedures combine the swap rule with each of the introduction rules, so that it is only applied for this purpose. This simulates the sequent calculus LK [13], where a sequent $P_1, \ldots, P_m \vdash Q_1, \ldots, Q_n$ can have multiple formulae on the right. In Isabelle, at least, this strange system seems to work better than LK itself.

The functor `ProverFun` makes theorem-proving tactics for arbitrary collections of natural deduction rules. It could be applied, for example, to some introduction and elimination rules for the constants of set theory. At present it is applied only once, to the basic rules of classical logic. The main tactics so defined are `fast_tac`, `best_tac`, and `comp_tac`; they belong to structure `Pc`.

The tactic `onestep_tac` performs one safe step. It is the classical counterpart of `Int_Prover.safestep_tac`.

The tactic `step_tac` performs one step, something like `Int_Prover.step_tac`. Calling `step_tac thms i` tries to reduce subgoal i by safe rules, or else by unsafe rules. The rules given as `thms` are treated like unsafe introduction rules. The tactic may produce multiple outcomes.

The main theorem-proving tactic is `fast_tac`. Calling `fast_tac thms i` tries to solve subgoal i by backtracking, with `step_tac thms i` as the step tactic.

A slower but more powerful tactic is `best_tac`. Calling `best_tac thms` tries to solve *all* subgoals by best-first search with `step_tac`.

A yet slower but 'almost' complete tactic is `comp_tac`. Calling `comp_tac thms` tries to solve *all* subgoals by best-first search. The step tactic is rather judicious about expanding quantifiers and so forth.

The following are proved, respectively, by `fast_tac`, `best_tac`, and `comp_tac`. They are all due to Pelletier [11].

```
(EX y. ALL x. J(y,x) <-> ~J(x,x))
       -->  ~ (ALL x. EX y. ALL z. J(z,y) <-> ~ J(z,x))

(ALL x. P(a) & (P(x)-->P(b))-->P(c))  <->
    (ALL x. (~P(a) | P(x) | P(c)) & (~P(a) | ~P(b) | P(c)))

(EX x. P-->Q(x)) & (EX x. Q(x)-->P) --> (EX x. P<->Q(x))
```

Summary of the tactics (which belong to structure `Pc`):

```
best_tac : thm list -> tactic
comp_tac : thm list -> tactic
fast_tac : thm list -> int -> tactic
onestep_tac : int -> tactic
step_tac : thm list -> int -> tactic
```

2.1.3 An intuitionistic example

Here is a session similar to one in my book on LCF [8, pages 222–3]. LCF users may want to compare its treatment of quantifiers with Isabelle's.

The proof begins by entering the goal in intuitionistic logic, then applying the rule $\supset$I.

```
> goal Int_Rule.thy
     "(EX y. ALL x. Q(x,y)) -->  (ALL x. EX y. Q(x,y))";
Level 0
(EX y. ALL x. Q(x,y)) --> (ALL x. EX y. Q(x,y))
 1. (EX y. ALL x. Q(x,y)) --> (ALL x. EX y. Q(x,y))
> by (resolve_tac [imp_intr] 1);
Level 1
(EX y. ALL x. Q(x,y)) --> (ALL x. EX y. Q(x,y))
 1. EX y. ALL x. Q(x,y) ==> ALL x. EX y. Q(x,y)
```

In this example we will never have more than one subgoal. Applying $\supset$I changed `-->` into `==>`, so $\exists y\,.\,\forall x\,.\,Q(x,y)$ is now an assumption. We have the choice of eliminating the $\exists x.$ or introducing the $\forall x.$. Let us apply $\forall$I.

```
> by (resolve_tac [all_intr] 1);
Level 2
(EX y. ALL x. Q(x,y)) --> (ALL x. EX y. Q(x,y))
 1. EX y. ALL x. Q(x,y) ==> (!ka. EX y. Q(ka,y))
```

Applying ∀I replaced the `ALL x` by `!ka`. The universal quantifier changes from object (∀) to meta (⋀). The bound variable is renamed `ka`, and is a *parameter* of the subgoal. We now must choose between ∃I and ∃E. What happens if the wrong rule is chosen?

```
> by (resolve_tac [exists_intr] 1);
Level 3
(EX y. ALL x. Q(x,y)) --> (ALL x. EX y. Q(x,y))
 1. EX y. ALL x. Q(x,y) ==> (!ka. Q(ka,?a2(ka)))
```

The new subgoal 1 contains the function variable `?a2`. Although `ka` is a bound variable, instantiating `?a2` can replace `?a2(ka)` by a term containing `ka`. Now we simplify the assumption $\exists y.\forall x.Q(x,y)$ using elimination rules. To apply ∃E to the assumption, call `eresolve_tac`.

```
> by (eresolve_tac [exists_elim] 1);
Level 4
(EX y. ALL x. Q(x,y)) --> (ALL x. EX y. Q(x,y))
 1. !ka kb. ALL x. Q(x,kb) ==> Q(ka,?a2(ka))
```

This step has produced the parameter `kb` and replaced the assumption by a universally quantified one. The next step is to eliminate that quantifier. But the subgoal is unprovable. There is no way to unify `?a2(ka)` with the bound variable `kb`: assigning `%(x)kb` to `?a2` is illegal.

Using `undo` we can return to where we went wrong, and correct matters. This time we apply ∃E.

```
...
Level 2
(EX y. ALL x. Q(x,y)) --> (ALL x. EX y. Q(x,y))
 1. EX y. ALL x. Q(x,y) ==> (!ka. EX y. Q(ka,y))
> by (eresolve_tac [exists_elim] 1);
Level 3
(EX y. ALL x. Q(x,y)) --> (ALL x. EX y. Q(x,y))
 1. !ka kb. ALL x. Q(x,kb) ==> EX y. Q(ka,y)
```

We now have two parameters and no scheme variables. Parameters should be produced early. Applying ∃I and ∀E will produce two scheme variables.

```
> by (resolve_tac [exists_intr] 1);
Level 4
(EX y. ALL x. Q(x,y)) --> (ALL x. EX y. Q(x,y))
 1. !ka kb. ALL x. Q(x,kb) ==> Q(ka,?a3(ka,kb))
> by (eresolve_tac [all_elim] 1);
Level 5
(EX y. ALL x. Q(x,y)) --> (ALL x. EX y. Q(x,y))
 1. !ka kb. Q(?a4(ka,kb),kb) ==> Q(ka,?a3(ka,kb))
```

The subgoal has variables `?a3` and `?a4` applied to both parameters. The obvious projection functions unify `?a4(ka,kb)` with `ka` and `?a3(ka,kb)` with `kb`.

```
> by (assume_tac 1);
Level 6
(EX y. ALL x. Q(x,y)) --> (ALL x. EX y. Q(x,y))
No subgoals!
```

The theorem was proved in six tactic steps, not counting the abandoned ones. But proof checking is tedious: `pc_tac` proves the theorem in one step.

```
Level 0
(EX y. ALL x. Q(x,y)) --> (ALL x. EX y. Q(x,y))
 1. (EX y. ALL x. Q(x,y)) --> (ALL x. EX y. Q(x,y))
> by (pc_tac 1);
Level 1
(EX y. ALL x. Q(x,y)) --> (ALL x. EX y. Q(x,y))
No subgoals!
```

2.1.4 A classical example

To illustrate classical logic, we shall prove the theorem $\exists y \,.\, \forall x \,.\, P(y) \supset P(x)$. This fails constructively because no y can be exhibited such that $\forall x \,.\, P(y) \supset P(x)$. Classically, $\exists$ does not have this meaning, and the theorem can be proved rather as follows. If there is any y such that $\neg P(y)$, then chose that y; otherwise $\forall x \,.\, P(x)$ is true. Either way the theorem holds. If this proof seems counterintuitive, then perhaps you are an intuitionist.

The formal proof does not conform in any obvious way to the sketch above. The key step is the very first rule, `exists_cintr`, which is the classical $\exists$I rule. This establishes the case analysis.

```
> goal cla_thy "EX y. ALL x. P(y)-->P(x)";
Level 0
EX y. ALL x. P(y) --> P(x)
 1. EX y. ALL x. P(y) --> P(x)
> by (resolve_tac [exists_cintr] 1);
Level 1
EX y. ALL x. P(y) --> P(x)
 1. ALL x. ~(ALL y. P(x) --> P(y)) ==> ALL x. P(?a) --> P(x)
```

We now can either exhibit a term `?a` to satisfy the conclusion of subgoal 1, or produce a contradiction from the assumption. The following steps are routine: the

conclusion and assumption are broken down using the obvious rules.

```
> by (resolve_tac [all_intr] 1);
Level 2
EX y. ALL x. P(y) --> P(x)
 1. ALL x. ~(ALL y. P(x) --> P(y)) ==> (!ka. P(?a) --> P(ka))
> by (resolve_tac [imp_intr] 1);
Level 3
EX y. ALL x. P(y) --> P(x)
 1. ALL x. ~(ALL y. P(x) --> P(y)) ==> (!ka. P(?a) ==> P(ka))
> by (eresolve_tac [all_elim] 1);
Level 4
EX y. ALL x. P(y) --> P(x)
 1. !ka. [| P(?a); ~(ALL x. P(?a3(ka)) --> P(x)) |] ==> P(ka)
```

In classical logic, a negated assumption is equivalent to a conclusion. To get this effect we invoke `eresolve_tac` with the swap rule. The current conclusion (`P(ka)`) becomes a negated assumption.

```
> by (eresolve_tac [swap] 1);
Level 5
EX y. ALL x. P(y) --> P(x)
 1. !ka. [| P(?a); ~P(ka) |] ==> ALL x. P(?a3(ka)) --> P(x)
```

Introduction rules analyse the new conclusion of subgoal 1.

```
> by (resolve_tac [all_intr] 1);
Level 6
EX y. ALL x. P(y) --> P(x)
 1. !ka. [| P(?a); ~P(ka) |] ==> !kb. P(?a3(ka)) --> P(kb)
> by (resolve_tac [imp_intr] 1);
Level 7
EX y. ALL x. P(y) --> P(x)
 1. !ka. [| P(?a); ~P(ka) |] ==> !kb. P(?a3(ka)) ==> P(kb)
```

The subgoal now has three assumptions. It may be hard to read: the third assumption is separated from the other two by a meta-quantifier (`!kb.`). We now produce a contradiction between the assumptions `~P(ka)` and `P(?a3(ka))`.

```
> by (eresolve_tac [not_elim] 1);
Level 8
EX y. ALL x. P(y) --> P(x)
 1. !ka. P(?a) ==> (!kb. P(?a3(ka)) ==> P(ka))
> by (assume_tac 1);
Level 9
EX y. ALL x. P(y) --> P(x)
No subgoals!
```

The civilized way of proving this theorem is `comp_tac`. The other classical tactics cannot prove it because they never expand a quantifier more than once.

```
> goal cla_thy "EX y. ALL x. P(y)-->P(x)";
Level 0
EX y. ALL x. P(y) --> P(x)
 1. EX y. ALL x. P(y) --> P(x)
> by (comp_tac []);
size=21
size=31
size=43
Level 1
EX y. ALL x. P(y) --> P(x)
No subgoals!
```

2.2 Classical first-order logic

The theory `LK` implements classical first-order logic through Gentzen's sequent calculus LK (see Gallier [3] or Takeuti [13]). Resembling the method of semantic tableaux, the calculus is well suited for backwards proof. Assertions have the form $\Gamma \vdash \Delta$, where Γ and Δ are lists of formulae. Associative unification, simulated by higher-order unification, handles lists. We easily get powerful proof procedures.

2.2.1 Syntax and rules of inference

Figure 2.4 gives the syntax for `LK`: sequents, quantifiers, and descriptions. The types include formulae and expressions, and a type *sobj* used in the representation of lists. The actual list type, *sequ*, is just $sobj \to sobj$. The infixes are equality and the connectives.

Traditionally Γ and Δ are sequence variables. Since fixed variable declarations are inconvenient, a dollar prefix designates sequence variables. In a sequence, any expression not prefixed by `$` is a formula.

Figure 2.5 presents the rules. The connective $\leftrightarrow$ is defined using & and $\supset$. Figure 2.6 presents derived rules, including rules for $\leftrightarrow$ and weakened quantifier rules. The automatic proof procedures, through these weakened rules, throw away each quantification after a single use. Thus they usually terminate quickly, but are incomplete. The multiple use of a quantifier can be obtained through a duplication rule. The tactic `res_inst_tac` can instantiate the variable `?P` in these rules, specifying the formula to duplicate.

2.2.2 Tactics for the cut rule

The theory `set`, which is built on `LK`, derives many rules through the cut rule. You might ask: what about cut-elimination? The cut rule can be eliminated from proofs of *sequents*, but it is still needed in derivations of *rules*.

symbol	*meta-type*	*precedence*	*description*
`=`	$[term, term] \to form$	Left 50	equality ($=$)
`&`	$[form, form] \to form$	Right 35	conjunction ($\&$)
`|`	$[form, form] \to form$	Right 30	disjunction ($\vee$)
`-->`	$[form, form] \to form$	Right 25	implication ($\supset$)
`<->`	$[form, form] \to form$	Right 25	if and only if ($\leftrightarrow$)

INFIXES

symbol	*meta-type*	*description*
`True`	$sequ \to sequ \to prop$	meta-predicate of truth
`Seqof`	$form \to sequ$	singleton formula list
`~`	$form \to form$	negation ($\neg$)
`Forall`	$(term \to form) \to form$	universal quantifier ($\forall$)
`Exists`	$(term \to form) \to form$	existential quantifier ($\exists$)
`The`	$(term \to form) \to term$	description operator (ϵ)

CONSTANTS

external	*internal*	*standard notation*
Γ `|-` Δ	`True(`Γ`,` Δ`)`	sequent $\Gamma \vdash \Delta$
`ALL` x`.` P	`Forall(`$\lambda x.P$`)`	$\forall x.P$
`EX` x`.` P	`Exists(`$\lambda x.P$`)`	$\exists x.P$
`THE` x`.` P	`The(`$\lambda x.P$`)`	$\epsilon x.P$

TRANSLATIONS

$$
\begin{array}{rcll}
prop & = & sequence_6 \texttt{ |- } sequence_6 & (5) \\
sequence & = & item\{ \texttt{ , } item\} & \\
 & | & empty & \\
item & = & \$identifier & \\
 & | & \$variable & \\
 & | & form & \\
form & = & \texttt{ALL } identifier \{ identifier \} \texttt{ . } form & (10) \\
 & | & \texttt{EX } identifier \{ identifier \} \texttt{ . } form & (10) \\
 & | & \texttt{~ } form_{40} & (40) \\
 & | & others \ldots &
\end{array}
$$

GRAMMAR

Figure 2.4: Syntax of LK

```
basic       $H, P, $G |- $E, P, $F
thin_right  $H |- $E, $F ==> $H |- $E, P, $F
thin_left   $H, $G |- $E ==> $H, P, $G |- $E
cut         [| $H |- $E, P;  $H, P |- $E |] ==> $H |- $E
```

Structural rules

```
conj_right  [| $H|- $E, P, $F;  $H|- $E, Q, $F |] ==> $H|- $E, P&Q, $F
conj_left   $H, P, Q, $G |- $E ==> $H, P & Q, $G |- $E

disj_right  $H |- $E, P, Q, $F ==> $H |- $E, P|Q, $F
disj_left   [| $H, P, $G |- $E;  $H, Q, $G |- $E |] ==> $H, P|Q, $G |- $E

imp_right   $H, P |- $E, Q, $F ==> $H |- $E, P-->Q, $
imp_left    [| $H,$G |- $E,P;  $H, Q, $G |- $E |] ==> $H, P-->Q, $G |- $E

not_right   $H, P |- $E, $F ==> $H |- $E, ~P, $F
not_left    $H, $G |- $E, P ==> $H, ~P, $G |- $E

iff_def     P<->Q == (P-->Q) & (Q-->P)
```

Propositional rules

```
all_right   (!x.$H|- $E, P(x), $F) ==> $H|- $E, ALL x.P(x), $F
all_left    $H, P(a), $G, ALL x.P(x) |- $E ==> $H, ALL x.P(x), $G|- $E

exists_right $H|- $E, P(a), $F, EX x.P(x) ==> $H|- $E, EX x.P(x), $F
exists_left  (!x.$H, P(x), $G|- $E) ==> $H, EX x.P(x), $G|- $E
```

Quantifier rules

Figure 2.5: Meta-axioms for the calculus LK

```
duplicate_right  $H |- $E, P, $F, P ==> $H |- $E, P, $F
duplicate_left   $H, P, $G, P |- $E ==> $H, P, $G |- $E

iff_right [| $H,P|- $E,Q,$F; $H,Q|- $E,P,$F |] ==> $H|- $E, P<->Q, $F
iff_left  [| $H,$G|- $E,P,Q; $H,Q,P,$G|- $E |] ==> $H, P<->Q, $G|- $E

all_left_thin      $H, P(a), $G |- $E ==> $H, ALL x.P(x), $G |- $E
exists_right_thin  $H |- $E, P(a), $F ==> $H |- $E, EX x.P(x), $F
```

Figure 2.6: Derived rules for LK

For example, there is a trivial cut-free proof of the sequent $P \,\&\, Q \vdash Q \,\&\, P$. Noting this, we might want to derive a rule for swapping the conjuncts in a right-hand formula:

$$\frac{\Gamma \vdash \Delta, P \,\&\, Q}{\Gamma \vdash \Delta, Q \,\&\, P}$$

The cut rule must be used, for $P \,\&\, Q$ is not a subformula of $Q \,\&\, P$.

A closer look at the derivations[1] shows that most cuts directly involve a premise of the rule being derived (a meta-assumption). In a few cases, the cut formula is not part of any premise, but serves as a bridge between the premises and the conclusion. In such proofs, the cut formula is specified by calling an appropriate tactic.

The tactic `cut_tac s i` reads the string s as an LK formula P, and applies the cut rule to subgoal i. The new subgoal i will have P on the right, while the new subgoal $i+1$ will have P on the left.

The tactic `cut_right_tac s i` is similar, but also deletes a formula from the right side of the new subgoal i. It is typically used to replace the right-hand formula by P.

The tactic `cut_left_tac s i` is similar, but also deletes a formula from the left side of the new subgoal $i+1$, replacing the left-hand formula by P.

2.2.3 Proof procedure

The LK proof procedure is less powerful than hand-coded theorem provers but is more natural and flexible. It is not restricted to a fixed set of rules. We may derive new rules and use these to derive others, working with abstract concepts rather than definitions.

Rules are classified into *safe* and *unsafe*. An unsafe rule (typically a weakened quantifier rule) is only used when no safe rule can be. A *pack* is simply a pair whose first component is a list of safe rules, and whose second is a list of unsafe rules. Packs can be joined in an obvious way to allow reasoning with various fragments of the logic and its extensions.

Backtracking over the choice of a safe rule accomplishes nothing: applying them in any order leads to essentially the same result. Backtracking may be necessary

[1] for example on `LK/set/resolve.ML`

over basic sequents when they perform unification. Suppose 0, 1, 2, 3 are constants in the subgoals

$$P(0), P(1), P(2) \vdash P(?a)$$
$$P(0), P(2), P(3) \vdash P(?a)$$
$$P(1), P(3), P(2) \vdash P(?a)$$

The only assignment that satisfies all three subgoals is $?a \mapsto 2$, and this can only be discovered by search.

For clarity, imagine that Isabelle declares the type `pack`.[2]

```
type pack = thm list * thm list;
```

The pack `triv_pack` consists of reflexivity and the basic sequent.

The pack `LK_pack` contains, as safe rules, the propositional rules plus `all_right` and `exists_left`. The unsafe rules are `all_left_thin` and `exists_right_thin`.

Calling `pjoin(pack1,pack2)` combines two packs in the obvious way: the lists of safe rules are concatenated, as are the lists of unsafe rules.

Calling `filseq_resolve_tac rules maxr i` determines which of the `rules` could affect a formula in subgoal `i`. If this number exceeds `maxr` then the tactic fails. Otherwise it behaves like `resolve_tac` (but runs much faster).

The tactic `reresolve_tac rules i` repeatedly applies the `rules` to subgoal i and the resulting subgoals. It keeps track of the number of new subgoals generated so as to only affect subgoal `i`.

The tactic `repeat_goal_tac packs i` applies the safe rules in the packs to a goal and resulting subgoals. If no safe rule is applicable then an unsafe rule is tried. For example, `disj_left` is tried before `all_left_thin`, though the former rule produces two subgoals.

The tactic `safe_goal_tac packs i` applies the safe rules in the packs to a goal and resulting subgoals. It ignores the unsafe rules.

For tracing a proof, `step_tac packs i` applies one rule to subgoal i. It considers the safe rules before unsafe rules.

Tactic `pc_tac i` attacks subgoal i with `triv_pack` and `LK_pack`.

Summary of the above tactics, etc.:

```
cut_left_tac: string -> int -> tactic
cut_right_tac: string -> int -> tactic
cut_tac: string -> int -> tactic
filseq_resolve_tac: thm list -> int -> int -> tactic
LK_pack: pack
pc_tac: int -> tactic
pjoin: pack*pack -> pack
repeat_goal_tac: pack list -> int -> tactic
reresolve_tac: thm list -> int -> tactic
safe_goal_tac: pack list -> int -> tactic
step_tac: pack list -> int -> tactic
triv_pack: pack
```

[2]Type synonyms do not work with ML modules.

2.2.4 Sample proofs

Several of Pelletier's problems [11] are solved by `pc_tac`. Here is Problem 39.

```
> goal LK_Rule.thy "|- ~ (EX x. ALL y. F(x,y) <-> ~F(y,y))";
Level 0
 |- ~(EX x. ALL y. F(x,y) <-> ~F(y,y))
 1.  |- ~(EX x. ALL y. F(x,y) <-> ~F(y,y))
> by (pc_tac 1);
Level 1
 |- ~(EX x. ALL y. F(x,y) <-> ~F(y,y))
No subgoals!
```

Problems 25 and 28 are also solved by `pc_tac`.

```
EX x. P(x),
ALL x. L(x) --> ~ (M(x) & R(x)),
ALL x. P(x) --> (M(x) & L(x)),
(ALL x. P(x)-->Q(x)) | (EX x. P(x)&R(x))    |- EX x. Q(x)&P(x)

ALL x. P(x) --> (ALL x. Q(x)),
(ALL x. Q(x)|R(x)) --> (EX x. Q(x)&S(x)),
(EX x.S(x)) --> (ALL x. L(x) --> M(x))
|- ALL x. P(x) & L(x) --> M(x)
```

Unfortunately, `LK` has no tactic similar to `comp_tac` of `FOL`, for repeated quantifier expansion. Problem 35 requires such a step.

```
> goal LK_Rule.thy   "|- EX x y. P (x,y) -->  (ALL x y.P(x,y))";
Level 0
 |- EX x y. P(x,y) --> (ALL x y. P(x,y))
 1.  |- EX x y. P(x,y) --> (ALL x y. P(x,y))
> by (resolve_tac [exists_right] 1);
Level 1
 |- EX x y. P(x,y) --> (ALL x y. P(x,y))
 1.  |- EX y. P(?a,y) --> (ALL x y. P(x,y)),
        EX u y. P(u,y) --> (ALL x y. P(x,y))
```

The rule `exists_right` has expanded the existential quantifier while leaving a copy unchanged. Now `pc_tab` can finish the proof, but instead let us perform it rule by rule.

The next rule to apply is ∃R. It applies to either of the formulae in subgoal 1; fortunately Isabelle chooses the first occurrence.[3]

```
> by (resolve_tac [exists_right_thin] 1);
Level 2
 |- EX x y. P(x,y) --> (ALL x y. P(x,y))
 1.  |- P(?a,?a1) --> (ALL x y. P(x,y)),
        EX u y. P(u,y) --> (ALL x y. P(x,y))
```

[3]The tactic returns the other occurrence as an additional outcome, which can be reached by backtracking.

The next rule is $\supset$R. Because LK is a sequent calculus, the formula P(?a,?a1) does not become an assumption. Instead, it moves to the left of the turnstile ($\vdash$).

```
> by (resolve_tac [imp_right] 1);
Level 3
 |- EX x y. P(x,y) --> (ALL x y. P(x,y))
 1. P(?a,?a1) |- ALL x y. P(x,y), EX u y. P(u,y) --> (ALL x y. P(x,y))
```

Next the universal quantifiers are stripped. Meanwhile, the existential formula lies dormant.

```
> by (resolve_tac [all_right] 1);
Level 4
 |- EX x y. P(x,y) --> (ALL x y. P(x,y))
 1. !ka. P(?a,?a1)
         |- ALL y. P(ka,y), EX u y. P(u,y) --> (ALL x y. P(x,y))
> by (resolve_tac [all_right] 1);
Level 5
 |- EX x y. P(x,y) --> (ALL x y. P(x,y))
 1. !ka kb. P(?a,?a1) |- P(ka,kb), EX u y. P(u,y) --> (ALL x y. P(x,y))
```

Note that the formulae P(?a,?a1) and P(ka,kb) are not unifiable, so this is not a basic sequent. Instead we strip the existential quantifiers.

```
> by (resolve_tac [exists_right_thin] 1);
Level 6
 |- EX x y. P(x,y) --> (ALL x y. P(x,y))
 1. !ka kb. P(?a,?a1)
            |- P(ka,kb), EX y. P(?a14(ka,kb),y) --> (ALL x y. P(x,y))
> by (resolve_tac [exists_right_thin] 1);
Level 7
 |- EX x y. P(x,y) --> (ALL x y. P(x,y))
 1. !ka kb. P(?a,?a1)
            |- P(ka,kb),
               P(?Ga18(ka,kb),?a15(ka,kb)) --> (ALL x y. P(x,y))
```

Applying $\supset$R again produces a basic sequent, solving the goal.

```
> by (resolve_tac [imp_right] 1);
Level 8
 |- EX x y. P(x,y) --> (ALL x y. P(x,y))
 1. !ka kb. P(?a,?a1), P(?Ga25(ka,kb),?Ga23(ka,kb))
            |- P(ka,kb), ALL x y. P(x,y)
> by (resolve_tac [basic] 1);
Level 9
 |- EX x y. P(x,y) --> (ALL x y. P(x,y))
No subgoals!
```

Chapter 3

Isabelle's Set and Type Theories

The discussion of Isabelle's object-logic continues with its set and type theories. They are all large and complicated, as well as involving subtle mathematical ideas. Each section will only make sense to people who are familiar with the logic in question.

The theories are as follows:

- Zermelo-Fraenkel set theory is a subtheory of LK. The Zermelo-Fraenkel axiom system is an established formulation of set theory.
- Martin-Löf's Constructive Type Theory is partially implemented. Universes are the main omission.
- Higher-order logic is the largest and most recent of the Isabelle logics. It is rather more elaborate than Church's well-known formulation. This is an object-logic and should not be confused with Isabelle's meta-logic, which is also a version of higher-order logic.

Theorem proving in these theories is extremely difficult, as may be imagined, for each encompasses large portions of mathematics. The proof procedures supplied with Isabelle must be regarded as first attempts.

3.1 Zermelo-Fraenkel set theory

The Isabelle theory called `set` implements Zermelo-Fraenkel set theory [12] over the classical sequent calculus `LK`. The theory includes a collection of derived rules that form a sequent calculus of sets. The simplistic sequent calculus proof procedure of `LK` works reasonably for set theory.

3.1.1 Syntax and rules of inference

Figure 3.1 gives the syntax for `set`, which extends `LK` with finite sets, ordered pairs, and comprehension. The constant :: is a 'set cons', for $a :: B = \{a\} \cup B$. Starting from the empty set $\emptyset$, it constructs finite sets in the obvious way:

$$\{a, b, c, d\} \quad = \quad a :: (b :: (c :: (d :: \emptyset)))$$

By the separation axiom, the set `Collect(`A`,`P`)` forms the set of all $x \in A$ that satisfy $P(x)$. By the replacement axiom, the set `Replace(`f`,`A`)` forms the set of all $f(x)$ for $x \in A$. The syntax of `set` can express three kinds of comprehension: separation, replacement, and both together.

Infixes include union and intersection ($A \cup B$ and $A \cap B$), and the subset and membership relations. The 'big union' and 'big intersection' operators ($\bigcup C$ and $\bigcap C$) form the union or intersection of a set of sets; $\bigcup C$ can also be written $\bigcup_{A \in C} A$. Of these operators only 'big union' is primitive.

The language of set theory, as studied by logicians, has no constants. The empty set axiom asserts that some set is empty, not that $\emptyset$ is the empty set; and so on for union, powerset, etc. Since formal proofs in this language would be barbarous, the Isabelle theory declares constants. Some of the constants are primitive (like the empty set), while others are defined (like intersection).

The axioms appear in Figure 3.2. They contain unusual definitions where one *formula* is defined to denote another. The extensionality axiom states that $A = B$ means the same thing as $A \subseteq B \mathbin{\&} B \subseteq A$. The power set axiom states that $A \in \mathrm{Pow}(B)$ means the same thing as $A \subseteq B$. Such definitions are not conservative. The theory also defines abbreviations for ordered pairs, successor, etc.

The axioms can be expressed in many different ways. For example, the rule `equal_members` could be expressed as the formula

$$\forall xyA \,.\, (x = y \mathbin{\&} y \in A) \supset x \in A$$

But applying this axiom would involve several `LK` rules. (Most books on set theory omit this axiom altogether!) The axiom of regularity is expressed in its most useful form: transfinite induction.

The replacement axiom involves the concept of *class function*, which is like a function defined on the entire universe of sets. Examples include the power set operator and the successor operator $\mathrm{succ}(x) = x \cup \{x\}$. In set theory, a function *is* its graph. Since the graph of a class function is 'too big' to be a set, it is represented by a 2-place predicate. The theory `set` assumes that every class function can be expressed by some Isabelle term — possibly involving `LK`'s description operator ('The'). Recent experience gives some evidence in favour of the traditional formulation of the replacement axiom, however.

3.1.2 Derived rules

The theory derives a sequent calculus from the axioms. Figures 3.3–3.4 present most of the rules, which refer to the constants of `set` rather than the logical constants.

A rule named X`_thin` has been weakened. In a typical weakened rule:

- A formula in the conclusion is omitted in the premises to allow repeated application of the rule without looping — but this proof procedure is incomplete.
- Some scheme variables appear in the premises but not in the conclusion. In backwards proof these rules introduce new variables in the subgoals.

symbol	*meta-type*	*precedence*	*description*
‘	$[term, term] \to term$	Left 65	function application
Int	$[term, term] \to term$	Right 60	intersection ($\cap$)
Un	$[term, term] \to term$	Right 55	union ($\cup$)
-	$[term, term] \to term$	Right 55	difference ($-$)
::	$[term, term] \to term$	Right 55	inclusion of an element
<=	$[term, term] \to form$	Right 50	subset ($\subseteq$)
:	$[term, term] \to form$	Right 50	membership ($\in$)

INFIXES

0	*term*	empty set
INF	*term*	infinite set
Pow	$term \to term$	powerset operator
Union	$term \to term$	‘big union’ operator
Inter	$term \to term$	‘big intersection’ operator
Pair	$[term, term] \to term$	pairing operator
succ	$term \to term$	successor operator
Choose	$term \to term$	choice operator
Collect	$[term, term \to form] \to term$	separation operator
Replace	$[term \to term, term] \to term$	replacement operator

CONSTANTS

external	*internal*	*standard notation*
{ a_1, ..., a_n }	a_1::$\cdots$::(a_n::0)	$\{a_1, \ldots, a_n\}$
[x || x:A, P]	Collect(A,$\lambda x.P$)	$\{x \in A \mid P\}$
[b || x:A]	Replace($\lambda x.b$,A)	$\{b \mid x \in A\}$
[b || x:A,$P(x)$]	Replace($\lambda x.b$,Collect(A,$\lambda x.P$))	$\{b \mid x \in A \,\&\, P\}$

NOTATION

term = < *term* , *term* >
| [*identifier* || *identifier* : *term* , *form*]
| [*term* || *identifier* : *term*]
| [*term* || *identifier* : *term* , *form*]
| *others* ...

GRAMMAR

Figure 3.1: Syntax of set

```
null_left       $H, a : 0, $G |- $E
setcons_def     a: (b::B)  ==  a=b | a:B
Pair_def        <a,b>  ==  { {a}, {a,b} }
```

Empty Set, Finite Sets, and Ordered Pairs

```
subset_def      A<=B  ==  ALL x. x:A --> x:B
equal_members
    [| $H |- $E, a=b, $F;  $H |- $E, b:A, $F |] ==>  $H |- $E, a:A, $F
ext_def         A=B   ==  A<=B & B<=A
```

Subsets, Equality, Extensionality

```
Pow_def         A: Pow(B)  ==  A<=B
Collect_def     a: Collect(A,P)  ==  a:A & P(a)
Replace_def     c: Replace(f,B)  == EXISTS a. a:B & c=f(a)
```

Power set, Separation, Replacement

```
Union_def    A: Union(C)  ==  (EX B. A:B  &  B:C)
Un_def       a Un b   ==  Union({a,b})
Inter_def    Inter(C) ==  [ x || x: Union(C), ALL y. y:C --> x:y ]
Int_def      a Int b  ==  [ x || x:a, x:b ]
Diff_def     a-b      ==  [ x || x:a, ~(x:b) ]
```

Union, Intersection, Difference

```
succ_def         succ(a)  ==  a Un {a}
INF_right_0      $H |- $E, 0:INF, $F
INF_right_succ   $H |- $E, a:INF --> succ(a):INF, $F

Choose       $H, A=0 |- $E, $F ==> $H |- $E, Choose(A) : A, $F
induction    (!u. $H, ALL v. v:u --> P(v) |- $E, P(u), $F) ==>
             $H |- $E, P(a), $F
```

Infinity, Choice, Transfinite Induction

Figure 3.2: Meta-axioms for `set`

```
null_right      $H |- $E, $F ==> $H |- $E, a:0, $F
setcons_right   $H |- $E, a=b, a:B, $F ==> $H |- $E, a : (b::B), $F
setcons_left    [| $H, a=b, $G |- $E;  $H, a:B, $G |- $E |] ==>
                $H, a:(b::B), $G |- $E

subset_right     (!x.$H, x:A |- $E, x:B, $F) ==> $H|- $E, A<=B, $F
subset_left_thin [| $H, $G |- $E, c:A;  $H, c:B, $G |- $E |] ==>
                 $H, A<=B, $G |- $E

equal_right   [| $H |- $E, A<=B, $F;  $H |- $E, B<=A, $F |] ==>
              $H |- $E, A=B, $F
equal_left_s  $H, A<=B, B<=A, $G |- $E ==> $H, A=B, $G |- $E
eqext_left_thin [| $H,$G|- $E, c:A, c:B;  $H, c:B, c:A, $G|- $E |] ==>
                $H, A=B, $G |- $
eqmem_left_thin [| $H,$G|- $E, a:c, b:c;  $H,$G, b:c, a:c |- $E |] ==>
                $H, a=b, $G |- $E
```

Finite Sets, Subsets, Equality, Extensionality

```
Union_right_thin [| $H |- $E, A:B, $F;  $H |- $E, B:C, $F |] ==>
                 $H |- $E, A : Union(C), $F
Union_left  (!x.$H, A:x, x:C, $G |- $E) ==> $H, A:Union(C), $G |- $E
Un_right    $H |- $E, $F, c:A, c:B ==> $H |- $E, c : A Un B, $F
Un_left     [| $H, c:A, $G|- $E;  $H, c:B, $G |- $E |] ==>
            $H, c: A Un B, $G|- $E

Inter_right (!x.$H, x:C |- $E, A:x, $F) ==> $H, C<=0 |- $E, $F ==>
            $H |- $E, A: Inter(C), $F
Inter_left_thin [| $H, A:B, $G |- $E;  $H, $G |- $E, B:C |] ==>
                $H, A: Inter(C), $G |- $E
Int_right  [| $H|- $E, c:A, $F;  $H|- $E, c:B, $F |] ==>
           $H|- $E, c: A Int B, $F
Int_left   $H, c:A, c:B, $G |- $E ==> $H, c : A Int B, $G |- $E

Diff_right [| $H|- $E, c:A, $F;  $H, c:B|- $E,$F |] ==> $H|- $E, c:A-B, $F
Diff_left  $H, c:A, $G |- $E, c:B ==> $H, c: A-B, $G |- $E
```

Union, Intersection, Difference

Figure 3.3: The derived sequent calculus for `set`

```
Pow_right $H |- $E, A<=B, $F ==> $H |- $E, A : Pow(B), $F
Pow_left  $H, A<=B, $G |- $E ==> $H, A : Pow(B), $G |- $E

Collect_right [| $H |- $E, a:A, $F;  $H |- $E, P(a), $F |] ==>
              $H |- $E, a : Collect(A,P), $F
Collect_left  $H, a: A, P(a), $G |- $E ==> $H, a: Collect(A,P), $G |- $E

Replace_right_thin [| $H |- $E, a:B, $F;  $H |- $E, c=f(a), $F |] ==>
                    $H |- $E, c : Replace(f,B), $F
Replace_left   (!x.$H, x:B, c=f(x), $G|- $E) ==>
               $H, c: Replace(f,B), $G|- $E
```

Power set, Separation, Replacement

Figure 3.4: The derived sequent calculus for `set` (continued)

Recall that a rule is called *unsafe* if it can reduce a provable goal to unprovable subgoals. The rule `subset_left_thin` uses the fact $A \subseteq B$ to reason, 'for any c, if $c \in A$ then $c \in B$.' It reduces $A \subseteq B \vdash A \subseteq B$, which is obviously valid, to the subgoals $\vdash A \subseteq B, ?c \in A$ and $?c \in B \vdash A \subseteq B$. These are not valid: if $A = \{2\}$, $B = \{1\}$, and $?c = 1$ then both subgoals are false.

A safe variant of the rule would reduce $A \subseteq B \vdash A \subseteq B$ to the subgoals $A \subseteq B \vdash A \subseteq B, c \in A$ and $A \subseteq B, c \in B \vdash A \subseteq B$, both trivially valid. In contrast, `subset_right` is safe: if the conclusion is true, then $A \subseteq B$, and thus the premise is also true: if $x \in A$ then $x \in B$ for arbitrary x.

The rules for big intersection are not completely analogous to those for big union because of the empty set. Clearly $\bigcup(\emptyset)$ equals $\emptyset$. We might expect $\bigcap(\emptyset)$ to equal the universal set, but there is none in ZF set theory. The definition happens to make $\bigcap(\emptyset)$ equal $\emptyset$; we may as well regard it as undefined. The rule `Inter_right` says $A \in \bigcap(C)$ if C is non-empty and $x \in C$ implies $A \in x$ for every x.

Another collection of derived rules considers the set operators under the subset relation, as in $A \cup B \subseteq C$. These are not shown here.

3.1.3 Tactics

The `set` theorem prover uses the 'pack' techniques of `LK`. The set theory sequent calculus can prove many theorems about sets without using logical connectives. Such proofs are shorter because they do not involve the rules of `LK`. Combining packs gives various collections of rules to `repeat_goal_tac`. Equality reasoning is difficult at present, although the extensionality rules can do a surprising amount with equalities. Rewriting ought to be provided.

The sequent rules for set theory belong to `set_pack`.

The extensionality rules `equal_right` and `eqext_left_thin` belong to `ext_pack`. These rules, which treat $A = B$ like $A \subseteq B \,\&\, B \subseteq A$, are not in

`set_pack` because they can be costly.

Calling `set_tac i` tackles subgoal i with `triv_pack` and `set_pack`. Calling `setpc_tac i` tackles subgoal i with both set theory and predicate calculus rules: `triv_pack`, `set_pack`, and `LK_pack`.

There are single-step tactics for debugging. Calling `set_step_tac i` applies to subgoal i a rule from `triv_pack` or `set_pack`. Calling `setpc_step_tac i` applies to subgoal i a rule from `triv_pack`, `set_pack`, or `LK_pack`.

Summary of the above tactics, etc.:

```
ext_pack: pack
set_pack: pack
set_step_tac: int -> tactic
set_tac: int -> tactic
setpc_step_tac: int -> tactic
setpc_tac: int -> tactic
```

3.1.4 Examples

For a simple example about intersections and powersets, let us prove half of the equation $Pow(A) \cap Pow(B) \subseteq Pow(A \cap B)$. Compared with first-order logic, set theory seems to involve a maze of rules. This proof will go straight to the solution. It might be instructive to try the proof yourself choosing other rules, observing where you end up.

We enter the goal and make the first (forced) step, namely $\subseteq$ on the right.

```
> goal Set_Rule.thy "|- Pow(A Int B) <= Pow(A) Int Pow(B)";
Level 0
 |- Pow(A Int B) <= Pow(A) Int Pow(B)
 1.  |- Pow(A Int B) <= Pow(A) Int Pow(B)
> by (resolve_tac [subset_right] 1);
Level 1
 |- Pow(A Int B) <= Pow(A) Int Pow(B)
 1. !ka. ka : Pow(A Int B) |- ka : Pow(A) Int Pow(B)
```

Note the parameter `ka`: the subgoal says every element (`ka`) of $Pow(A \cap B)$ is also an element of $Pow(A) \cap Pow(B)$. Next, the `Pow` on the left is replaced by an instance of the subset relation (`<=`).

```
> by (resolve_tac [Pow_left] 1);
Level 2
 |- Pow(A Int B) <= Pow(A) Int Pow(B)
 1. !ka. ka <= A Int B |- ka : Pow(A) Int Pow(B)
```

Intersection being like conjunction, the intersection on the right produces two subgoals. Henceforth we work on subgoal 1.

```
> by (resolve_tac [Int_right] 1);
Level 3
 |- Pow(A Int B) <= Pow(A) Int Pow(B)
 1. !ka. ka <= A Int B |- ka : Pow(A)
 2. !ka. ka <= A Int B |- ka : Pow(B)
```

The Pow is converted to a subset, which is then reduced.

```
> by (resolve_tac [Pow_right] 1);
Level 4
 |- Pow(A Int B) <= Pow(A) Int Pow(B)
 1. !ka. ka <= A Int B |- ka <= A
 2. !ka. ka <= A Int B |- ka : Pow(B)
> by (resolve_tac [subset_right] 1);
Level 5
 |- Pow(A Int B) <= Pow(A) Int Pow(B)
 1. !ka kb. ka <= A Int B, kb : ka |- kb : A
 2. !ka. ka <= A Int B |- ka : Pow(B)
val it = () : unit
```

Subgoal 1 has two parameters, ka and kb. We now reduce the subset relation on the left, asking for an element of ka and asserting that it is also an element of $A \cap B$. This element may depend upon ka and kb.

```
> by (resolve_tac [subset_left_thin] 1);
Level 6
 |- Pow(A Int B) <= Pow(A) Int Pow(B)
 1. !ka kb. kb : ka |- kb : A, ?c5(ka,kb) : ka
 2. !ka kb. ?c5(ka,kb) : A Int B, kb : ka |- kb : A
 3. !ka. ka <= A Int B |- ka : Pow(B)
```

Subgoal 1 now has two formulae on the right. Recall that it suffices to prove either of them assuming all those on the left. Resolution with a basic sequent instantiates the term ?c5(ka,kb) to kb, solving this subgoal.

```
> by (resolve_tac [basic] 1);
Level 7
 |- Pow(A Int B) <= Pow(A) Int Pow(B)
 1. !ka kb. kb : A Int B, kb : ka |- kb : A
 2. !ka. ka <= A Int B |- ka : Pow(B)
```

Note that both occurrences of ?c5(ka,kb) have been instantiated. Showing that every element of $A \cap B$ is also an element of A is now a simple matter.

```
> by (resolve_tac [Int_left] 1);
Level 8
 |- Pow(A Int B) <= Pow(A) Int Pow(B)
 1. !ka kb. kb : A, kb : B, kb : ka |- kb : A
 2. !ka. ka <= A Int B |- ka : Pow(B)
> by (resolve_tac [basic] 1);
Level 9
 |- Pow(A Int B) <= Pow(A) Int Pow(B)
 1. !ka. ka <= A Int B |- ka : Pow(B)
```

The remaining subgoal is like the one we tackled back in Level 3. Instead of repeating those steps, let us go for the automatic tactic.

```
> by (set_tac 1);
Level 10
 |- Pow(A Int B) <= Pow(A) Int Pow(B)
No subgoals!
```

Calling `set_tac` could have been the first and only step of this proof.

For another example, we prove that big union is monotonic: $A \subseteq B$ implies $\bigcup(A) \subseteq \bigcup(B)$. As above, the first inference reduces the $\subseteq$ on the right.

```
> goal Set_Rule.thy "A<=B |- Union(A) <= Union(B)";
Level 0
A <= B |- Union(A) <= Union(B)
 1. A <= B |- Union(A) <= Union(B)
> by (resolve_tac [subset_right] 1);
Level 1
A <= B |- Union(A) <= Union(B)
 1. !ka. A <= B, ka : Union(A) |- ka : Union(B)
```

Big union is like an existential quantifier, so the occurrence on the left must be reduced before that on the right.

```
> by (resolve_tac [Union_left] 1);
Level 2
A <= B |- Union(A) <= Union(B)
 1. !ka kb. A <= B, ka : kb, kb : A |- ka : Union(B)
```

The next two steps infer that since $A \subseteq B$ and $kb \in A$, it follows that $kb \in B$.

```
> by (resolve_tac [subset_left_thin] 1);
Level 3
A <= B |- Union(A) <= Union(B)
 1. !ka kb. ka : kb, kb : A |- ka : Union(B), ?c2(ka,kb) : A
 2. !ka kb. ?c2(ka,kb) : B, ka : kb, kb : A |- ka : Union(B)
> by (resolve_tac [basic] 1);
Level 4
A <= B |- Union(A) <= Union(B)
 1. !ka kb. kb : B, ka : kb, kb : A |- ka : Union(B)
```

The only possible step is to reduce the big union operator on the right.

```
> by (resolve_tac [Union_right_thin] 1);
Level 5
A <= B |- Union(A) <= Union(B)
 1. !ka kb. kb : B, ka : kb, kb : A |- ka : ?B5(ka,kb)
 2. !ka kb. kb : B, ka : kb, kb : A |- ?B5(ka,kb) : B
```

To show $ka \in \bigcup(B)$ it suffices to find some element of B, for the moment called `?B5(ka,kb)`, containing ka as an element. These two subgoals are proved by resolution with the basic sequent.

```
> by (resolve_tac [basic] 1);
Level 6
A <= B |- Union(A) <= Union(B)
 1. !ka kb. kb : B, ka : kb, kb : A |- kb : B
> by (resolve_tac [basic] 1);
Level 7
A <= B |- Union(A) <= Union(B)
No subgoals!
```

Again, calling `set_tac` could have proved this theorem immediately.

Proofs about 'big intersection' tend to be complicated because $\bigcap$ is ill-behaved on the empty set. One theorem which has a difficult proof is

```
~(C<=0) |- Inter([ A(x) Int B(x) || x:C ]) =
           Inter([ A(x) || x:C ])  Int  Inter([ B(x) || x:C ]
```

In traditional notation this is

$$C \neq \emptyset \supset \bigcap_{x \in C} (A(x) \cap B(x)) = (\bigcap_{x \in C} A(x)) \cap (\bigcap_{x \in C} B(x))$$

Another large example justifies the standard definition of pairing, $\langle a, b \rangle = \{\{a\}, \{a, b\}\}$. Proving that $\langle a, b \rangle = \langle c, d \rangle$ implies $a = c$ and $b = d$ is easier said than done!

3.2 Constructive Type Theory

Isabelle was first written for the Intuitionistic Theory of Types, a formal system of great complexity.[1] The original formulation was a kind of sequent calculus. It included rules for building the context, namely variable bindings with their types. A typical judgement was

$$a(x_1, \ldots, x_n) \in A(x_1, \ldots, x_n)\ [x_1 \in A_1, x_2 \in A_2(x_1), \ldots, x_n \in A_n(x_1, \ldots, x_{n-1})]$$

In early versions of Isabelle, the object-logic `ITT` implemented this sequent calculus. It was not satisfactory because assumptions like 'suppose A is a type' or 'suppose $B(x)$ is a type for all x in A' could not be formalized. Now Isabelle permits a natural deduction formulation of Type Theory. The judgement above is expressed using $\bigwedge$ and $\Longrightarrow$:

$$\bigwedge x_1 \,.\, x_1 \in A_1 \Longrightarrow \bigwedge x_2 \,.\, x_2 \in A_2(x_1) \Longrightarrow \cdots \bigwedge x_n \,.\, x_n \in A_n(x_1, \ldots, x_{n-1)} \Longrightarrow a(x_1, \ldots, x_n) \in A(x_1, \ldots, x_n)$$

[1] This section presupposes knowledge of Martin-Löf [6].

Assumptions can use all the judgement forms, for instance to express that B is a family of types over A:

$$\bigwedge x\,.\,x \in A \Longrightarrow B(x)\ \mathrm{type}$$

This Isabelle logic is called `CTT` (Constructive Type Theory) to distinguish it from its obsolete predecessor. To justify the `CTT` formulation it is probably best to appeal directly to the semantic explanations of the rules [6], rather than to the rules themselves. The order of assumptions no longer matters, unlike in standard Type Theory. Frankly I am not sure how faithfully `CTT` reflects Martin-Löf's semantics; but many researchers using Type Theory do not bother with such considerations.

All of Type Theory is supported apart from list types, well ordering types, and universes. Universes could be introduced *à la Tarski*, adding new constants as names for types. The formulation *à la Russell*, where types denote themselves, is only possible if we identify the meta-types of `Aterm` and `Atype`.

`CTT` uses the 1982 version of equality, where the judgements $a = b \in A$ and $c \in Eq(A, a, b)$ are interchangeable. Its rewriting tactics prove theorems of the form $a = b \in A$. Under the new equality rules, rewriting tactics would have to prove theorems of the form $c \in Eq(A, a, b)$, where c would be a large construction.

3.2.1 Syntax and rules of inference

There are meta-types of types and expressions. The constants are shown in Figure 3.6. The infixes include the function application operator (sometimes called 'apply'), and the 2-place type operators. Note that meta-level abstraction and application ($\lambda x.b$ and $f(a)$) differ from object-level abstraction and application (`lam` $x\,.\,b$ and b`‘`a)

The empty type is called F and the one-element type is T; other finite sets are built as $T + T + T$, etc. The `CTT` syntax (Figure 3.5) is similar to that used at the University of Gothenburg, Sweden. We can write `SUM y:B. PROD x:A. C(x,y)` instead of

```
Sum(B,%(y)Prod(A,%(x)C(x,y)))
```

Recall that the type $(\Pi x \in A)B$ is abbreviated $A \to B$, and $(\sum x \in A)B$ is abbreviated $A \times B$, provided B does not depend upon x. Isabelle accepts these abbreviations in parsing and uses them whenever possible for printing.

The equality versions of the rules are called *long* versions; the rules describing the computation of eliminators are called *computation* rules. Some rules are reproduced here to illustrate the syntax. Figure 3.7 shows the rules for N. These include `zero_ne_succ`, the fourth Peano axiom ($0 \neq n + 1$) because it cannot be derived without universes [6, page 91]. Figure 3.8 shows the rules for the general product.

The extra judgement `Reduce` is used to implement rewriting. The judgement `Reduce`(a, b, A) holds when $a = b : A$ holds. It also holds when a and b are syntactically identical, even if they are ill-typed, because rule `refl_red` does not verify that a belongs to A. These rules do not give rise to new theorems about the standard judgements — note that the only rule that makes use of `Reduce` is

symbol	*meta-type*	*precedence*	*description*
‘	$[term, term] \to term$	Left 55	function application
*	$[type, type] \to type$	Right 35	product of two types
+	$[type, type] \to type$	Right 30	sum of two types
-->	$[type, type] \to type$	Right 25	function type

INFIXES

external	*internal*	*standard notation*
PROD x:A.B	Prod(A, $\lambda x.B$)	$(\Pi x \in A)B$
A --> B	Prod(A, $\lambda x.B$)	$A \to B$
SUM x:A.B	Sum(A, $\lambda x.B$)	$(\Sigma x \in A)B$
A * B	Sum(A, $\lambda x.B$)	$A \times B$
lam x.b	lambda($\lambda x.b$)	$(\lambda x)b$

TRANSLATIONS

prop	=	$type_{10}$ type	(5)
	|	$type_{10}$ = $type_{10}$	(5)
	|	$term_{10}$: $type_{10}$	(5)
	|	$term_{10}$ = $term_{10}$: $type_{10}$	(5)
type	=	PROD *identifier* : *type* . *type*	(10)
	|	SUM *identifier* : *type* . *type*	(10)
	|	*others* . . .	
term	=	lam *identifier* . *term*	(10)
	|	< *term* , *term* >	
	|	*others* . . .	

GRAMMAR

Figure 3.5: Syntax of CTT

symbol	*meta-type*	*description*
`Type`	$type \rightarrow prop$	judgement form
`Eqtype`	$[type, type] \rightarrow prop$	judgement form
`Elem`	$[term, type] \rightarrow prop$	judgement form
`Eqelem`	$[term, term, type] \rightarrow prop$	judgement form
`Reduce`	$[term, term, type] \rightarrow prop$	extra judgement form
`N`	$type$	natural numbers type
`0`	$term$	constructor
`succ`	$term \rightarrow term$	constructor
`rec`	$[term, term, [term, term] \rightarrow term] \rightarrow term$	eliminator
`Prod`	$[type, term \rightarrow type] \rightarrow type$	general product type
`lambda`	$(term \rightarrow term) \rightarrow term$	constructor
`Sum`	$[type, term \rightarrow type] \rightarrow type$	general sum type
`pair`	$[term, term] \rightarrow term$	constructor
`split`	$[term, [term, term] \rightarrow term] \rightarrow term$	eliminator
`fst snd`	$term \rightarrow term$	projections
`inl inr`	$term \rightarrow term$	constructors for +
`when`	$[term, term \rightarrow term, term \rightarrow term] \rightarrow term$	eliminator for +
`Eq`	$[type, term, term] \rightarrow type$	equality type
`eq`	$term$	constructor
`F`	$type$	empty type
`contr`	$term \rightarrow term$	eliminator
`T`	$type$	singleton type
`tt`	$term$	constructor

Figure 3.6: The constants of `CTT`

```
N_form            N type
N_intr0           0 : N
N_intr_succ       a : N ==> succ(a) : N
N_intr_succ_long  a = b : N ==> succ(a) = succ(b) : N
N_elim
    [| p: N;  a: C(0);
       !u v. [| u: N; v: C(u) |] ==> b(u,v): C(succ(u)) |]
    ==> rec(p,a,b) : C(p)
N_elim_long
    [| p = q : N;  a = c : C(0);
       !u v.[| u:N; v:C(u) |] ==> b(u,v)=d(u,v): C(succ(u)) |]
     ==> rec(p,a,b) = rec(q,c,d) : C(p)
N_comp0
    [| a: C(0);
       !u v. [| u: N; v: C(u) |] ==> b(u,v): C(succ(u)) |]
    ==> rec(0,a,b) = a : C(0)
N_comp_succ
    [| p: N;  a: C(0);
       !u v. [| u: N; v: C(u) |] ==> b(u,v): C(succ(u)) |]
    ==> rec(succ(p),a,b) = b(p, rec(p,a,b)) : C(succ(p))
zero_ne_succ  [| a: N;  0 = succ(a) : N |] ==> 0: F
```

Figure 3.7: Meta-axioms for the type N

```
Prod_form
    [| A type; !w. w:A ==> B(w) type |] ==> Prod(A,B) type
Prod_form_long
    [| A = C;  !w. w:A ==> B(w) = D(w)|] ==> Prod(A,B) = Prod(C,D)
Prod_intr
    [| A type; !w. w:A ==> b(w):B(w)|] ==> lambda(b): Prod(A,B)
Prod_intr_long
    [| A type;  !w. w:A ==> b(w) = c(w) : B(w)|] ==>
    lambda(b) = lambda(c) : Prod(A,B)
Prod_elim
    [| p : Prod(A,B);  a : A |] ==> p`a : B(a)
Prod_elim_long
    [| p=q: Prod(A,B);  a=b : A |] ==> p`a = q`b : B(a)
Prod_comp
    [| a : A;  !w. w:A ==> b(w) : B(w)|] ==> lambda(b)`a = b(a) : B(a)
Prod_comp2
    p : Prod(A,B) ==> (lam x. p`x) = p : Prod(A,B)"),
```

Figure 3.8: Meta-axioms for the product type

```
Plus_form       [| A type;  B type |] ==> A+B type
Plus_form_long  [| A = C;   B = D |]  ==> A+B = C+D
Plus_intr_inl   [| a : A;   B type |] ==> inl(a) : A+B
Plus_intr_inr   [| A type;  b : B |]  ==> inr(b) : A+B
Plus_intr_inl_long
    [| a = c : A;  B type |] ==> inl(a) = inl(c) : A+B
Plus_intr_inr_long
    [| A type;  b = d : B |] ==> inr(b) = inr(d) : A+B
Plus_elim
    [| p: A+B;  !x. x:A ==> c(x): C(inl(x));
                !y. y:B ==> d(y): C(inr(y)) |]
    ==> when(p,c,d) : C(p)
Plus_elim_long
    [| p = q : A+B;  !x. x: A ==> c(x) = e(x) : C(inl(x));
                     !y. y: B ==> d(y) = f(y) : C(inr(y)) |]
    ==> when(p,c,d) = when(q,e,f) : C(p)
Plus_comp_inl
    [| a: A;  !x. x:A ==> c(x): C(inl(x));  !y. y:B ==> d(y): C(inr(y)) |]
    ==> when(inl(a),c,d) = c(a) : C(inl(a))
Plus_comp_inr
    [| b: B;  !x. x:A ==> c(x): C(inl(x));  !y. y:B ==> d(y): C(inr(y)) |]
    ==> when(inr(b),c,d) = d(b) : C(inr(b))
```

The type +

```
refl_red      a = b : A ==> Reduce(a,b)
red_if_equal  a = b : A ==> Reduce(a,b)
trans_red     [| a = b : A;  Reduce(b,c) |] ==> a = c : A
```

The judgement Reduce

```
fst_def         fst(a) == split(a, %(x,y)x)
snd_def         snd(a) == split(a, %(x,y)y)
```

Definitions

Figure 3.9: Other meta-axioms for CTT

```
subst_prod_elim
    [| p: Prod(A,B);  a: A;  !z. z: B(a) ==> c(z): C(z) |]
    ==> c(p`a): C(p`a)
Sum_elim_fst    p : Sum(A,B) ==> fst(p) : A
Sum_elim_snd
    [| p: Sum(A,B);  A type;  !x. x:A ==> B(x) type |]
    ==> snd(p) : B(fst(p))
```

Figure 3.10: Derived rules for CTT

`trans_red`, whose first premise ensures that a and b (and thus c) are well-typed. Figure 3.9 shows the rules for `Reduce` and the definitions of `fst` and `snd`. It also shows the rules for $+$, the sum of two types.

Many proof procedures work by repeatedly resolving certain Type Theory rules against a proof state. The theory defines lists — each with type `thm list` — of related rules.

- `form_rls`: formation rules for the types N, Π, Σ, $+$, Eq, F, and T.
- `form_long_rls`: long formation rules for Π, Σ, $+$, and Eq. (For other types use `refl_type`.)
- `intr_rls`: introduction rules for the types N, Π, Σ, $+$, and T.
- `intr_long_rls`: long introduction rules for N, Π, Σ, and $+$. (For T use `refl_elem`.)
- `elim_rls`: elimination rules for the types N, Π, Σ, $+$, and F. The rules for Eq and T are omitted because they involve no eliminator.
- `elim_long_rls`: long elimination rules for N, Π, Σ, $+$, and F.
- `comp_rls`: computation rules for the types N, Π, Σ, and $+$. Those for Eq and T involve no eliminator.
- `basic_defs`: the definitions shown in Figure 3.9.

3.2.2 Tactics

The Type Theory tactics provide rewriting, type inference, and logical reasoning. Derived rules are shown in Figure 3.10. The rule `subst_prod_elim` is derived from `prod_elim`, and is easier to use in backwards proof. The rules `Sum_elim_fst` and `Sum_elim_snd` express properties of `fst` and `snd`.

Subgoal reordering

Blind application of rules seldom leads to a proof. Many rules, especially elimination rules, create subgoals containing new schematic variables. Such variables unify with anything, causing an undirectional search. The standard tactics `filt_resolve_tac` and `compat_resolve_tac` (Appendix A) can reject ambiguous goals; so does the `CTT` tactic `test_assume_tac`. Used with the tactical `REPEAT_FIRST` they achieve a simple kind of subgoal reordering: inappropriate subgoals are ignored. Try doing some single step proofs, or study the examples below, to see why this is necessary.

Object-level simplification is accomplished through proof, using the `CTT` equality rules and the built-in rewriting functor. The rewrites are the computation rules and the long versions of the other rules. Also used are transitivity and the extra judgement form `Reduce`. Meta-level simplification handles only definitional equality.

Calling `test_assume_tac i`, where subgoal i has the form $a \in A$ and the head of a is not a scheme variable, calls `assume_tac` to solve the subgoal by assumption. All other cases are rejected.

Many `CTT` tactics work by subgoal reordering. The most important are the following, which all have type `thm list->tactic`. They uses built-in rules as well as additional rules given in the argument.

Calling `typechk_tac thms` uses formation, introduction, and elimination rules to check the typing of constructions. It is designed to solve goals like $a \in ?A$ where a is rigid and $?A$ is flexible. Thus it performs type inference using essentially Milner's algorithm, which is expressed in the rules. The tactic can also solve goals of the form A type.

Calling `equal_tac thms` solves goals like $a = b \in A$, where a is rigid, using the long introduction and elimination rules. It is intended for deriving the long rules for defined constants such as the arithmetic operators. The tactic can also perform type checking.

Calling `intr_tac thms` uses introduction rules to break down a type. It is designed for goals like $?a \in A$ where $?a$ is flexible and A rigid. These typically arise when trying to prove a proposition A, expressed as a type.

Calling `rew_tac thms` applies left-to-right rewrite rules. It solves the goal $a = b \in A$ by rewriting a to b. If b is a scheme variable then it is assigned the rewritten form of a. All subgoals are rewritten.

Calling `hyp_rew_tac thms` rewrites each subgoal with the given theorems and also with any rewrite rules present in the assumptions.

Tactics for logical reasoning

Interpreting propositions as types lets `CTT` express statements of intuitionistic logic. The proof procedures of `FOL`, adapted for `CTT`, can prove many such statements automatically.

However, Constructive Type Theory is not just another syntax for first-order logic. A key question: can assumptions be deleted after use? Not every occurrence of a type represents a proposition, and Type Theory assumptions declare variables. In first-order logic, $\vee$-elimination with the assumption $P \vee Q$ creates one subgoal

assuming P and another assuming Q, and $P \vee Q$ can be deleted. In Type Theory, $+$-elimination with the assumption $z \in A + B$ creates one subgoal assuming $x \in A$ and another assuming $y \in B$ (for arbitrary x and y). Deleting $z \in A + B$ may render the subgoals unprovable if other assumptions refer to z. Some people might argue that such subgoals are not even meaningful.

The tactic `mp_tac` performs Modus Ponens among the assumptions. Calling `mp_tac i` searches for assumptions of the form $f \in \Pi(A, B)$ and $a \in A$ in subgoal i. It replaces that subgoal by one with a parameter z where $f \in \Pi(A, B)$ has been replaced by $z \in B(a)$. Unification may take place, selecting any implication whose antecedent is unifiable with another assumption.

The tactic `add_mp_tac` is like `mp_tac` except that the assumption $f \in \Pi(A, B)$ is retained.

Calling `safestep_tac thms i` attacks subgoal i using formation rules and certain other 'safe' rules (`F_elim`, `Prod_intr`, `Sum_elim`, `Plus_elim`), calling `mp_tac` when appropriate. It also uses the theorems `thms`, which are typically premises of the rule being derived.[2]

Calling `safe_tac thms i` tries to solve subgoal i by backtracking, with `safestep_tac thms i` as the step tactic.

Calling `step_tac thms i` tries to reduce subgoal i by `safestep_tac i`, then tries unsafe rules. It may produce multiple outcomes.

Calling `pc_tac thms i` tries to solve subgoal i by backtracking, with `step_tac thms i` as the step tactic. This tactic is for logical reasoning, not type checking.

Summary of the tactics (which belong to structure `CTT_Resolve`):

```
add_mp_tac: int -> tactic
mp_tac: int -> tactic
pc_tac: thm list -> int -> tactic
safestep_tac: thm list -> int -> tactic
safe_tac: thm list -> int -> tactic
step_tac: thm list -> int -> tactic
```

3.2.3 An example of type inference

Type inference involves proving a goal of the form $a \in ?A$, where a is a term and $?A$ is a scheme variable standing for its type. The type, initially unknown, takes shape in the course of the proof. Our example is the predecessor function on the natural numbers.

```
> goal CTT_Rule.thy "lam n. rec(n, 0, %x y.x) : ?A";
Level 0
lam n. rec(n,0,%x y. x) : ?A
 1. lam n. rec(n,0,%x y. x) : ?A
```

[2]The name, borrowed from `FOL`, is even less inappropriate for Type Theory. This tactic performs unsafe steps and is therefore incomplete.

Since the term is a Constructive Type Theory λ-abstraction (not to be confused with a meta-level abstraction), the only rule worth considering is Π-introduction. As a result, `?A` gets instantiated to a product type, though its domain and range are still unknown.

```
> by (resolve_tac [Prod_intr] 1);
Level 1
lam n. rec(n,0,%x y. x) : Prod(?A1,?B1)
 1. ?A1 type
 2. !ka. ka : ?A1 ==> rec(ka,0,%x y. x) : ?B1(ka)
```

Subgoal 1 can be solved in many ways, most of which would invalidate subgoal 2. We therefore tackle the latter subgoal. It asks the type of a term beginning with `rec`, which can be found by N-elimination.

```
> by (eresolve_tac [N_elim] 2);
Level 2
lam n. rec(n,0,%x y. x) : PROD ka:N. ?C2(ka,ka)
 1. N type
 2. !ka. 0 : ?C2(ka,0)
 3. !ka kb kc. [| kb : N; kc : ?C2(ka,kb) |] ==> kb : ?C2(ka,succ(kb))
```

We now know that type `?A1` is the natural numbers. However, let us continue with subgoal 2. What is the type of 0?

```
> by (resolve_tac [N_intr0] 2);
Level 3
lam n. rec(n,0,%x y. x) : N --> N
 1. N type
 2. !ka kb kc. [| kb : N; kc : N |] ==> kb : N
```

The type `?A` is now determined: it is $(\Pi ka : N)N$, which is equivalent to $N \to N$. But we must prove all the subgoals to show that the original term is validly typed. The current subgoal 2 is provable by assumption. The remaining subgoal falls by N-formation.

```
> by (assume_tac 2);
Level 4
lam n. rec(n,0,%x y. x) : N --> N
 1. N type
> by (resolve_tac [N_form] 1);
Level 5
lam n. rec(n,0,%x y. x) : N --> N
No subgoals!
```

Calling `typechk_tac []` can prove this theorem in one step.

To extract the theorem we must call `uresult()`, not `result()`. The latter function insists that the theorem be identical to the goal, while here `?A` has been

instantiated.

```
> result();
result: proved a different theorem
Exception- ERROR raised
> prth(uresult());
lam n. rec(n,0,%x y. x) : N --> N
```

3.2.4 Examples of logical reasoning

Logical reasoning in Type Theory involves proving a goal of the form $?a \in A$, where type A expresses a proposition and $?a$ is a scheme variable standing for its proof term: a value of type A. This term, of course, is initially unknown, as with type inference. It takes shape during the proof.

Our first example expresses, by propositions-as-types, a theorem about quantifiers in a sorted logic:

$$\frac{\exists x \in A \,.\, P(x) \vee Q(x)}{(\exists x \in A \,.\, P(x)) \vee (\exists x \in A \,.\, Q(x))}$$

It can also be seen as the generalization from $\times$ to Σ of a distributive law of Type Theory:

$$\frac{A \times (B + C)}{(A \times B) + (A \times C)}$$

We derive the rule

$$\frac{A \text{ type} \qquad \begin{array}{c}[x \in A] \\ B(x) \text{ type}\end{array} \qquad \begin{array}{c}[x \in A] \\ C(x) \text{ type}\end{array} \qquad p \in \sum_{x \in A} B(x) + C(x)}{?a \in (\textstyle\sum_{x \in A} B(x)) + (\sum_{x \in A} C(x))}$$

To derive a rule, its premises must be bound to an ML variable. The premises are returned by `goal` and given the name `prems`.

```
> val prems = goal CTT_Rule.thy
#     "[| A type;  !x. x:A ==> B(x) type;  !x. x:A ==> C(x) type; \
# \       p : SUM x:A. B(x) + C(x) |] ==> \
# \    ?a : (SUM x:A. B(x)) + (SUM x:A. C(x))";
Level 0
?a : (SUM x:A. B(x)) + (SUM x:A. C(x))
 1. ?a : (SUM x:A. B(x)) + (SUM x:A. C(x))
val prems = [?, ?, ?, ?] : thm list
```

One of the premises involves summation (Σ). Since it is a premise rather than the assumption of a goal, it cannot be found by `eresolve_tac`. We could insert it

by calling `cut_facts_tac prems 1`, but instead let us apply the Σ-elimination rule by calling `resolve_tac`.

```
> by (resolve_tac [Sum_elim] 1);
Level 1
split(?p1,?c1) : (SUM x:A. B(x)) + (SUM x:A. C(x))
 1. ?p1 : Sum(?A1,?B1)
 2. !ka kb. [| ka : ?A1; kb : ?B1(ka) |] ==>
            ?c1(ka,kb) : (SUM x:A. B(x)) + (SUM x:A. C(x))
```

Another call to `resolve_tac`, this time with the premises, solves subgoal 1. The other subgoal has two new parameters. In the main goal, `?a` has been instantiated with a `split` term.

```
> by (resolve_tac prems 1);
Level 2
split(p,?c1) : (SUM x:A. B(x)) + (SUM x:A. C(x))
 1. !ka kb. [| ka : A; kb : B(ka) + C(ka) |] ==>
            ?c1(ka,kb) : (SUM x:A. B(x)) + (SUM x:A. C(x))
```

The assumption $kb \in B(ka) + C(ka)$ is now eliminated, causing a case split and a new parameter. Observe how the main goal appears now.

```
> by (eresolve_tac [Plus_elim] 1);
Level 3
split(p,%ka kb. when(kb,?c2(ka,kb),?d2(ka,kb)))
: (SUM x:A. B(x)) + (SUM x:A. C(x))
 1. !ka kb. ka : A ==>
            (!kc. kc : B(ka) ==>
                  ?c2(ka,kb,kc) : (SUM x:A. B(x)) + (SUM x:A. C(x)))
 2. !ka kb. ka : A ==>
            (!kc. kc : C(ka) ==>
                  ?d2(ka,kb,kc) : (SUM x:A. B(x)) + (SUM x:A. C(x)))
```

To complete the proof object for the main goal, we need to instantiate the terms `?c2(ka,kb,kc)` and `?d2(ka,kb,kc)`. We attack subgoal 1 by introduction of $+$. Since there are assumptions $ka \in A$ and $kc \in B(ka)$, we take the left injection (`inl`).

```
> by (resolve_tac [Plus_intr_inl] 1);
Level 4
split(p,%ka kb. when(kb,%kc. inl(?a3(ka,kb,kc)),?d2(ka,kb)))
: (SUM x:A. B(x)) + (SUM x:A. C(x))
 1. !ka kb. ka : A ==>
            (!kc. kc : B(ka) ==> ?a3(ka,kb,kc) : SUM x:A. B(x))
 2. !ka kb. ka : A ==> (!kc. kc : B(ka) ==> SUM x:A. C(x) type)
 3. !ka kb. ka : A ==>
            (!kc. kc : C(ka) ==>
                  ?d2(ka,kb,kc) : (SUM x:A. B(x)) + (SUM x:A. C(x)))
```

A new subgoal has appeared, to verify that $\sum_{x \in A} C(x)$ is a type. Continuing with subgoal 1, we introduce the Σ:

```
> by (resolve_tac [Sum_intr] 1);
Level 5
split(p,%ka kb. when(kb,%kc. inl(<?a4(ka,kb,kc),?b4(ka,kb,kc)>),
                     ?d2(ka,kb))) : (SUM x:A. B(x)) + (SUM x:A. C(x))
 1. !ka kb. ka : A ==> (!kc. kc : B(ka) ==> ?a4(ka,kb,kc) : A)
 2. !ka kb. ka : A ==>
            (!kc. kc : B(ka) ==> ?b4(ka,kb,kc) : B(?a4(ka,kb,kc)))
 3. !ka kb. ka : A ==> (!kc. kc : B(ka) ==> SUM x:A. C(x) type)
 4. !ka kb. ka : A ==>
            (!kc. kc : C(ka) ==>
                  ?d2(ka,kb,kc) : (SUM x:A. B(x)) + (SUM x:A. C(x)))
```

The two new subgoals (from Σ-introduction) both hold by assumption:

```
> by (assume_tac 1);
Level 6
split(p,%ka kb. when(kb,%kc. inl(<ka,?b4(ka,kb,kc)>),?d2(ka,kb)))
: (SUM x:A. B(x)) + (SUM x:A. C(x))
 1. !ka kb. ka : A ==> (!kc. kc : B(ka) ==> ?b4(ka,kb,kc) : B(ka))
 2. !ka kb. ka : A ==> (!kc. kc : B(ka) ==> SUM x:A. C(x) type)
 3. !ka kb. ka : A ==>
            (!kc. kc : C(ka) ==>
                  ?d2(ka,kb,kc) : (SUM x:A. B(x)) + (SUM x:A. C(x)))
> by (assume_tac 1);
Level 7
split(p,%ka kb. when(kb,%kc. inl(<ka,kc>),?d2(ka,kb)))
: (SUM x:A. B(x)) + (SUM x:A. C(x))
 1. !ka kb. ka : A ==> (!kc. kc : B(ka) ==> SUM x:A. C(x) type)
 2. !ka kb. ka : A ==>
            (!kc. kc : C(ka) ==>
                  ?d2(ka,kb,kc) : (SUM x:A. B(x)) + (SUM x:A. C(x)))
```

Subgoal 1 is just type checking, and yields to `typechk_tac`, provided the premises (the ML variable `prems`) are supplied.

```
> by (typechk_tac prems);
Level 8
split(p,%ka kb. when(kb,%kc. inl(<ka,kc>),?d2(ka,kb)))
: (SUM x:A. B(x)) + (SUM x:A. C(x))
 1. !ka kb. ka : A ==>
            (!kc. kc : C(ka) ==>
                  ?d2(ka,kb,kc) : (SUM x:A. B(x)) + (SUM x:A. C(x)))
```

Yes, this has been tedious. Let us prove the other case by `pc_tac`.

```
> by (pc_tac prems 1);
Level 9
split(p,%ka kb. when(kb,%kc. inl(<ka,kc>),%kc. inr(<ka,kc>)))
: (SUM x:A. B(x)) + (SUM x:A. C(x))
No subgoals!
```

The following proof derives a currying functional in `?a`. Its type includes, as a special case, the type

$$(A \times B \to C) \to (A \to (B \to C))$$

The argument of the functional is a function that maps $z : \Sigma(A, B)$ to $C(z)$; the resulting function maps $x \in A$ and $y \in B(x)$ to $C(\langle x, y\rangle)$. Here B is a family over A while C is a family over $\Sigma(A, B)$.

```
> val prems = goal CTT_Rule.thy
#     "[| A type; !x. x:A ==> B(x) type; \
# \                !z. z: (SUM x:A. B(x)) ==> C(z) type|] \
# \    ==> ?a : (PROD z : (SUM x:A . B(x)) . C(z)) \
# \         --> (PROD x:A . PROD y:B(x) . C(<x,y>))";
Level 0
?a : (PROD z:SUM x:A. B(x). C(z)) --> (PROD x:A. PROD y:B(x). C(<x,y>))
 1. ?a : (PROD z:SUM x:A. B(x). C(z)) -->
         (PROD x:A. PROD y:B(x). C(<x,y>))
val prems = [?, ?, ?] : thm list
```

It is proved in one step: `pc_tac`. Observe how `?a` becomes instantiated.

```
> by (pc_tac prems 1);
Level 1
lam ka. lam kb. lam kc. ka ‘ <kb,kc>
: (PROD z:SUM x:A. B(x). C(z)) --> (PROD x:A. PROD y:B(x). C(<x,y>))
No subgoals!
```

Two examples by Martin-Löf [6] have been performed in Isabelle. The law of $\vee$ elimination [6, page 58] is derived in one step by `pc_tac`. The choice principle [6, page 50] has a rather delicate proof of nine steps. The resulting constructions are equivalent to those published by Martin-Löf.

3.2.5 Arithmetic

The largest example develops elementary arithmetic — the properties of addition, multiplication, subtraction, division, and remainder — culminating in the theorem

$$a \bmod b + (a/b) \times b = a$$

```
local open Syntax_Def
in  val mixfix =
      [ Infixr("#+",    [Aterm,Aterm]--->Aterm, 25),
        Infixr("-",     [Aterm,Aterm]--->Aterm, 25),
        Infixr("|-|",   [Aterm,Aterm]--->Aterm, 25),
        Infixr("#*",    [Aterm,Aterm]--->Aterm, 35),
        Infixr("'/'/",  [Aterm,Aterm]--->Aterm, 35),
        Infixr("'/",    [Aterm,Aterm]--->Aterm, 35)];

    val arith_ext = {logical_types=[],
                     mixfix=mixfix,
                     parse_translation=[],
                     print_translation=[]};

    val arith_const_decs = constants mixfix;
    val arith_syn = Syntax.extend CTT_Syntax.syn arith_ext
end;


val arith_thy = enrich_theory CTT_Rule.thy  "arith"
    ([], arith_const_decs,  arith_syn)

  [ ("add_def",  "a#+b == rec(a, b, %u v.succ(v))"),
    ("diff_def",  "a-b == rec(b, a, %u v.rec(v, 0, %x y.x))"),
    ("absdiff_def",  "a|-|b == (a-b) #+ (b-a)"),
    ("mult_def",  "a#*b == rec(a, 0, %u v. b #+ v)"),
    ("mod_def",  "a//b == rec(a, 0, %u v.   \
  \                     rec(succ(v) |-| b, 0, %x y.succ(v)))"),
    ("quo_def",  "a/b == rec(a, 0, %u v.   \
  \                     rec(succ(u) // b, succ(v), %x y.v))") ];
```

Figure 3.11: Defining the theory of arithmetic

The declaration of `arith_thy`[3] appears in Figure 3.11. It demonstrates how to extend a theory. See Chapter 5 for a complete description of infix and other kinds of syntax definitions.

Axioms define each operator in terms of others. Although here no operator is used before it is defined, Isabelle accepts arbitrary axioms without complaint. Since Type Theory permits only primitive recursion, some of these definitions may be unfamiliar. The difference $a - b$ is computed by computing b predecessors of a; the predecessor function is

```
%(v)rec(v, 0, %(x,y)x)
```

The remainder $a//b$ counts up to a in a cyclic fashion: whenever the count would reach b, the cyclic count returns to zero. Here the absolute difference gives an equality test. The quotient $a//b$ is computed by adding one for every number x such that $0 \leq x \leq a$ and $x//b = 0$.

Lemmas leading up to the main result include commutative, distributive, and associative laws.

```
[| a:N;  b:N |] ==> a #+ b = b #+ a : N
[| a:N;  b:N |] ==> a #* b = b #* a : N
[| a:N;  b:N;  c:N |] ==> (a #+ b) #* c = (a #* c) #+ (b #* c) : N
[| a:N;  b:N;  c:N |] ==> (a #* b) #* c = a #* (b #* c) : N
```

3.3 Higher-order logic

Isabelle's theory `HOL` combines aspects of all the other object-logics. This formulation of higher-order logic includes a type system as rich as that of Constructive Type Theory, logical rules generalizing those of first-order logic, and a theory of classes. It has gone a long way from its origins in Church's version of the Simple Theory of Types, and this development is continuing. The paper discussing Isabelle's version is already becoming out of date. This logic is too large and too preliminary to justify detailed discussion. We shall just consider its syntax and a few of the rules.

Figures 3.12 and 3.13 present the syntax and figure 3.15 the inference rules. Figure 3.14 lists the constants while figure 3.16 gives their definitions. There are two forms of judgement:

- $a : A$ means that the value a has type A
- P means that the formula P is true

The formulae are those of a many-sorted first-order logic. For higher-order reasoning, the reflection functions *term* and *form* map between formulae and terms of type *bool*. Terms include descriptions, pairs, λ-abstractions, and class abstractions. The latter are boolean-valued λ-abstractions that are viewed as sets.

[3]The theory is on `CTT/arith.ML`; sample proofs are on `CTT/ex/arith.ML`

The types include the general products and sums $\prod_{x:A}.B(x)$ and $\sum_{x:A}.B(x)$, generalizing the types $A \to B$ and $A \times B$. Dependent types are possible because there are subtypes of the form $\{x : A \,.\, P(x)\}$, where the formula $P(x)$ may depend upon other variables. At present they are used with well-founded recursion. A type of natural numbers embodies the Axiom of Infinity, permitting the construction of list and tree types. The Isabelle implementation derives disjoint sums (the type $A + B$) and lists.

3.3.1 Tactics

The theory `HOL` provides tactics similar to those for classical first-order logic (see Section 2.1.2). Like those, they exploit a swap rule to get the effect of a sequent calculus. However, they must also perform type checking. They are generated by functor `ProverFun`, which accepts classical natural deduction systems. Module `Pc` works with just the logical connectives, while module `Class` uses class theory also. These use the natural deduction rules derived for the logical constants (which are defined equationally) and for class theory (union, intersection, etc., on boolean-valued functions).

The main tactics are described below. Remember that they belong to the above structures: to refer to `fast_tac` you must write `Pc.fast_tac` or `Class.fast_tac`, and so forth.

The tactic `onestep_tac` performs one safe step. Calling `onestep_tac i` tries to solve subgoal i completely by assumption or absurdity, then tries `mp_tac`, then tries other 'safe' rules.

Calling `step_tac thms i` tries to reduce subgoal i by safe rules, or else by unsafe rules, including those given as `thms`. It may produce multiple outcomes.

Calling `fast_tac thms i` tries to solve subgoal i by backtracking, with `step_tac thms i` as the step tactic. The argument `thms` supplies it with additional rules.

Calling `best_tac thms` tries to solve *all* subgoals by best-first search with `step_tac`.

Calling `comp_tac thms` tries to solve *all* subgoals by best-first search. The step tactic will expand quantifiers repeatedly if necessary.

Summary of the tactics (which belong to structures `Pc` and `Class`):

```
best_tac : thm list -> tactic
comp_tac : thm list -> tactic
fast_tac : thm list -> int -> tactic
onestep_tac : int -> tactic
step_tac : thm list -> int -> tactic
```

Rewriting tactics are also available. Both have type `thm list -> tactic`.

Calling `rew_tac thms` applies left-to-right rewrite rules. It solves the subgoal $a = b \in A$ by rewriting a to b. If b is a scheme variable then it is assigned the rewritten form of a. All subgoals are rewritten.

symbol	*meta-type*	*precedence*	*description*
`<:`	$[term, term] \to form$	Left 55	class membership
`&`	$[form, form] \to form$	Right 35	conjunction ($\&$)
`|`	$[form, form] \to form$	Right 30	disjunction ($\vee$)
`-->`	$[form, form] \to form$	Right 25	implication ($\supset$)
`<->`	$[form, form] \to form$	Right 25	if and only if ($\leftrightarrow$)
‘	$[term, term] \to term$	Left 55	function application
`*`	$[type, type] \to type$	Right 35	product of two types
`+`	$[type, type] \to type$	Right 30	sum of two types
`->`	$[type, type] \to type$	Right 25	function type

INFIXES

external	*internal*	*standard notation*
`ALL` x`:`A`.`P	`Forall(`A`,`$\lambda x.P$`)`	$\forall x : \alpha.P$
`EX` x`:`A`.`P	`Exists(`A`,`$\lambda x.P$`)`	$\exists x : \alpha.P$
`PICK` x`:`A`.`P	`Pick(`A`,`$\lambda x.P$`)`	$\eta x : \alpha.P$
`lam` x`:`A`.`b	`Lambda(`A`, `$\lambda x.b$`)`	abstraction $\lambda x : \alpha.b$
`{|` x`:`A`.`P `|}`	`Lambda(`A`,`$\lambda x.$ `term(`P`))`	class $\{\!\!\{x : \alpha.P\}\!\!\}$
`{` x`:`A`.`P `}`	`subtype(`A`,`$\lambda x.P$`)`	subtype $\{x : \alpha.P\}$
`PROD` x`:`A`.`B	`Pi(`A`, `$\lambda x.B$`)`	product $\prod_{x:A}.B(x)$
`SUM` x`:`A`.`B	`Sigma(`A`, `$\lambda x.B$`)`	sum $\sum_{x:A}.B(x)$
A `-->` B	`Prod(`A`, `$\lambda x.B$`)`	function space $A \to B$
A `*` B	`Sum(`A`, `$\lambda x.B$`)`	binary product $A \times B$

TRANSLATIONS

Figure 3.12: Syntax of `HOL`

$$
\begin{array}{rcll}
prop & = & form & (5) \\
 & | & term_6 \ \texttt{:}\ type_6 & (5) \\
 & & & \\
form & = & \texttt{ALL}\ identifier\ \texttt{:}\ type\ \texttt{.}\ form & (10) \\
 & | & \texttt{EX}\ identifier\ \texttt{:}\ type\ \texttt{.}\ form & (10) \\
 & | & \texttt{\~{}}\ form_{40} & (40) \\
 & | & term_{20}\ \texttt{=}\ term_{20}\ \texttt{:}\ type_{20} & (10) \\
 & | & others\ldots & \\
 & & & \\
term & = & \texttt{PICK}\ identifier\ \texttt{:}\ type\ \texttt{.}\ form & (10) \\
 & | & \texttt{lam}\ identifier\ \texttt{:}\ type\ \texttt{.}\ form & (10) \\
 & | & \{\texttt{|}\ identifier\ \texttt{:}\ type\ \texttt{.}\ form\texttt{|}\} & \\
 & | & others\ldots & \\
 & & & \\
type & = & \texttt{PROD}\ identifier\ \texttt{:}\ type\ \texttt{.}\ type & (10) \\
 & | & \texttt{SUM}\ identifier\ \texttt{:}\ type\ \texttt{.}\ type & (10) \\
 & | & others\ldots &
\end{array}
$$

Figure 3.13: Grammar of HOL

Calling `hyp_rew_tac thms` rewrites each subgoal with the given theorems and also with any rewrite rules present in the assumptions. (At present these may begin with one universal quantifier.)

3.3.2 Examples

The theory HOL comes with a body of derived rules. They range from simple properties of the logical constants and class theory, and simplification lemmas, to Tarski's Theorem and well-founded recursion. Dozens of these are worth studying.

Deriving natural deduction rules for the logical constants from their defining equations requires higher-order reasoning. This also illustrates how to derive rules involving definitions. Let us derive the rules for conjunction.

We enter the desired rule, namely $P, Q/P \,\&\, Q$. Since we are deriving a rule, the list of premises $[P, Q]$ is bound to the ML variable `prems`.

```
> val prems = goal HOL_Rule.thy "[| P;  Q |] ==> P&Q";
Level 0
P & Q
 1. P & Q
```

Next, the subgoals are rewritten by the definition of conjunction:

```
> by (rewrite_goals_tac [conj_def]);
Level 1
P & Q
 1. ALL r:bool. (P --> Q --> form(r)) --> form(r)
```

symbol	*meta-type*	*description*
`Trueprop`	$form \to prop$	truth of a formula
`Elem`	$[term, type] \to prop$	membership in type
`~`	$form \to form$	negation ($\neg$)
`Eq Reduce`	$[term, term, type] \to form$	typed equality
`Pick`	$[type, term \to form] \to term$	description (η)
`Forall Exists`	$[type, term \to form] \to form$	quantifiers
`term`	$form \to term$	reflection to term
`form`	$term \to form$	reflection to form
`void`	$type$	empty type
`unit`	$type$	singleton type
`subtype`	$[type, term \to form] \to type$	subtypes
`bool`	$type$	type of truth values
`True False`	$form$	elements of `bool`
`cond`	$[type, term, term, term] \to term$	conditional
`nat`	$type$	natural numbers type
`0`	$term$	constructor
`Succ`	$term \to term$	constructor
`rec`	$[term, term, [term, term] \to term] \to term$	eliminator
`Pi`	$[type, term \to type] \to type$	general product type
`Lambda`	$(term \to term) \to term$	constructor
`Sigma`	$[type, term \to type] \to type$	general sum type
`Pair`	$[term, term] \to term$	constructor
`split`	$[term, [term, term] \to term] \to term$	eliminator
`fst snd`	$term \to term$	projections
`Inl Inr`	$term \to term$	constructors for +
`when`	$[type, type, type, term, term \to term, term \to term] \to term$	eliminator for +
`subset`	$[type, term, term] \to form$	subset relation
`un`	$[type, term, term] \to term$	union of classes
`int`	$[type, term, term] \to term$	intersection of classes
`union`	$[type, term] \to term$	union of a family
`inter`	$[type, term] \to term$	intersection of family
`pow`	$[type, term] \to term$	powerclass

Figure 3.14: The constants of `HOL`

```
pair_type    [| a: A;  b: B(a) |] ==> <a,b> : SUM x:A.B(x)
sigma_elim   [| p: SUM x:A.B(x);
                !x y.[| x: A; y: B(x) |] ==> Q(<x,y>) |]   ==>  Q(p)
pair_inject  [| <a,b> = <c,d> : (SUM x:A.B(x));
                [| a = c : A;  b = d : B(a) |] ==> R |]   ==>  R

fst_type    p: SUM x:A.B(x) ==> fst(p) : A
snd_type    p: SUM x:A.B(x) ==> snd(p) : B(fst(p))
fst_conv    [| a: A;  b: B(a) |] ==> fst(<a,b>) = a: A
snd_conv    [| a: A;  b: B(a) |] ==> snd(<a,b>) = b: B(a)

split_def    split(p,f) == f(fst(p), snd(p))
```

The type $\sum_{x:A}.B(x)$

```
subtype_intr     [| a: A;  P(a) |] ==> a : {x:A.P(x)}
subtype_elim1    a: {x:A.P(x)} ==> a:A
subtype_elim2    a: {x:A.P(x)} ==> P(a)
```

Subtypes

```
term_type    term(P) : bool
term_conv    p: bool ==> term(form(p)) = p : bool
form_intr    P ==> form(term(P))
form_elim    form(term(P)) ==> P
term_congr   [| P ==> Q;  Q ==> P |] ==> term(P) = term(Q) : bool
```

Reflection

```
Pick_type    EX x:A.P(x) ==> (PICK x:A.P(x)) : A
Pick_congr   [| !x.x: A ==> P(x) <-> Q(x);  EX x:A.P(x) |] ==>
                   (PICK x:A.P(x)) = (PICK x:A.Q(x)) : A
Pick_intr    EX x:A.P(x) ==> P(PICK x:A.P(x))
```

Descriptions

Figure 3.15: The rules of HOL

```
False_def   False == term(ALL p:bool.form(p))
True_def    True == term(ALL p:bool.form(p)-->form(p))
conj_def    P&Q == ALL r:bool. (P-->Q-->form(r)) --> form(r)
disj_def    P|Q == ALL r:bool. (P-->form(r)) --> (Q-->form(r)) --> form(r)
not_def     ~P == (P-->form(False))
iff_def     P<->Q == (P-->Q) & (Q-->P)
exists_def  (EX x:A. P(x)) ==
                 ALL r:bool. (ALL x:A. P(x)-->form(r)) --> form(r)
```

Definitions of the logical constants

```
void_def    void == {p: bool. form(False)}
unit_def    unit == {p: bool. (p=True:bool)}

plus_def  A+B == {w: (A->bool) * (B->bool).
                     (EX x:A. w = Inl(A,B,x) : (A->bool) * (B->bool)) |
                     (EX y:B. w = Inr(A,B,y) : (A->bool) * (B->bool)) }
Inl_def    Inl(A,B,a) == <{|x:A.a=x:A|}, lam y:B.False>
Inr_def    Inr(A,B,b) == <lam x:A.False, {|y:B.b=y:B|}>
when_def   when(A,B,C,p,c,d) == PICK z:C.
              (ALL x:A. (p = Inl(A,B,x) : A+B) --> (z = c(x) : C)) &
              (ALL y:B. (p = Inr(A,B,y) : A+B) --> (z = d(y) : C))
```

Definitions of types

Figure 3.16: Definitions in HOL

The rules ∀I and ⊃I are now applied:

```
> by (resolve_tac [all_intr] 1);
Level 2
P & Q
 1. !ka. ka : bool ==> (P --> Q --> form(ka)) --> form(ka)
> by (resolve_tac [imp_intr] 1);
Level 3
P & Q
 1. !ka. [| ka : bool; P --> Q --> form(ka) |] ==> form(ka)
```

The rule ⊃E is applied twice, to subgoal 1 and then to the resulting subgoal 2, completely decomposing the implication.

```
> by (eresolve_tac [imp_elim] 1);
Level 4
P & Q
 1. !ka. ka : bool ==> P
 2. !ka. [| ka : bool; Q --> form(ka) |] ==> form(ka)
> by (eresolve_tac [imp_elim] 2);
Level 5
P & Q
 1. !ka. ka : bool ==> P
 2. !ka. ka : bool ==> Q
 3. !ka. [| ka : bool; form(ka) |] ==> form(ka)
```

The first two subgoals are proved using the premises, which are supplied to `resolve_tac`.

```
> by (resolve_tac prems 1);
Level 6
P & Q
 1. !ka. ka : bool ==> Q
 2. !ka. [| ka : bool; form(ka) |] ==> form(ka)
> by (resolve_tac prems 1);
Level 7
P & Q
 1. !ka. [| ka : bool; form(ka) |] ==> form(ka)
```

The remaining subgoal holds by assumption. The rule just derived has P and Q as premises.

```
> by (assume_tac 1);
Level 8
P & Q
No subgoals!
> prth (result());
[| ?P; ?Q |] ==> ?P & ?Q
```

The derivation of the elimination rule, $P \,\&\, Q/P$, requires reflection. Again, we bind the list of premises (in this case $[P \,\&\, Q]$) to `prems`.

```
> val prems = goal HOL_Rule.thy "P & Q ==> P";
Level 0
P
 1. P
```

Working with premises that involve defined constants can be tricky. The simplest method uses `cut_facts_tac`. The premises are made into assumptions of subgoal 1, where they can be rewritten by `rewrite_goals_tac`.

```
> by (cut_facts_tac prems 1);
Level 1
P
 1. P & Q ==> P
> by (rewrite_goals_tac [conj_def]);
Level 2
P
 1. ALL r:bool. (P --> Q --> form(r)) --> form(r) ==> P
```

The rewritten assumption can now be broken down by $\forall$E and $\supset$E.

```
> by (eresolve_tac [all_elim] 1);
Level 3
P
 1. (P --> Q --> form(?a1)) --> form(?a1) ==> P
 2. ?a1 : bool
> by (eresolve_tac [imp_elim] 1);
Level 4
P
 1. P --> Q --> form(?a1)
 2. form(?a1) ==> P
 3. ?a1 : bool
```

Applying the `form`-elimination rule to subgoal 2 solves it, instantiating `?a1` to `term(P)`. This is the key step.

```
> by (eresolve_tac [form_elim] 2);
Level 5
P
 1. P --> Q --> form(term(P))
 2. term(P) : bool
```

The implications in subgoal 1 are decomposed by two applications of $\supset$I.

```
> by (resolve_tac [imp_intr] 1);
Level 6
P
 1. P ==> Q --> form(term(P))
 2. term(P) : bool
> by (resolve_tac [imp_intr] 1);
Level 7
P
 1. [| P; Q |] ==> form(term(P))
 2. term(P) : bool
```

The first subgoal holds because `form(term(P))` is equivalent to `P`, while the second is a trivial typing condition.

```
> by (eresolve_tac [form_intr] 1);
Level 8
P
 1. term(P) : bool
> by (resolve_tac [term_type] 1);
Level 9
P
No subgoals!
> prth (result());
?P & ?Q ==> ?P
```

Chapter 4

Developing Tactics, Rules, and Theories

The single-step proofs we have studied until now give a gentle introduction to Isabelle and its logics. As you gain experience with these, however, you are likely to realize that this is not the way to prove anything significant. Single-step proofs are too long. Isabelle's basic tactics can be combined to form sophisticated ones using operators called *tacticals*. The resulting tactics can perform hundreds or thousands of inferences in one invocation. A simple PROLOG interpreter is easily expressed.

This chapter also describes how to extend a logic with new constants and axioms, and how to make definitions. Definitions, if viewed as abbreviations, are meta-level rewrite rules. Expanding abbreviations can lead to gigantic formulae and tedious proofs, however. Deriving rules for the new constants permits shorter proofs.

4.1 Tacticals

Tacticals provide a control structure for tactics, with operators resembling those for regular expressions. The tacticals `THEN`, `ORELSE`, and `REPEAT` form new tactics as follows:

- The tactic *tac*1 `THEN` *tac*2 performs *tac*1 followed by *tac*2: a form of sequencing.
- The tactic *tac*1 `ORELSE` *tac*2 performs *tac*1, or if that fails then *tac*2: a form of choice.
- The tactic `REPEAT` *tac* performs *tac* repeatedly until it fails: a form of repetition.

Users of the systems LCF, HOL, or Nuprl will recognize these tacticals, and other people may find the above descriptions simple enough. But these descriptions are mere sketches of complex operations involving sequences of proof states. To understand this, we must take a closer look at tactics.

4.1.1 The type of tactics

An Isabelle tactic maps a theorem, representing a proof state, to a possibly infinite sequence of proof states:

```
datatype tactic = Tactic of thm -> thm Sequence.seq;
```

Lazy lists are implemented in ML by the type `Sequence.seq`, which is described in Appendix A. You need not know the details of this type to use Isabelle's built-in tacticals, but may find the discussion more comprehensible if you have some programming experience with lazy lists.

Higher-order unification can return an infinite sequence of results, and so can tactics that invoke it: especially `resolve_tac`, `eresolve_tac`, and `assume_tac`. Multiple proof states also arise when these tactics are supplied with more than one rule that is unifiable with the subgoal. When the conclusion of a subgoal is unifiable with more than one of its assumptions, `assume_tac` returns multiple outcomes.

If the above considerations did not apply, the basic tactics might not require sequences of proof states. In LCF, a tactic either returns one result or fails. By returning a possibly infinite sequence of outcomes, Isabelle tactics can perform backtracking search. Furthermore, search strategies can be expressed by tacticals: as operations on tactics.

Chapter 1 describes most of Isabelle's tactics. Two more, `all_tac` and `no_tac`, are of interest because they are identities of tacticals.

The tactic `all_tac` is the identity element of the tactical `THEN`. It maps any proof state ϕ to the 1-element sequence $[\phi]$ containing that state. Thus it succeeds for all states.

The tactic `no_tac` is the identity element of the tacticals `ORELSE` and `APPEND`. It maps any proof state to the empty sequence. Thus it succeeds for no state.

4.1.2 Basic tacticals

Tacticals work by combining sequences of proof states. For each tactic constructed by one of these tacticals, we consider the output sequence that results when it is applied to the proof state ϕ.

The tactic *tac*1 `THEN` *tac*2 computes $tac1(\phi)$, obtaining a sequence of states $[\psi_1, \psi_2, \psi_3, \ldots]$. It applies *tac*2 to each of these states. The output sequence is the concatenation of the sequences $tac2(\psi_1), tac2(\psi_2), tac2(\psi_3), \ldots$. In a common notation for lazy lists, the output is

$$[\theta \mid \psi \in tac1(\phi), \theta \in tac2(\psi)]$$

The tactic *tac*1 `ORELSE` *tac*2 computes $tac1(\phi)$. If this sequence is non-empty then it is returned as the overall result; otherwise $tac2(\phi)$ is returned. This is a deterministic choice, a bit like PROLOG's cut directive. If *tac*1 succeeds then *tac*2 is excluded from consideration.

The tactic *tac*1 `APPEND` *tac*2 returns the concatenation of the sequences $tac1(\phi)$ and $tac2(\phi)$. This is not a fair interleaving, for if $tac1(\phi)$ is infinite then that

sequence is the overall result. A tactical with interleaving could be written, but even APPEND has few uses, for it causes excessive branching during search.

The tactic DETERM *tac* truncates the sequence $tac(\phi)$ to have at most one element. Thus the resulting tactic is deterministic. The tactical DETERM is used to limit the search space.

The tacticals EVERY and FIRST are n-ary versions of THEN and ORELSE. They are given a list of tactics. The tactic EVERY $[tac_1, \ldots, tac_n]$ behaves like

*tac*_1 THEN ... THEN *tac*_*n*

The tactic FIRST $[tac_1, \ldots, tac_n]$ behaves like

*tac*_1 ORELSE ... ORELSE *tac*_*n*

4.1.3 Derived tacticals

The tacticals given above suffice to describe (and to implement) more sophisticated ones. Recursively defined tacticals perform repetition and search. Again, we describe the output sequence returned when the new tactic is applied to the state ϕ.

The tactic TRY *tac* returns $tac(\phi)$ if this sequence is non-empty, and returns the singleton sequence $[\phi]$ otherwise. The definition of the tactical in ML is

```
fun TRY tac = tac ORELSE all_tac;
```

The tactic REPEAT *tac* computes $tac(\phi)$. If this sequence is non-empty then the tactic recursively applies itself to each element, concatenating the results. Otherwise it returns the singleton sequence $[\phi]$. Its ML definition makes use of the function tapply, which applies a tactic to a state:

```
fun REPEAT tac = Tactic (fn state =>
  tapply((tac THEN REPEAT tac) ORELSE all_tac, state));
```

Recursive tacticals must be coded in this awkward fashion to avoid infinite recursion.[1]

If *tac* can return multiple outcomes then so can REPEAT *tac*. Since REPEAT uses ORELSE and not APPEND, it applies *tac* as many times as possible in each outcome. A common idiom is

REPEAT (*tac*1 ORELSE *tac*2)

This repeatedly applies *tac*1 or *tac*2 as many times as possible, giving priority to *tac*1.

The tactic DEPTH_FIRST *satpred tac* performs a depth-first search for satisfactory proof states, which are those for which *satpred* returns true. If $satpred(\phi)$ is true

[1] This more attractive definition loops:
`fun REPEAT tac = (tac THEN REPEAT tac) ORELSE all_tac;`

then the tactic returns $[\phi]$; otherwise the tactic is recursively applied to each element of $tac(\phi)$, and the results concatenated. The ML definition of `DEPTH_FIRST` involves `STATE`, a function permitting inspection of the initial state:

```
fun DEPTH_FIRST satpred tac = STATE (fn state =>
   if satpred state  then  all_tac
                     else  tac THEN DEPTH_FIRST satpred tac);
```

The tactic `BEST_FIRST` $(satpred, costf)$ *tac* returns a sequence of satisfactory proof states by best-first search. Function *satpred* tests whether a state is satisfactory, while *costf* returns its cost. Given a collection of states, the tactic chooses some state ψ having least cost, and computes $tac(\psi)$. If this sequence contains any satisfactory states then they are returned as the overall result of the search. Otherwise ψ is replaced by the elements of $tac(\psi)$, and the search continues. The initial collection of states contains just ϕ, the input state. The cost function is typically the size of the state.

The tactic `BREADTH_FIRST` *satpred tac* performs a breadth-first search for satisfactory proof states, which are those for which *satpred* returns true. For most applications it is too slow.

4.1.4 Tacticals for numbered subgoals

Most tactics, like `resolve_tac`, designate a subgoal by number. Tacticals can exploit the numbering of subgoals. A typical example is

```
REPEAT (resolve_tac rules 1)
```

This repeatedly applies the *rules* to subgoal 1, and can eventually solve all the subgoals. Since `resolve_tac` fails if the numbered subgoal does not exist, the repetition terminates when no subgoals remain.

The tactic `DEPTH_SOLVE_1` *tac* performs a depth-first search for states having fewer subgoals than the initial state.

The function `has_fewer_prems` is a common satisfaction predicate. Testing the number of subgoals, `has_fewer_prems` n ϕ holds just when state ϕ has fewer than n subgoals. In particular, the tactic

```
DEPTH_FIRST (has_fewer_prems 1) tac
```

searches using *tac* for a proof state having no subgoals.

The tacticals `ALLGOALS` and `SOMEGOAL` are applied to a function *tf* of type `int->tactic`, such as `assume_tac` and (by currying) `resolve_tac` *rules*. They apply the tactic $tf(i)$, for $i = n, n-1, \ldots, 1$, to a proof state.[2]

Tactic `ALLGOALS` *tf*, when applied to a state with n subgoals, behaves like

$tf(n)$ `THEN` ... `THEN` $tf(1)$

[2]They go from n down to 1 since $tf(i)$ may add or delete subgoals, altering the numbering above subgoal i.

Tactic `SOMEGOAL` *tf*, when applied to a state with n subgoals, behaves like

tf(n) `ORELSE` ... `ORELSE` *tf*(1)

For instance, `SOMEGOAL assume_tac` solves some subgoal by `assume_tac`, or else fails. The tactical works well with `REPEAT`, repeatedly attacking some subgoal.

The infixes `THEN'`, `ORELSE'`, and `APPEND'` are provided to ease the writing of tactics involving subgoal numbers. They are defined as follows:

```
fun tac1 THEN' tac2 = fn x => tac1 x THEN tac2 x;
fun tac1 ORELSE' tac2 = fn x => tac1 x ORELSE tac2 x;
fun tac1 APPEND' tac2 = fn x => tac1 x APPEND tac2 x;
```

For instance, the tactic

```
SOMEGOAL (resolve_tac thms ORELSE' assume_tac)
```

solves some subgoal by either `resolve_tac thms` or `assume_tac`. This is more readable than

```
SOMEGOAL (fn i => resolve_tac thms i  ORELSE  assume_tac i)
```

Summary of tactics and tacticals:

```
all_tac: tactic
ALLGOALS: (int -> tactic) -> tactic
APPEND: tactic * tactic -> tactic
APPEND': ('a -> tactic) * ('a -> tactic) -> 'a -> tactic
BEST_FIRST: (thm -> bool) * (thm -> int) -> tactic -> tactic
BREADTH_FIRST: (thm -> bool) -> tactic -> tactic
DEPTH_FIRST: (thm -> bool) -> tactic -> tactic
DEPTH_SOLVE_1: tactic -> tactic
DETERM: tactic -> tactic
EVERY: tactic list -> tactic
FIRST: tactic list -> tactic
no_tac: tactic
ORELSE: tactic * tactic -> tactic
ORELSE': ('a -> tactic) * ('a -> tactic) -> 'a -> tactic
REPEAT: tactic -> tactic
SOMEGOAL: (int -> tactic) -> tactic
THEN: tactic * tactic -> tactic
THEN': ('a -> tactic) * ('a -> tactic) -> 'a -> tactic
TRY: tactic -> tactic
```

4.2 Examples with tacticals

By far the most valuable tacticals are `THEN`, `ORELSE`, and `REPEAT`. They express natural control structures, letting us avoid typing repetitious command sequences. For instance, consider proving the first-order theorem

$$\forall x\,.\,P(x) \supset (\forall y\,.\,Q(x,y) \supset Q(x,y)\,\&\,P(x))$$

We could apply the rules ∀I, ⊃I, ∀I, ⊃I, &I in succession (by `resolve_tac`) and then apply `assume_tac` twice: a total of seven commands. You might want to try this proof now.

With tacticals the proof can be accomplished in two steps, or even one.

```
> goal Int_Rule.thy "ALL x. P(x) --> (ALL y. Q(x,y) --> Q(x,y) & P(x))";
Level 0
ALL x. P(x) --> (ALL y. Q(x,y) --> Q(x,y) & P(x))
 1. ALL x. P(x) --> (ALL y. Q(x,y) --> Q(x,y) & P(x))
```

First we repeatedly apply `resolve_tac` to subgoal 1. We supply all the rules that are needed: ∀I, ⊃I, and &I:

```
> by (REPEAT (resolve_tac [all_intr,imp_intr,conj_intr] 1));
Level 1
ALL x. P(x) --> (ALL y. Q(x,y) --> Q(x,y) & P(x))
 1. !ka. P(ka) ==> (!kb. Q(ka,kb) ==> Q(ka,kb))
 2. !ka. P(ka) ==> (!kb. Q(ka,kb) ==> P(ka))
```

Those rules have been applied as much as possible. The two subgoals that result are solved by assumption.

```
> by (REPEAT (assume_tac 1));
Level 2
ALL x. P(x) --> (ALL y. Q(x,y) --> Q(x,y) & P(x))
No subgoals!
```

A single tactic, combining the repetition of `assume_tac` and `resolve_tac`, can prove the theorem in a single command:

```
REPEAT    (assume_tac 1  ORELSE
           resolve_tac [all_intr,imp_intr,conj_intr] 1)
```

Elimination rules can be applied repeatedly through `eresolve_tac`. This tactic selects an assumption and finally deletes it, to allow termination. A proof of

$$(\exists x \,.\, P(f(x))) \vee (\exists y \,.\, P(g(y))) \supset (\exists z \,.\, P(z))$$

goes by ⊃I, ∨E, ∃E, ∃E, and ∃I. The order is critical: ∃E must be applied before ∃I in each subgoal. We could tackle the proof as follows:

```
> goal Int_Rule.thy "(EX x. P(f(x))) | (EX y. P(g(y))) --> (EX z. P(z))";
Level 0
(EX x. P(f(x))) | (EX y. P(g(y))) --> (EX z. P(z))
 1. (EX x. P(f(x))) | (EX y. P(g(y))) --> (EX z. P(z))
> by (resolve_tac [imp_intr] 1);
Level 1
(EX x. P(f(x))) | (EX y. P(g(y))) --> (EX z. P(z))
 1. (EX x. P(f(x))) | (EX y. P(g(y))) ==> EX z. P(z)
```

Repeatedly applying the eliminations does not work because subgoal 1 gets stuck. It needs an introduction rule:

```
> by (REPEAT (eresolve_tac [exists_elim,disj_elim] 1));
Level 2
(EX x. P(f(x))) | (EX y. P(g(y))) --> (EX z. P(z))
 1. !ka. P(f(ka)) ==> EX z. P(z)
 2. EX y. P(g(y)) ==> EX z. P(z)
```

We could deal with subgoal 1, then resume applying eliminations. But the theorem can be proved in one step by a tactic that combines the introduction and elimination rules and proof by assumption.

We revert to the original state by calling `choplev`, and then apply the tactic:

```
> choplev 0;
Level 0
(EX x. P(f(x))) | (EX y. P(g(y))) --> (EX z. P(z))
 1. (EX x. P(f(x))) | (EX y. P(g(y))) --> (EX z. P(z))
> by (REPEAT   (assume_tac 1
#       ORELSE  eresolve_tac [exists_elim,disj_elim] 1
#       ORELSE  resolve_tac  [imp_intr,exists_intr] 1));
Level 1
(EX x. P(f(x))) | (EX y. P(g(y))) --> (EX z. P(z))
No subgoals!
```

Note that ∃E (`exists_elim`) is tried before ∃I (`exists_intr`). The z that 'exists' is $f(x)$ in one case and $g(y)$ in the other; unification finds these terms automatically.

4.3 A Prolog interpreter

To demonstrate the power of tacticals, let us construct a Prolog interpreter and execute programs involving lists. This will also illustrate how to extend a theory with new constants and axioms. The function for adding rules to a theory, called `extend_theory`, is fully described in the next chapter. The present example may serve as a model, allowing you to make small theory extensions without knowing everything about theories.

The Prolog program will take the form of a theory. To avoid having to define extra syntax, this theory will be an extension of first-order logic, although the interpreter will not exploit any `FOL` rules. The theory defines the following constants, with their meta-types:

$$\begin{array}{rcl} \mathit{Nil} & : & \mathit{term} \\ \mathit{Cons} & : & [\mathit{term}, \mathit{term}] \to \mathit{term} \\ \mathit{append} & : & [\mathit{term}, \mathit{term}, \mathit{term}] \to \mathit{form} \\ \mathit{reverse} & : & [\mathit{term}, \mathit{term}] \to \mathit{form} \end{array}$$

Viewed from the object-level, *Nil* is a constant, *Cons* is a 2-place function, *append* is a 3-place predicate, and *reverse* is a 2-place predicate. We now declare `const_decs` to hold these constant declarations:

```
val const_decs =
  [ (["Nil"],      Aterm),
    (["Cons"],     [Aterm,Aterm]--->Aterm),
    (["append"],   [Aterm,Aterm,Aterm]--->Aform),
    (["reverse"],  [Aterm,Aterm]--->Aform) ];
```

The theory defines *append* by two rules that correspond to the usual PROLOG clauses:

$$append(Nil, ys, ys) \qquad \frac{append(xs, ys, zs)}{append(Cons(x, xs), ys, Cons(x, zs))}$$

It also defines *reverse*, the slow version using *append*:

$$reverse(Nil, Nil) \qquad \frac{reverse(xs, ys) \quad append(ys, Cons(x, Nil), zs)}{reverse(Cons(x, xs), zs)}$$

These rules are included in the theory declaration:

```
val prolog_thy =
  extend_theory Int_Rule.thy "prolog" ([], const_decs)
    [ ("appnil",
         "append(Nil,ys,ys)"),
      ("appcons",
         "append(xs,ys,zs) ==> append(Cons(x,xs), ys, Cons(x,zs))"),
      ("revnil",
         "reverse(Nil,Nil)"),
      ("revcons",
         "[| reverse(xs,ys);  append(ys, Cons(x,Nil), zs) |] ==>  \
\         reverse(Cons(x,xs), zs)") ];
```

Let us go through this point by point. The theory extends `Int_Rule.thy`, the theory of intuitionistic first-order logic. The string `"prolog"` is essentially a comment; it is part of the theory value and helps to identify it. The pair `([], const_decs)` combines an empty list of type names with our constant declarations. (New types are seldom required in theory extensions.) The final list consists of the rules paired with their names, both represented by strings.

Evaluating the above declaration parses and type-checks the rules, constructs the theory, and binds it to the ML identifier `prolog_thy`. The rules can be bound

to ML identifiers using the function `get_axiom`.

```
> val appnil = get_axiom prolog_thy "appnil";
val appnil = ? : thm
> val appcons = get_axiom prolog_thy "appcons";
val appcons = ? : thm
> val revnil = get_axiom prolog_thy "revnil";
val revnil = ? : thm
> val revcons = get_axiom prolog_thy "revcons";
val revcons = ? : thm
```

Repeated application of these rules solves PROLOG goals. Let us append the lists $[a, b, c]$ and $[d, e]$:

```
> goal prolog_thy
#    "append(Cons(a,Cons(b,Cons(c,Nil))), Cons(d,Cons(e,Nil)), ?x)";
Level 0
append(Cons(a,Cons(b,Cons(c,Nil))),Cons(d,Cons(e,Nil)),?x)
 1. append(Cons(a,Cons(b,Cons(c,Nil))),Cons(d,Cons(e,Nil)),?x)
```

As the rules are applied, observe how the answer builds up in the scheme variable `?x`.

```
> by (resolve_tac [appnil,appcons] 1);
Level 1
append(Cons(a,Cons(b,Cons(c,Nil))),Cons(d,Cons(e,Nil)),Cons(a,?zs1))
 1. append(Cons(b,Cons(c,Nil)),Cons(d,Cons(e,Nil)),?zs1)
> by (resolve_tac [appnil,appcons] 1);
Level 2
append(Cons(a,Cons(b,Cons(c,Nil))),Cons(d,Cons(e,Nil)),
       Cons(a,Cons(b,?zs2)))
 1. append(Cons(c,Nil),Cons(d,Cons(e,Nil)),?zs2)
```

As seen in the main goal, the first two elements of the result list are a and b.

```
> by (resolve_tac [appnil,appcons] 1);
Level 3
append(Cons(a,Cons(b,Cons(c,Nil))),Cons(d,Cons(e,Nil)),
       Cons(a,Cons(b,Cons(c,?zs3))))
 1. append(Nil,Cons(d,Cons(e,Nil)),?zs3)
> by (resolve_tac [appnil,appcons] 1);
Level 4
append(Cons(a,Cons(b,Cons(c,Nil))),Cons(d,Cons(e,Nil)),
       Cons(a,Cons(b,Cons(c,Cons(d,Cons(e,Nil))))))
No subgoals!
```

PROLOG can run functions backwards. What list x can be appended with $[c, d]$ to produce $[a, b, c, d]$?

```
> goal prolog_thy
#   "append(?x, Cons(c,Cons(d,Nil)), Cons(a,Cons(b,Cons(c,Cons(d,Nil)))))";
Level 0
append(?x,Cons(c,Cons(d,Nil)),Cons(a,Cons(b,Cons(c,Cons(d,Nil)))))
 1. append(?x,Cons(c,Cons(d,Nil)),Cons(a,Cons(b,Cons(c,Cons(d,Nil)))))
```

Using the tactical REPEAT, the answer is found at once: $[a, b]$.

```
> by (REPEAT (resolve_tac [appnil,appcons] 1));
Level 1
append(Cons(a,Cons(b,Nil)),Cons(c,Cons(d,Nil)),
       Cons(a,Cons(b,Cons(c,Cons(d,Nil)))))
No subgoals!
```

Now for an example of backtracking. What lists x and y can be appended to form the list $[a, b, c, d]$?

```
> goal prolog_thy "append(?x, ?y, Cons(a,Cons(b,Cons(c,Cons(d,Nil)))))";
Level 0
append(?x,?y,Cons(a,Cons(b,Cons(c,Cons(d,Nil)))))
 1. append(?x,?y,Cons(a,Cons(b,Cons(c,Cons(d,Nil)))))
```

Using REPEAT to apply the rules, we find one solution quickly: $x = []$ and $y = [a, b, c, d]$.

```
> by (REPEAT (resolve_tac [appnil,appcons] 1));
Level 1
append(Nil,Cons(a,Cons(b,Cons(c,Cons(d,Nil)))),
       Cons(a,Cons(b,Cons(c,Cons(d,Nil)))))
No subgoals!
```

The subgoal module's back() command finds the next possible outcome from the tactic: $x = [a]$ and $y = [b, c, d]$.

```
> back();
Level 1
append(Cons(a,Nil),Cons(b,Cons(c,Cons(d,Nil))),
       Cons(a,Cons(b,Cons(c,Cons(d,Nil)))))
No subgoals!
```

Other solutions are generated similarly.

```
> back();
Level 1
append(Cons(a,Cons(b,Nil)),Cons(c,Cons(d,Nil)),
       Cons(a,Cons(b,Cons(c,Cons(d,Nil)))))
No subgoals!
```

This solution is $x = [a, b]$ and $y = [c, d]$. There are two more solutions:

```
> back();
Level 1
append(Cons(a,Cons(b,Cons(c,Nil))),Cons(d,Nil),
       Cons(a,Cons(b,Cons(c,Cons(d,Nil)))))
No subgoals!
> back();
Level 1
append(Cons(a,Cons(b,Cons(c,Cons(d,Nil)))),Nil,
       Cons(a,Cons(b,Cons(c,Cons(d,Nil)))))
No subgoals!
```

The solution $x = [a, b, c, d]$ and $y = []$ is the last:

```
> back();
backtrack: no alternatives
Exception- ERROR raised
```

Now let us try *reverse*. What is the reverse of the list $[a, b, c, d]$?

```
> goal prolog_thy "reverse(Cons(a,Cons(b,Cons(c,Cons(d,Nil)))), ?x)";
Level 0
reverse(Cons(a,Cons(b,Cons(c,Cons(d,Nil)))),?x)
 1. reverse(Cons(a,Cons(b,Cons(c,Cons(d,Nil)))),?x)
```

We bundle the rules together as the ML identifier `rules` for giving to `resolve_tac`. Naive reverse runs surprisingly fast!

```
> val rules = [appnil,appcons,revnil,revcons];
val rules = [?, ?, ?, ?] : thm list
> by (REPEAT (resolve_tac rules 1));
Level 1
reverse(Cons(a,Cons(b,Cons(c,Cons(d,Nil)))),
        Cons(d,Cons(c,Cons(b,Cons(a,Nil)))))
No subgoals!
```

Now let us run *reverse* backwards: this, too, should reverse a list. What list is the reverse of $[a, b, c]$?

```
> goal prolog_thy "reverse(?x, Cons(a,Cons(b,Cons(c,Nil))))";
Level 0
reverse(?x,Cons(a,Cons(b,Cons(c,Nil))))
 1. reverse(?x,Cons(a,Cons(b,Cons(c,Nil))))
> by (REPEAT (resolve_tac rules 1));
Level 1
reverse(Cons(?x1,Nil),Cons(a,Cons(b,Cons(c,Nil))))
 1. append(Nil,Cons(?x1,Nil),Cons(a,Cons(b,Cons(c,Nil))))
```

The tactic has failed to find a solution! It reaches a dead end at subgoal 1: there is no $x1$ such that [] appended with $[x1]$ equals $[a, b, c]$. Other outcomes can be considered by backtracking.

```
> back();
Level 1
reverse(Cons(?x1,Cons(a,Nil)),Cons(a,Cons(b,Cons(c,Nil))))
 1. append(Nil,Cons(?x1,Nil),Cons(b,Cons(c,Nil)))
```

This is also a dead end, but the next outcome is a success.

```
> back();
Level 1
reverse(Cons(c,Cons(b,Cons(a,Nil))),Cons(a,Cons(b,Cons(c,Nil))))
No subgoals!
```

The problem with REPEAT is that it stops when it cannot continue, regardless of what state is reached. The tactical DEPTH_FIRST searches for a satisfactory state: here, one with no subgoals. We return to the start of the proof and solve it by depth-first search.

```
> choplev 0;
Level 0
reverse(?x,Cons(a,Cons(b,Cons(c,Nil))))
 1. reverse(?x,Cons(a,Cons(b,Cons(c,Nil))))
> by (DEPTH_FIRST (has_fewer_prems 1) (resolve_tac rules 1));
Level 1
reverse(Cons(c,Cons(b,Cons(a,Nil))),Cons(a,Cons(b,Cons(c,Nil))))
No subgoals!
```

Since PROLOG uses depth-first search, the above tactic is a PROLOG interpreter. Although real PROLOG systems run overwhelmingly faster, logic programming techniques are of great importance when designing Isabelle tactics. Let us name this one.

```
> val prolog_tac = DEPTH_FIRST (has_fewer_prems 1) (resolve_tac rules 1);
val prolog_tac = Tactic fn : tactic
```

We try it on one more example:

```
> goal prolog_thy "reverse(Cons(a,Cons(?x,Cons(c,Cons(?y,Nil)))), \
# \                         Cons(d,Cons(?z,Cons(b,?u))))";
Level 0
reverse(Cons(a,Cons(?x,Cons(c,Cons(?y,Nil)))),
        Cons(d,Cons(?z,Cons(b,?u))))
 1. reverse(Cons(a,Cons(?x,Cons(c,Cons(?y,Nil)))),
            Cons(d,Cons(?z,Cons(b,?u))))
```

Tactic prolog_tac solves it immediately:

```
> by prolog_tac;
Level 1
reverse(Cons(a,Cons(b,Cons(c,Cons(d,Nil)))),
        Cons(d,Cons(c,Cons(b,Cons(a,Nil)))))
No subgoals!
```

4.4 Deriving rules

Tacticals are a rather complicated mechanism for writing short and abstract proofs. The simplest way to write short proofs is through derived rules. This section describes how to derive rules in Isabelle, showing the pitfalls to avoid.

Suppose your style of proof involves replacing $P \,\&\, Q$ by $Q \,\&\, P$ frequently. You might want to derive the rule $P \,\&\, Q / Q \,\&\, P$. Its logical derivation is simple:

$$\dfrac{\dfrac{P \,\&\, Q}{Q} \qquad \dfrac{P \,\&\, Q}{P}}{Q \,\&\, P}$$

The wrong way to derive a rule in Isabelle is to make its conclusion your goal and try to reduce it to the subgoals. Let us try this, stating the goal and applying &I. At first things seem to be working:

```
> goal Int_Rule.thy "?Q & ?P";
Level 0
?Q & ?P
 1. ?Q & ?P
> by (resolve_tac [conj_intr] 1);
Level 1
?Q & ?P
 1. ?Q
 2. ?P
```

Now we intend to apply the conjunction elimination rules to these subgoals, reducing both to `?P&?Q`.

```
> by (resolve_tac [conjunct2] 1);
Level 2
?Q & ?P
 1. ?P2 & ?Q
 2. ?P
```

Subgoal 1 refers not to `?P` but to the new variable `?P2`. And things get worse, especially if we make a mistake.

```
> by (resolve_tac [conj_intr] 2);
Level 3
?Q & ?P3 & ?Q3
 1. ?P2 & ?Q
 2. ?P3
 3. ?Q3
```

The rule `conjunct1` should have been applied. But `conj_intr` is accepted, modifying the main goal. In other words, the conclusion of our intended rule has

changed! This will not do.

```
> undo();
Level 2
?Q & ?P
 1. ?P2 & ?Q
 2. ?P
> by (resolve_tac [conjunct1] 2);
Level 3
?Q & ?P
 1. ?P2 & ?Q
 2. ?P & ?Q3
```

This cannot be the way to derive $P \,\&\, Q / Q \,\&\, P$. To avoid such problems, remember two things:

- Never use scheme variables in the main goal unless they are intended to be instantiated in the proof. Use ordinary free variables.
- State the premises of the rule as meta-assumptions, as discussed below.

Isabelle's meta-logic is a natural deduction system, and therefore its theorems may depend upon assumptions. When theorems are used in proofs, the assumptions accumulate. At the end of the proof, they can be discharged, forming meta-implications. At the same time, free variables in the theorem are converted into schematic variables. You need not perform such meta-inferences directly; Isabelle's subgoal commands take care of them.

Here is the correct way to derive the rule. The premise of the proposed rule, namely $P \,\&\, Q$, is stated in the original goal. The command `goal` returns a one-element list of theorems, containing this premise. The premise depends upon the assumption $P \,\&\, Q$, but may be used in the proof as though it were an axiom.

```
> val prems = goal Int_Rule.thy "P&Q ==> Q&P";
Level 0
Q & P
 1. Q & P
val prems = [?] : thm list
```

The variable `prems` is bound to the list of premises returned by `goal`. We have seldom done this before: when there are no premises, the resulting empty list can be ignored. We now apply the three rules of the derivation.

```
> by (resolve_tac [conj_intr] 1);
Level 1
Q & P
 1. Q
 2. P
> by (resolve_tac [conjunct2] 1);
Level 2
Q & P
 1. ?P1 & Q
 2. P
```

Last time we incorrectly applied `conj_intr` to subgoal 2 here. Now that error is impossible because `P` cannot be instantiated.

```
> by (resolve_tac [conjunct1] 2);
Level 3
Q & P
 1. ?P1 & Q
 2. P & ?Q2
```

Resolving these subgoals with the premise $P \,\&\, Q$ instantiates the variables `?P1` and `?Q2`, completing the derivation.

```
> by (resolve_tac prems 1);
Level 4
Q & P
 1. P & ?Q2
> by (resolve_tac prems 1);
Level 5
Q & P
No subgoals!
```

The function `result` discharges the premise and makes P and Q into scheme variables.

```
> prth (result());
?P & ?Q ==> ?Q & ?P
```

An expert tactician might replace all the steps after the initial `conj_intr` by the following tactic. It is designed to prevent the rules `conjunct1` and `conjunct2` from causing infinite repetition. They could apply to any goal, so the body of the `REPEAT` insists upon solving the resulting subgoal by resolution with the premise, $P \,\&\, Q$.

```
by (REPEAT    (resolve_tac [conjunct1,conjunct2] 1
         THEN  resolve_tac prems 1));
```

The call to

```
resolve_tac [conjunct1,conjunct2] 1
```

always returns two outcomes, but the following call to

```
resolve_tac prems 1
```

kills at least one of them.

A collection of derivations may be kept on a file and executed in 'batch mode' using the function `prove_goal`. This function creates an initial proof state, then applies a series of tactics to it. The resulting state (the first if multiple outcomes are generated) is returned by `prove_goal`, after checking that it is identical to the

theorem that was originally proposed. Unlike the subgoal commands, `prove_goal` is purely functional. The above derivation can be packaged as follows:

```
val conj_rule = prove_goal Int_Rule.thy "P&Q ==> Q&P"
 (fn prems=>
  [ (resolve_tac [conj_intr] 1),
    (REPEAT    (resolve_tac [conjunct1,conjunct2] 1
         THEN   resolve_tac prems 1)) ]);
```

The theorem is proved and bound to the ML identifier `conj_rule`.

4.5 Definitions and derived rules

Serious proofs are seldom performed in pure logic, but concern the properties of constructions or the consequences of additional assumptions. In Isabelle this involves extending a theory with new constants and definitions or other axioms. Isabelle makes little distinction between defining a trivial abbreviation like $1 \equiv Succ(0)$ and defining an entire logic. We have already considered one example of extending a theory, when we made a PROLOG interpreter. Now we shall make a definition and prove some abstract rules for it.

Classical first-order logic can be extended with the propositional connective $if(P, Q, R)$, where

$$if(P, Q, R) \equiv P \,\&\, Q \vee \neg P \,\&\, R\ .$$

Theorems about *if* can be proved by treating this as an abbreviation. The tactic `rewrite_goals_tac` can replace $if(P, Q, R)$ by $P \,\&\, Q \vee \neg P \,\&\, R$ in subgoals. Unfortunately, this may produce an unreadable subgoal. The formula P is duplicated, possibly causing an exponential blowup. This problem tends to get worse as more and more abbreviations are introduced.

Natural deduction demands rules that introduce and eliminate $if(P, Q, R)$ directly, without reference to its definition. The design of natural rules is seldom easy. The identity

$$if(P, Q, R) \leftrightarrow (P \supset Q) \,\&\, (\neg P \supset R)\ ,$$

which is straightforwardly demonstrated, suggests that the *if*-introduction rule should be

$$\frac{\begin{array}{cc}[P] & [\neg P] \\ Q & R\end{array}}{if(P, Q, R)}$$

The *if*-elimination rule follows the definition of $if(P, Q, R)$ by the elimination rules for $\vee$ and $\&$.

$$\frac{if(P, Q, R) \qquad \begin{array}{cc}[P, Q] & [\neg P, R] \\ S & S\end{array}}{S}$$

Having made these plans, we get down to work with Isabelle. The theory of natural deduction classical logic, `cla_thy`, is extended with the constant *if* of type

$$if \quad : \quad [form, form, form] \to form$$

The axiom `if_def` equates $if(P, Q, R)$ with $P \,\&\, Q \vee \neg P \,\&\, R$.

```
> val if_thy =
#   extend_theory cla_thy "if"
#     ([],   [ (["if"], [Aform,Aform,Aform]--->Aform) ])
#     [ ("if_def", "if(P,Q,R) == P&Q | ~P & R") ];
val if_thy = ? : theory
> val if_def = get_axiom if_thy "if_def";
val if_def = ? : thm
```

The derivations of the introduction and elimination rules demonstrate the methods for rewriting with definitions. Complicated classical reasoning is involved, but the tactic `Pc.fast_tac` is able to cope.

First, the introduction rule. We state it, with its premises $P \Longrightarrow Q$ and $\neg P \Longrightarrow R$, and the goal $if(P, Q, R)$. Calling `rewrite_goals_tac` rewrites occurrences of *if* in the subgoals, while leaving the main goal unchanged.

```
> val prems = goal if_thy "[| P ==> Q; ~P ==> R |] ==> if(P,Q,R)";
Level 0
if(P,Q,R)
 1. if(P,Q,R)
val prems = [?, ?] : thm list
> by (rewrite_goals_tac [if_def]);
Level 1
if(P,Q,R)
 1. P & Q | ~P & R
```

The premises (in the variable `prems`) are passed to `Pc.fast_tac`, which uses them in proving the goal.

```
> by (Pc.fast_tac prems 1);
Level 2
if(P,Q,R)
No subgoals!
> val if_intr = result();
val if_intr = ? : thm
```

Next, we state the elimination rule. It has three premises, two of which are themselves rules. The conclusion is simply S.

```
> val prems = goal if_thy
    "[| if(P,Q,R);  [| P; Q |] ==> S; [| ~P; R |] ==> S |] ==> S";
# Level 0
S
 1. S
val prems = [?, ?, ?] : thm list
```

How can we rewrite the occurrences of *if* in the premises? One way is to incorporate the premises into the subgoal so that `rewrite_goals_tac` will work. The

tactic `cut_facts_tac` inserts theorems into a subgoal as assumptions. Only simple theorems are inserted: not rules having premises.

```
> by (cut_facts_tac prems 1);
Level 1
S
 1. if(P,Q,R) ==> S
```

The assumption in the subgoal is rewritten. The resulting subgoal falls to `Pc.fast_tac`.

```
> by (rewrite_goals_tac [if_def]);
Level 2
S
 1. P & Q | ~P & R ==> S
> by (Pc.fast_tac prems 1);
Level 3
S
No subgoals!
> val if_elim = result();
val if_elim = ? : thm
```

Another way of rewriting in the premises is by `rewrite_rule`, which is a meta-inference rule: a function from theorems to theorems. Calling

```
rewrite_rule [if_def] th
```

rewrites all occurrences of *if* in the theorem *th*. All occurrences in the premises may be rewritten with the help of the standard list functional `map`:

```
> map (rewrite_rule [if_def]) prems;
val it = [?, ?, ?] : thm list
> prths it;
P & Q | ~P & R  [ if(P,Q,R) ]

[| P; Q |] ==> S  [ [| P; Q |] ==> S ]

[| ~P; R |] ==> S  [ [| ~P; R |] ==> S ]
```

Only the first premise is affected. Observe how the assumptions appear in square brackets to the right.

The rules just derived have been saved with the names `if_intr` and `if_elim`. They permit natural proofs of theorems such as the following:

$$\begin{aligned} \mathit{if}(P, \mathit{if}(Q,A,B), \mathit{if}(Q,C,D)) &\leftrightarrow \mathit{if}(Q, \mathit{if}(P,A,C), \mathit{if}(P,B,D)) \\ \mathit{if}(\mathit{if}(P,Q,R),A,B) &\leftrightarrow \mathit{if}(P, \mathit{if}(Q,A,B), \mathit{if}(R,A,B)) \end{aligned}$$

Some additional rules are needed, such as those for classical reasoning and the $\leftrightarrow$-introduction rule (called `iff_intr`: do not confuse with `if_intr`).

To display the *if*-rules in action, let us analyse a proof step by step.

```
> goal if_thy
#     "if(P, if(Q,A,B), if(Q,C,D)) <-> if(Q, if(P,A,C), if(P,B,D))";
Level 0
if(P,if(Q,A,B),if(Q,C,D)) <-> if(Q,if(P,A,C),if(P,B,D))
 1. if(P,if(Q,A,B),if(Q,C,D)) <-> if(Q,if(P,A,C),if(P,B,D))
> by (resolve_tac [iff_intr] 1);
Level 1
if(P,if(Q,A,B),if(Q,C,D)) <-> if(Q,if(P,A,C),if(P,B,D))
 1. if(P,if(Q,A,B),if(Q,C,D)) ==> if(Q,if(P,A,C),if(P,B,D))
 2. if(Q,if(P,A,C),if(P,B,D)) ==> if(P,if(Q,A,B),if(Q,C,D))
```

The *if*-elimination rule can be applied twice in succession.

```
> by (eresolve_tac [if_elim] 1);
Level 2
if(P,if(Q,A,B),if(Q,C,D)) <-> if(Q,if(P,A,C),if(P,B,D))
 1. [| P; if(Q,A,B) |] ==> if(Q,if(P,A,C),if(P,B,D))
 2. [| ~P; if(Q,C,D) |] ==> if(Q,if(P,A,C),if(P,B,D))
 3. if(Q,if(P,A,C),if(P,B,D)) ==> if(P,if(Q,A,B),if(Q,C,D))
> by (eresolve_tac [if_elim] 1);
Level 3
if(P,if(Q,A,B),if(Q,C,D)) <-> if(Q,if(P,A,C),if(P,B,D))
 1. [| P; Q; A |] ==> if(Q,if(P,A,C),if(P,B,D))
 2. [| P; ~Q; B |] ==> if(Q,if(P,A,C),if(P,B,D))
 3. [| ~P; if(Q,C,D) |] ==> if(Q,if(P,A,C),if(P,B,D))
 4. if(Q,if(P,A,C),if(P,B,D)) ==> if(P,if(Q,A,B),if(Q,C,D))
```

In the first two subgoals, all formulae have been reduced to atoms. Now *if*-introduction can be applied. Observe how the *if*-rules break down occurrences of *if* when they come to the top — as the outermost connective.

```
> by (resolve_tac [if_intr] 1);
Level 4
if(P,if(Q,A,B),if(Q,C,D)) <-> if(Q,if(P,A,C),if(P,B,D))
 1. [| P; Q; A; Q |] ==> if(P,A,C)
 2. [| P; Q; A; ~Q |] ==> if(P,B,D)
 3. [| P; ~Q; B |] ==> if(Q,if(P,A,C),if(P,B,D))
 4. [| ~P; if(Q,C,D) |] ==> if(Q,if(P,A,C),if(P,B,D))
 5. if(Q,if(P,A,C),if(P,B,D)) ==> if(P,if(Q,A,B),if(Q,C,D))
> by (resolve_tac [if_intr] 1);
Level 5
if(P,if(Q,A,B),if(Q,C,D)) <-> if(Q,if(P,A,C),if(P,B,D))
 1. [| P; Q; A; Q; P |] ==> A
 2. [| P; Q; A; Q; ~P |] ==> C
 3. [| P; Q; A; ~Q |] ==> if(P,B,D)
 4. [| P; ~Q; B |] ==> if(Q,if(P,A,C),if(P,B,D))
 5. [| ~P; if(Q,C,D) |] ==> if(Q,if(P,A,C),if(P,B,D))
 6. if(Q,if(P,A,C),if(P,B,D)) ==> if(P,if(Q,A,B),if(Q,C,D))
```

Where do we stand? The first subgoal holds by assumption; the second and third, by contradiction. This is starting to get tedious. Let us revert to the initial proof state and solve the whole thing at once. The following tactic uses `mp_tac` to detect the contradictions.

```
> choplev 0;
Level 0
if(P,if(Q,A,B),if(Q,C,D)) <-> if(Q,if(P,A,C),if(P,B,D))
 1. if(P,if(Q,A,B),if(Q,C,D)) <-> if(Q,if(P,A,C),if(P,B,D))
> by (REPEAT (assume_tac 1
#     ORELSE mp_tac 1
#     ORELSE eresolve_tac [if_elim] 1
#     ORELSE resolve_tac [iff_intr,if_intr] 1));
Level 1
if(P,if(Q,A,B),if(Q,C,D)) <-> if(Q,if(P,A,C),if(P,B,D))
No subgoals!
```

Our other example is harder to prove, for it requires the swap rule.[3] The classical logic tactic `Pc.step_tac` takes care of this. You might try this proof in single steps.

```
> goal if_thy
#     "if(if(P,Q,R), A, B) <-> if(P, if(Q,A,B), if(R,A,B))";
Level 0
if(if(P,Q,R),A,B) <-> if(P,if(Q,A,B),if(R,A,B))
 1. if(if(P,Q,R),A,B) <-> if(P,if(Q,A,B),if(R,A,B))
> by (REPEAT (Pc.step_tac [if_intr] 1 ORELSE eresolve_tac [if_elim] 1));
Level 1
if(if(P,Q,R),A,B) <-> if(P,if(Q,A,B),if(R,A,B))
No subgoals!
```

Of course, we can dispense with the *if*-rules entirely, instead treating *if* as an abbreviation. The resulting propositional formula is easily proved by `Pc.fast_tac`.

```
> choplev 0;
Level 0
if(if(P,Q,R),A,B) <-> if(P,if(Q,A,B),if(R,A,B))
 1. if(if(P,Q,R),A,B) <-> if(P,if(Q,A,B),if(R,A,B))
> by (rewrite_goals_tac [if_def]);
Level 1
if(if(P,Q,R),A,B) <-> if(P,if(Q,A,B),if(R,A,B))
 1. (P & Q | ~P & R) & A | ~(P & Q | ~P & R) & B <->
    P & (Q & A | ~Q & B) | ~P & (R & A | ~R & B)
> by (Pc.fast_tac [] 1);
Level 2
if(if(P,Q,R),A,B) <-> if(P,if(Q,A,B),if(R,A,B))
No subgoals!
```

[3] Formulating the theory in the sequent calculus `LK` would avoid this problem, but deriving rules in a sequent calculus is complicated.

Formally speaking, problems in the extended logic are reduced to the basic logic. This approach has its merits, especially if the prover for the basic logic is very good[4]. It is poor at error detection, however. Suppose some goal is hard to prove. If by reducing the problem we have made it unreadable, then we have little hope of determining what is wrong. Let us try to prove something invalid:

```
> goal if_thy
#     "if(if(P,Q,R), A, B) <-> if(P, if(Q,A,B), if(R,B,A))";
 Level 0
if(if(P,Q,R),A,B) <-> if(P,if(Q,A,B),if(R,B,A))
 1. if(if(P,Q,R),A,B) <-> if(P,if(Q,A,B),if(R,B,A))
> by (REPEAT (Pc.step_tac [if_intr] 1 ORELSE eresolve_tac [if_elim] 1));
Level 1
if(if(P,Q,R),A,B) <-> if(P,if(Q,A,B),if(R,B,A))
 1. [| ~P; R; A; ~P; R |] ==> B
 2. [| ~P; R; ~if(P,Q,R); B |] ==> B
 3. [| if(if(P,Q,R),A,B); ~P; ~R |] ==> A
 4. if(P,if(Q,A,B),if(R,B,A)) ==> if(if(P,Q,R),A,B)
```

Subgoal 1 is unprovable, and tells us that if P and B are false while R and A are true then the original goal becomes false. In fact it evaluates to $\mathit{true} \leftrightarrow \mathit{false}$. The other subgoals are in a recognizable form.

If we tackle the original goal by `rewrite_goals_tac` and `Pc.fast_tac`, then the latter tactic fails to terminate!

```
> undo();
Level 0
if(if(P,Q,R),A,B) <-> if(P,if(Q,A,B),if(R,B,A))
 1. if(if(P,Q,R),A,B) <-> if(P,if(Q,A,B),if(R,B,A))
> by (rewrite_goals_tac [if_def]);
Level 1
if(if(P,Q,R),A,B) <-> if(P,if(Q,A,B),if(R,B,A))
 1. (P & Q | ~P & R) & A | ~(P & Q | ~P & R) & B <->
    P & (Q & A | ~Q & B) | ~P & (R & B | ~R & A)
val it = () : unit
> by (Pc.fast_tac [] 1);
```

This is a bad advertisement for `Pc.fast_tac`. Even if the tactic simply failed, however, we would have no idea why the goal was invalid.

[4]which will not be the case for Isabelle!

Chapter 5

Defining Logics

This chapter is for Isabelle experts only. It explains how to define new logical systems, Isabelle's *raison d'être*. In doing so, Isabelle's internals have to be discussed in a certain amount of detail. Section 5.1 describes the internal representations of types and terms, a prerequisite for understanding the rest of the chapter. This is followed by sections on how to define hierarchical theories, on the precise definition of the meta-logic's syntax, and on how to specify a concrete syntax and a pretty printer for some new object-logic. Section 5.6 shows how all this fits together by giving some simple examples of complete logic definitions.

5.1 Types and terms

Isabelle is based on the idea that proofs in many logics can be naturally represented in intuitionistic higher-order logic [9], henceforth called the *meta-logic*. In Isabelle, like in higher-order logic, the terms are those of the typed λ-calculus. Types and terms are represented by the ML types `typ` and `term`.

5.1.1 The ML type `typ`

Every term has a type. The type `typ` is defined as follows:

```
infixr 5 -->;
datatype typ = Ground of string
             | Poly of string
             | op --> of typ * typ;
```

A *ground* type has a name, represented by a string, as does a *polymorphic* type (or type variable). A *function* type has the form `S-->T`. Two types are equal if they have identical structure: ML's equality test is correct for types.

A term of type `S-->T` denotes a function: if applied to a term of type `S`, the resulting term has type `T`. A term should obey the type-checking rules of the typed λ-calculus. It is possible to construct ill-typed terms, but the meta-rules ensure that all terms in theorems are well-typed.

Functions of several arguments are expressed by currying. The operator `-->` associates to the right, so

```
S1-->(S2-->(S3-->T))
```

can be written `S1-->S2-->S3-->T`. There is an ML operator `--->` for writing this as `[S1,S2,S3]--->T`.

Example: suppose that `f` has type `S-->T-->S`, where `S` and `T` are distinct types, `a` has type S, and `b` has type `T`. Then `f(a)` has type `T-->S` and `f(a,b)` has type `S`, while `f(b)`, `f(a,a)`, and `f(a,b,a)` are ill-typed. Note that `f(a,b)` means `(f(a))(b)`.

Type variables (`Poly`) permit ML-style type inference (see Section 5.5). They are used only internally and may not appear in theorems. A typed object-logic can be represented by making the object-level typing rules explicit. See the section on Constructive Type Theory, which is the ultimate example of a typed logic.

5.1.2 The ML type `term`

There are six kinds of term.

```
type indexname = string * int;

infix 9 $;
datatype term = Const of string * typ
              | Free  of string * typ
              | Var   of indexname * typ
              | Bound of int
              | Abs   of string * typ * term
              | op $  of term * term;
```

A *constant* has a name and a type. Constants include connectives like $\wedge$ and $\forall$ (logical constants), as well as constants like 0 and succ. Other constants may be required to define the concrete syntax of a logic.

A *free variable* (or `Free`) has a name and a type.

A *scheme variable* (or `Var`) has an indexname and a type, where an indexname is a string paired with a non-negative index. A `Var` is logically the same as a free variable. It stands for a variable that may be instantiated during unification. The `Var`s in a term can be systematically renamed by incrementing the indices. In PROLOG jargon these are 'logical variables' and they may be 'standardized apart'.

A *bound variable* has no name, only a number: de Bruijn's representation [1]. The number counts the number of lambdas, starting from zero, between an occurrence of the variable and the lambda that binds it. The representation prevents capture of bound variables, allowing a simple and quick substitution function. The type of a bound variable is stored with its binding lambda (an `Abs` node). For more information see de Bruijn [1] or look at the functions `incr_boundvars`, `subst_bounds`, `aconv`.

An *abstraction* stores the name and type of its bound variable, and its body. The name is used only for parsing and printing; it has no logical significance.

An *application* consists of a term applied to another term. The constructor is the infix `$`, so the ML expression `t$u` constructs the application of `t` to `u`.

5.2 Theories

Isabelle logics are hierarchies of *theories*. Each theory consists of a signature and inference rules. *Signatures* describe the syntax of a logic, both abstract and concrete. The definition of the concrete syntax is described in great detail in Section 5.4. For this section let us assume the following ML type abbreviations:

```
type sorts = string list;
type ops = (string list * typ) list;
type rules = (string * string) list;
```

The types `sorts` and `ops` describe the *abstract syntax* and corresponds to a signature in universal algebra: `sorts` is the list of all type names, `ops` the list of all typed *constants* that are declared in a theory.

Example 5.1 The abstract syntax defined by `isorts` and `iops` introduces the two type names `"int"` and `"form"`, two binary operations `"+"` and `"-"` on `"int"`, and a binary function `"="` from `"int"` to `"form"`.

```
val Tint = Ground "int";
val Tform = Ground "form";
val isorts = ["int", "form"];
val iops = [(["+", "-"], [Tint,Tint] ---> Tint),
            (["="],      [Tint,Tint] ---> Tform)];
```

The Isabelle equivalent of abstract syntax trees is the type `term`. An abstract syntax determines which terms are valid and which are not: all constants and ground types must have been declared and the whole term must be type correct. In the context of the above example,

```
Const("+",[Tint,Tint] ---> Tint) $ Free("x",Tint)
```

is a valid term of `typ Tint --> Tint`.

The names and types of the constants `"+"`, `"-"`, and `"="` suggest they represent addition, subtraction, and equality respectively. However, we could have used arbitrary strings since there is no a priori connection between abstract and concrete syntax, with one exception: if the name of a constant is an identifier (as opposed to an arbitrary string), any occurrence of that identifier in an input string is treated as that constant; no special concrete syntax is necessary. The parser does not even know about these identifiers and their types. For that reason, it ignores type information completely: the dummy `typ Adummy` is used by the parser in all terms it generates. The real types are inferred at a later stage (see Section 5.5).

Many Isabelle users will never need to know about internal concepts like types and constants because they think in terms of the external representation determined by the concrete syntax associated with a theory.

The ML type `rules` is a list of pairs of rule names and rules, for example

```
val irls = [("+-comm", "x+y = y+x"), ("?" "x-x = x")];
```

The name of a rule can be an arbitrary string; the rule itself has to conform to the syntax associated with the theory.

The following functions construct elements of the abstract type `theory` of theories in an incremental way:

```
pure_thy: theory
enrich_theory:
    theory -> string -> sorts * ops * syntax -> rules -> theory
extend_theory: theory -> string -> sorts * ops -> rules -> theory
merge_theories: theory * theory -> theory
```

The abstract type `syntax` and the hierarchical definition of syntaxes is described in Section 5.4.

`pure_thy` contains just the types, constants, and syntax of the meta-logic. Note that there are no rules since meta-logical inferences are carried out by ML functions as in LCF.

```
enrich_theory thy s (sorts,ops,syn) axs
```

returns a new theory which enriches `thy` with the ground types in `sorts`, the constants in `ops`, and the rules `axs`. The concrete syntax of the new theory is given by `syn`. Note that concrete syntaxes are also built up incrementally, but with a different set of functions described in Section 5.4. The rules in `axs` are interpreted under the new signature. All `Free`s are replaced by `Var`s, so that axioms can be written without question marks. The name `s` of the new theory is there merely for reasons of documentation. Example: `enrich_theory pure_thy "int" (isorts,iops,syn) irls`, where `isorts`, `iops` and `irls` are as in Example 5.1 above and `syn` defines some appropriate syntax.

```
extend_theory thy s (sorts,ops) axs
```

is equivalent to

```
enrich_theory thy s (sorts,ops,syn) axs
```

where `syn` is the concrete syntax of theory `thy`. This is a simplified way of enriching theories in case no new concrete syntax information has to be added. In essence, this means that the signature extension involves only identifiers.

`merge_theories(th1,th2)` merges the two theories `th1` and `th2`. The resulting theory contains the union of all types, constants and axioms of the constituent theories.

Isabelle does not support overloading, i.e. a theory cannot contain two constants with the same name but different types. Trying to create such a theory by enriching or merging raises an exception. Constants with the same name and the same type are identified when theories are enriched or merged.

Axioms can be extracted from theories with the function

```
get_axiom: theory -> string -> thm
```

If you have created a theory with more than one axiom with the same name, `get_axiom` returns an arbitrary one of them. In there is no axiom with a given name, an exception is raised.

Examples of the use of all of these functions are given in Section 5.6.

5.3 The meta-logic

Before you can define new object-logics you have to know the meta-logic. Section 1.2.1 gave a brief account of its features, Figure 5.1 presents the precise description of its syntax.

prop	=	*atomic* `$$ prop`	
	|	`(` *prop* `)`	
	|	Any_3 `==` Any_2	(2)
	|	$prop_2$ `==>` $prop_1$	(1)
	|	`[|` *prop* { `;` *prop* } `|]` `==>` $prop_1$	(1)
	|	`!` *id-list* `.` *prop*	(0)
Any	=	*prop*	
atomic	=	*identifier*	
	|	*variable*	
	|	*identifier* `(` *Any* { `,` *Any* } `)`	
	|	*variable* `(` *Any* { `,` *Any* } `)`	
id-list	=	*identifier* { *identifier* }	

Figure 5.1: Meta-Logic Syntax

The syntactic category *prop* corresponds to the ML identifier `Aprop:typ`. *Any* is the type of all types. Initially, it contains just *prop*. As the syntax is extended by new object-logics, more productions for *Any* are added (see Section 5.4.4). The abstract syntax constants for equality, implication and universal quantification are `"=="`, `"==>"`, and `"all"` respectively.

Atomic propositions have to be qualified with `$$ prop` in order to allow object-logic formulae to be embedded into the meta-logic with minimal syntactic overhead. This can be seen as a type constraint.

The syntax of the meta-logic is defined in exactly the same way as that of any object-logic. The module `Pure_Ext` contains its definition and can be changed easily to suit different tastes.

5.4 Defining the syntax

Isabelle supports syntax definitions in the form of unrestricted context-free grammars. The module `Syntax`, which exports the type `syntax`, is the interface to this definition facility. Syntaxes are built up in an incremental fashion using the functions

```
pure:   syntax
extend: syntax -> extension -> syntax
merge:  syntax * syntax -> syntax
```

The constant `pure` holds the syntax for *pure* Isabelle, namely higher-order logic (Figure 5.1). New syntaxes are defined either by extending or merging existing ones. In the sequel let `extension` abbreviate the following record type:

```
type extension =
     {logical_types: typ list,
      mixfix:  mixfix list,
      parse_translation: (string * (term list -> term)) list,
      print_translation: (string * (term -> term)) list}
```

The four fields of `extension` are explained in sections 5.4.4, 5.4.1, 5.4.3, and 5.4.5 respectively. Sections 5.4.1 to 5.4.4 explain how to define those aspects of the syntax that have to do with parsing, Section 5.4.5 is concerned with unparsing.

5.4.1 Mixfix syntax

The `mixfix` field of an extension defines the actual syntax through a list of productions of type `mixfix`. In fact, `mixfix` describes the *concrete syntax*, its translation into the abstract syntax, and a pretty-printing scheme, all in one. Isabelle syntax definitions are inspired by OBJ's [2] *mixfix* syntax. Each element of type `mixfix`

```
datatype mixfix = Mixfix of string * typ * string * int list * int
                | Delimfix of string * typ * string
                | Infixl of string * typ * int
                | Infixr of string * typ * int
```

defines a production for a context-free concrete syntax with priorities and associates a typed abstract syntax constant with it. [1] For the moment we concentrate on `Mixfix` because the other ML constructors (`Delimfix`, ...) are derived forms.

As a simple introductory example of the relationship between OBJ's mixfix syntax and the Isabelle variant, consider the OBJ declaration

```
_+_ : int,int -> int
```

In Isabelle this becomes

```
Mixfix("_+_", [Ground"int", Ground"int"] ---> Ground"int", ... )
```

[1] The ML function `constants` explained in Section 5.4.6 can be used to avoid having to declare constants both in the abstract and concrete syntax

where the last three arguments encode information such as priority, which is not part of the OBJ declaration above.

The general form `Mixfix(sy,ty,con,pl,p)` is interpreted as follows:

`sy` : the right-hand side of this production, specified in OBJ-like mixfix form. In general, `sy` is of the form α_0`_`$\alpha_1 \ldots \alpha_{n-1}$`_`$\alpha_n$, where each occurrence of "`_`" denotes an argument/nonterminal and the strings α_i do not contain "`_`".

`ty` : the types of the nonterminals on the left and right hand side. If `sy` is of the form above, `ty` must be of the form `[`$A1, \ldots, An$`] ---> `T. Both the Ai and T may be function types.

`con` : name of the abstract syntax constant associated with this production. Parsing the phrase `sy` generates the `term Const(con,Adummy`[2]`)$a1$...$an`, where `ai` is the term generated by parsing the `i`th argument.

`pl` : must be of the form `[`$p_1, \ldots, p_n$`]`, where p_i is the minimal priority required of any phrase that may appear as the i^{th} argument. The null list is interpreted as a list of `0`'s of the appropriate length.

`p` : priority of this production.

Notice that there is a close connection between abstract and concrete syntax: each production has an associated abstract syntax constant, and the ML type `typ` represents types in the abstract syntax as well as *syntactic categories* in the concrete syntax. To emphasize this connection, we sometimes refer to the nonterminals on the right-hand side of a production as its arguments and to the nonterminal on the left-hand side as its result.

In terms of the priority grammar format introduced in Section 1.6, the declaration `Mixfix(sy,ty,con,pl,p)` defines the production

$$T_p \longrightarrow \alpha_0 A1_{p_1} \alpha_1 \ldots \alpha_{n-1} An_{p_n} \alpha_n$$

The maximal legal priority (m in Section 1.6) is called `max_pri`. If you want to ignore priorities, the safest way to do so is to use `Mixfix(_, _, _, [], max_pri)`: this production puts no priority constraints on any of its arguments (remember that `[]` is equivalent to `[0, ..., 0]`) and has maximum priority itself, i.e. it is always applicable and does not exclude any productions of its arguments.

Example 5.2 In mixfix notation the grammar in Example 1.1 can be written as follows:

```
Mixfix("0",     Aterm, "0", [], 9),
Mixfix("_ + _", [Aterm,Aterm] ---> Aterm, "+", [0,1], 0),
Mixfix("_ * _", [Aterm,Aterm] ---> Aterm, "*", [3,2], 2),
Mixfix("- _",   Aterm --> Aterm, "-", [3], 3)
```

[2]The dummy type `Adummy` is used instead of the obvious `ty` to be consistent with the treatment of types of identifiers. See Section 5.5

Parsing the string "0 + - 0 + 0" produces the term

```
P $ (P $ (M $ Z) $ Z) $ Z
```

where P is Const("+",Adummy), M is Const("-",Adummy), and Z is Const("0", Adummy).

The interpretation of "_" in a mixfix declaration is always as a *meta-character* which does not represent itself but an argument position. The following characters are also meta-characters:

```
'   (   )   /
```

Preceding any character with a quote (') turns it into an ordinary character. Thus you can write "''" if you really want a single quote. The purpose of the other meta-characters is explained in Section 5.4.5. Remember that in ML strings "\" is already a (different kind of) meta-character.

Derived forms

All other ML constructors of type mixfix are derived from Mixfix. Their semantics is explained by the following translation into lists of Mixfix expressions:

```
Delimfix(sy,ty,con)  ⟹  [Mixfix(sy,ty,con,[],max_pri)]
Infixr(sy,ty,p)      ⟹  [Mixfix("op "^sy,ty,sy,[],max_pri),
                         Mixfix("_ "^sy^" _",ty,sy,[p+1,p],p)]
Infixl(sy,ty,p)      ⟹  [Mixfix("op "^sy,ty,sy,[],max_pri),
                         Mixfix("_ "^sy^" _",ty,sy,[p,p+1],p)]
```

Delimfix abbreviates a common form of priority-independent production. Infixl and Infixr declare infix operators which associate to the left and right respectively. As in ML, prefixing infix operators with op turns them into curried functions.

Copy productions

Productions which do not create a new node in the abstract syntax tree are called *copy productions*. They must have exactly one nonterminal on the right hand side. The term generated when parsing that nonterminal is simply passed up as the result of parsing the whole copy production. In Isabelle a copy production is indicated by an empty constant name, i.e. by Mixfix(sy,_,"",_,_).

A special kind of copy production is one where, modulo white space, sy is "_". It is called a *chain production*. Chain productions should be seen as an abbreviation mechanism. Conceptually, they are removed from the grammar by adding appropriate new rules. Priority information attached to chain productions is ignored. The following example demonstrates the effect:

Example 5.3 The grammar defined by

```
Mixfix("A _", B --> A, "AB", [10], _),
Mixfix("_", C --> B, "", [0], 100),
Mixfix("x", C, "x", [],  5),
Mixfix("y", C, "y", [], 15)
```

admits the string `"A y"` but not `"A x"`. Had the constant in the second production been some non-empty string, both `"A y"` and `"A x"` would be legal.

5.4.2 Lexical conventions

The lexical analyzer distinguishes 3 kinds of tokens: *delimiters*, *identifiers* and *variables*. Delimiters are user-defined, i.e. they are extracted from the syntax definition. If α_0`_`$\alpha_1 \ldots \alpha_{n-1}$`_`$\alpha_n$ is some mixfix declaration, each α_i is decomposed into substrings $\beta_1\ \beta_2\ \ldots\ \beta_k$ which are separated by and do not contain *white space* (= blanks, tabs, newlines). Each β_j becomes a delimiter. Thus a delimiter can be an arbitrary string not containing white space.

Identifiers and (scheme) variables are predefined. An identifier is any sequence of letters, digits, primes (`'`) or underbars (`_`) starting with a letter. A variable is a `?` followed by an identifier, optionally followed by a dot (`.`) and a sequence of digits. Parsing an identifier i generates `Free(`i`,Adummy)`. Parsing a variable `?`v generates `Var((`u`,`i`),Adummy)` where i is the integer value of the longest numeric suffix of v (possibly 0), and u is the remaining prefix. Parsing a variable `?`v`.`i generates `Var((`v`,`i`),Adummy)`. The following table covers the four different cases that can arise:

`"?v"`	`"?v.7"`	`"?v5"`	`"?v7.5"`
`Var(("v",0),T)`	`Var(("v",7),T)`	`Var(("v",5),T)`	`Var(("v7",5),T)`

`T` is always `Adummy`.

The two identifiers

```
SId: typ
SVar: typ
```

can be used to refer to identifiers and variables in a syntax definition.

Example 5.4 The production `Delimfix("_", SId --> A, "")` adds identifiers as another alternative for the syntactic category `A`.

`Mixfix("ALL _. _", [SId,Aform] --> Aform, "ALL", [0,3], 2)` defines the syntax for universal quantification. Note how the chosen priorities prohibit nested quantifiers.

If we think of `SId` and `SVar` as nonterminals with predefined syntax, we may assume that all their productions have priority `max_pri`.

The lexical analyzer translates input strings to token lists by repeatedly taking the maximal prefix of the input string that forms a valid token. A maximal prefix

that is both a delimiter and an identifier or variable (like "ALL") is treated as a delimiter. White spaces are separators.

An important consequence of this translation scheme is that delimiters need not be separated by white space to be recognized as separate. If "-" is a delimiter but "--" is not, the string "--" is treated as two consecutive occurrences of "-". This is in contrast to ML which would treat "--" as a single (undeclared) identifier. The consequence of Isabelle's more liberal scheme is that the same string may be parsed in a different way after extending the syntax: after adding "--" as a delimiter, the input "--" is treated as a single occurrence of "--".

5.4.3 Parse translations

So far we have pretended that there is a close enough relationship between concrete and abstract syntax to allow an automatic translation from one to the other using the constant names supplied with each production. In many cases this scheme is not powerful enough, especially for constructs involving variable bindings. Therefore each `extension` can associate a user-defined translation function with a constant name via the `parse_translation` field of type

```
(string * (term list -> term)) list
```

After the input string has been translated into a term according to the syntax definition, there is a second phase in which the term is translated using the user-supplied functions. Given a list `tab` of the above type, a term `t` is translated as follows. If `t` is not of the form `Const(s,T)$t1$...$tn`, then `t` is returned unchanged. Otherwise all `ti` are translated into `ti'`. Let `t' = Const(s,T)$t1'$...$tn'`. If there is no pair `(s,f)` in `tab`, return `t'`. Otherwise apply `f` to `[t1',...,tn']`. If that raises an exception, return `t'`, otherwise return the result.

Example 5.5 Isabelle represents the phrase $\forall x.P(x)$ internally by $\forall(\lambda x.P(x))$, where $\forall$ is a constant of type $(term \to form) \to form$. Assuming that *term* and *form* are represented by the typs `Aterm` and `Aform`, the concrete syntax can be given as

```
Mixfix("ALL _. _", ALLt, "ALL", _, _)
```

where `ALLt` is `[SId,Aform] ---> Aform`. Parsing `"ALL x. ..."` according to this syntax yields the term `Const("ALL",Adummy)$Free("x",Adummy)$t`, where `t` is the body of the quantified formula. What we need is the term `Const("All",Adummy)$Abs("x",Adummy,t')`, where `"All"` is an abstract syntax constant of type `(Aterm --> Aform) --> Aform` and `t'` is some "abstracted" version of `t`. Therefore we define a function

```
fun ALLtr[Free(s,T), t] = Const("All", Adummy) $
                          Abs(s, T, abstract_over(Free(s,T), t));
```

and associate it with `"ALL"` in this fragment of syntax extension:

```
{mixfix = [..., Mixfix("ALL _. _", ALLt, "ALL", _, _), ...],
 parse_translation = [..., ("ALL", ALLtr), ...], ...}
```

Remember that `Adummy` is replaced by the correct types at a later stage (see Section 5.5).

5.4.4 Logical types and default syntax

Isabelle is concerned with mathematical languages which have a certain minimal vocabulary: identifiers, variables, parentheses, and the lambda calculus. Syntactic categories which allow all these phrases are called *logical types* to distinguish them from auxiliary categories like *id-list* which one would not expect to be able to parenthesize or abstract over. The `logical_types` field of an extension lists all logical types introduced in this extension. Logical types must be ground, i.e., of the form `Ground(_)`.

For any logical type A the following productions are added by default:

A	=	*identifier*		
	|	*variable*		
	|	`(` A `)`		
	|	Any_m `(` Any { `,` Any } `)`		Application
	|	`%` *id-list* `.` Any	(0)	Abstraction
	|	A `$$` *type*		Type constraint
Any	=	A		

where

type	=	*identifier*		Base type
	|	`(` *type* `)`		
	|	$type_1$ `=>` *type*	(0)	Function type
id-list	=	*identifier* { *identifier* }		

Application binds very tightly, abstraction very loosely. *Any* is the type of all logical types.

The syntactic categories *Any* and *id-list* can be referred to via the ML identifiers `Any` and `id_list` of type `typ`, respectively.

The infix `$$` introduces a type constraint. The interpretation of a *type* is a `typ`: an identifier `s` denotes `Ground"s"`, `t1 => t2` denotes `T1 --> T2`, where `T1` and `T2` are the denotations of `t1` and `t2`. The phrase `s $$ t` produces the term `Const("_constrain",T-->T)$S`, where `S` is the term produced by `s` and `T` the `typ` denoted by `t`. Thus `S` is forced to have `typ T`. Type constraints disappear with type checking but are still visible for the translation functions.

5.4.5 Printing

Syntax definitions provide printing information in three distinct ways: through

- the syntax of the language (as used for parsing),
- pretty printing information, and
- print translation functions.

The bare mixfix declarations enable Isabelle to print terms, but the result will not necessarily be pretty and may look different from what you expected. To produce a pleasing layout, you need to read the following sections.

Printing with mixfix declarations

Let `t = Const(s,_)$t1$...$tn` be a term and `M = Mixfix(sy,_,s,_,_)` be a mixfix declaration where `sy` is of the form α_0`_`$\alpha_1 \ldots \alpha_{n-1}$`_`$\alpha_n$. Printing `t` according to `M` means printing the string $\alpha_0\beta_1\alpha_1 \ldots \alpha_{n-1}\beta_n\alpha_n$, where β_i is the result of printing `ti`.

Note that the system does *not* insert blanks. They should be part of the mixfix syntax if they are required to separate tokens or achieve a certain layout.

Pretty printing

In order to format the output, it is possible to include pretty printing directions as part of the mixfix syntax definition. Of course these directions are ignored during parsing and affect only printing. The characters "`(`", "`)`", and "`/`" are interpreted as meta-characters when found in a mixfix definition. Their meaning is

`(` Open a block. A sequence of digits following it is interpreted as the *indentation* of this block. It causes the output to be indented by n positions if a line break occurs within the block. If "`(`" is not followed by a digit, the indentation defaults to 0.

`)` Close a block.

`/` Allow a line break. White spaces immediately following "`/`" are not printed if the line is broken at this point.

Print translations

Since terms can be subject to a translation after parsing (see Section 5.4.3), there is a similar mechanism to translate them back before printing. Therefore each `extension` can associate a user-defined translation function with a constant name via its `print_translation` field of type

```
(string * (term -> term)) list
```

Including a pair `(s,f)` results in `f` being applied to any term with head `Const(s,_)` before printing it.

Example 5.6 In Example 5.5 we showed how to translate the concrete syntax for universal quantification into the proper internal form. The string `"ALL x. ..."` now parses as `Const("All", _)$Abs("x", _, _)`. If, however, the latter term is printed without translating it back, it would result in `"All(%x. ...)"`. Therefore the abstraction has to be turned back into a term that matches the concrete mixfix syntax:

```
fun Alltr(_ $ Abs(id,T,P)) =
      let val (id',P') = variant_abs(id,T,P)
      in Const("ALL",ALLt) $ Free(id',T) $ P' end;
```

The function `variant_abs`, a basic term manipulation function, replaces the bound variable `id` by a `Free` variable `id'` having a unique name. A term produced by `Alltr` can now be printed according to the concrete syntax

```
Mixfix("ALL _. _", ALLt, "ALL", _, _).
```

Notice that the application of `Alltr` fails if the second component of the argument is not an abstraction, but for example just a `Free` variable. This is intentional because it signals to the caller that the intended translation is inapplicable.

The syntax extension, including concrete syntax and both translation functions, would have the following form:

```
{mixfix =  [..., Mixfix("ALL _. _", ALLt, "ALL", _, _), ...],
 parse_translation = [..., ("ALL", ALLtr), ...],
 print_translation = [..., ("All", Alltr), ...], ...}
```

As during the parse translation process, the types attached to variables or constants during print translation are immaterial because printing is insensitive to types. This means that in the definition of function `Alltr` above, we could have written `Const("ALL",Adummy) $ Free(id',Adummy)`.

Printing a term

Let G be the set of all `Mixfix` declarations and T the set of all string-function pairs of print translations in the current syntax.

Terms are printed recursively.

- `Free(s,_)` is printed as `s`.
- `Var((s,i),_)` is printed as `s`, if `i` $= 0$ and `s` does not end with a digit, as `s` followed by `i`, if `i` $\neq 0$ and `s` does not end with a digit, and as `s.i`, if `s` ends with a digit. Thus the following cases can arise:

```
Var(("v",0),_)  Var(("v",7),_)  Var(("v5",0),_)
     "?v"            "?v7"           "?v5.0"
```

- `Abs(x,_,Abs(y,_,...Abs(z,_,b)...)`, where `b` is not an abstraction, is printed as `%u v ... w. t`, where `t` is the result of printing `b`, and `x`, `y` and `z` are replaced by `u`, `v` and `w`. The latter are new unique names.

- `Bound(i)` is printed as `B.i` [3] .
- The application `t = Const(s,_)$t1$...$tn` (where `n` may be 0!) is printed as follows:

  If there is a pair `(s,f)` in T, print `f(t)`. If this application raises an exception or there is no pair `(s,f)` in T, let S be the set of `sy` such that `Mixfix(sy,_,s,_,_)` is in G. If S is empty, `t` is printed as an application. Otherwise let `sy'` $\in S$ be a mixfix declaration with the maximal number of nonterminals, n. If `sy'` has n arguments, print `t` according to `sy'`, otherwise print it as an application.

  All other applications are printed as applications.

Printing a term `c$t1$...$tn` as an application means printing it as `s(s1,...,sn)`, where `si` is the result of printing `ti`. If `c` is a `Const`, `s` is its first argument. Other terms `c` are printed as described above.

The printer also inserts parentheses where they are necessary for reasons of priority.

5.4.6 Miscellaneous

In addition to the ML identifiers introduced so far, the syntax module provides the following functions:

```
read: syntax -> typ -> string -> term
```

Calling `read sy t s` parses `s` as a term of type `t` according to syntax `sy`. Valid types are `Aprop`, `Any`, all logical types (see Section 5.4.4), and all function types. If the input does not conform to the syntax, an error message is printed and exception `ERROR` is raised.

```
prin: syntax -> term -> unit
print_top_level: syntax -> term -> unit
```

Calling `prin sy tm` performs the inverse translation: it prints the term `tm` according to syntax `sy`. `print_top_level` is like `prin`, except that the output is written on a separate line.

```
print_syntax: syntax -> unit
```

prints a syntax in some readable format.

```
constants: mixfix list -> (string list * typ) list
```

Calling `constants(ml)` builds a list of all pairs `([s],T)` such that `Mixfix(_,T,s,_,_)`, `Delimfix(_,T,s)`, `Infixl(s,T)`, or `Infixr(s,T)` is in `ml`.

[3] The occurrence of such "loose" bound variables indicates that either you are trying to print a subterm of an abstraction, or there is something wrong with your print translations.

This table of typed constants can be used as part of the `ops` argument to `extend_theory` and `enrich_theory`, thus removing the need to declare many constants twice.

Syntax extensions often introduce constants which are never part of a term that the parser produces because they are removed by translation. A typical example is `"ALL"` in Example 5.5: in contrast to `"All"` it is not part of the logical language represented with this syntax. To avoid picking up those constants, `constants` only selects those strings which do *not* start with a blank. Therefore we should have used `" ALL"` in examples 5.5 and 5.6.

Syntactic building blocks

This section introduces predefined syntactic classes and utilities operating on them.

Since abstraction and quantification over lists of variables is very common, the category *id-list* (see Figure 5.1) is available as the `typ id_list`. For example the syntax of object-level universal quantification can be defined by

```
Mixfix("ALL _. _", [id_list,Aform]--->Aform, " ALL", _, _),
```

permitting phrases like `"ALL x y. P(x,y)"`. To facilitate translation to and from the abstract syntax, two additional functions are defined.

```
abs_list_tr: term * term * term -> term
```

translates concrete to abstract syntax. If `idl` is a term of type `id_list` representing a list of variables v_{n-1} ... v_0, then `abs_list_tr(const,idl,body)` yields `const$Abs(`v_{n-1}`,_, ...const$Abs(`v_0`,_,body')...)` where `body'` is `body` with every free occurrence of v_i replaced by `Bound(`i`)`.

Given the syntax for universal quantification introduced above, the parse translation function associated with `" ALL"` can be defined as

```
fun ALLtr[idl,body] = abs_list_tr(Const("All",Adummy), idl, body)
```

It replaces the function of the same name in Example 5.5.

```
abs_list_tr': term * (string*typ)list * term -> term
```

performs the reverse translation. If `vars` is a list of variable names and their types `[`$(v_{n-1}, T_{n-1}), \ldots, (v_0, T_0)$`]`, then `abs_list(const,vars,body)` evaluates to `const$varl$body'` where

- `varl` is a term of type `id_list` representing a list of variables u_{n-1} ... u_0,
- u_{n-1} ... u_0 is a list of distinct names not occurring in `body`, and
- `body'` is `body` with `Bound(`i`)` replaced by `Free(`u_i`,`T_i`)`.

In the case of universal quantification, the print translation is achieved by

```
fun Alltr(tm) = abs_list_tr'(Const(" ALL",Adummy),
                             strip_qnt_vars "All" tm,
                             strip_qnt_body "All" tm);
```

which replaces the function of the same name in Example 5.6. The functions

```
fun strip_qnt_body qnt (tm as Const(c,_)$Abs(_,_,t)) =
       if c=qnt then strip_qnt_body qnt t else tm
  | strip_qnt_body qnt tm = tm;

fun strip_qnt_vars qnt (Const(c,_)$Abs(a,T,t)) =
       if c=qnt then (a,T) :: strip_qnt_vars qnt t else []
  | strip_qnt_vars qnt tm  =  [];
```

are predefined.

5.4.7 Restrictions

In addition to the restrictions mentioned throughout the text, the following ones apply.

Constant names must not start with an underbar, e.g. `"_ALL"`.

Type names must not start with an underbar, e.g. `Ground "_int"`.

The parser is designed to work for any context-free grammar. If, however, an ambiguous grammar is used, and the input string has more than one parse, the parser selects an arbitrary one and does not issue a warning.

5.5 Identifiers, constants, and type inference

There is one final step in the translation from strings to terms that we have not covered yet. It explains how constants are distinguished from `Free`s and how `Free`s and `Var`s are typed. Both issues arise because `Free`s and `Var`s are not declared.

An identifier `f` that does not appear as a delimiter in the concrete syntax can be either a free variable or a constant from the abstract syntax. Since the parser knows only about some of the constants, namely those that appear in the concrete syntax description, it parses `"f"` as `Free("f",Adummy)`, where `Adummy` is a predefined dummy `typ`. Although the parser function `read` produces these very raw terms, all user interface level functions like `goal` type terms according to the given abstract syntax, say A. In a first step, every occurrence of `Free(s,_)` or `Const(s,_)` is replaced by `Const(s,T)`, provided there is a constant `s` of `typ T` in A. This means that identifiers are treated as `Free`s iff they are not declared in the abstract syntax. The types of the remaining `Free`s (and `Var`s) are inferred as in ML. If the resulting term still contains a polymorphic `typ`, an exception is raised: Isabelle does not (yet) support polymorphism. Type constraints as introduced in Section 5.4.4 can be used to remove unwanted ambiguities or polymorphism.

One peculiarity of the current type inference algorithm is that variables with the same name must have the same type, irrespective of whether they are free or bound. For example, take the first-order formula $f(x) = x \wedge (\forall f.\, f = f)$ where the constants $=$ and $\forall$ have type $term \to term \to form$ and $(term \to form) \to form$. The first conjunct forces $x : term$ and $f : term \to term$, the second one $f : term$. Although the two f's are distinct, they are required to have the same type, thus leading to a type clash in the above formula.

5.6 Putting it all together

Having discussed the individual building blocks of a logic definition, it remains to be shown how they fit together. In particular we need to say how an object-logic syntax is hooked up to the meta-logic. Since all theorems must conform to the syntax for *prop* (see Figure 5.1), that syntax has to be extended with the object-level syntax. Assume that the syntax of your object-logic defines a category *form* of formulae. These formulae can now appear in axioms and theorems wherever *prop* does if you add the production

$$prop \;=\; form.$$

More precisely, you need a coercion from formulae to propositions:

```
Mixfix("_", Aform --> Aprop, "Prop", [0], 5)
```

The constant `"Prop"` (the name is arbitrary) acts as an invisible coercion function.

One of the simplest nontrivial logics is *minimal logic* of implication. Its definition in Isabelle needs no advanced features but illustrates the overall mechanism quite nicely:

```
val Aform  = Ground "form";
val mixfix = [Mixfix("_", Aform --> Aprop, "Prop", [], 5),
              Infixr("-->", [Aform,Aform]--->Aform, 10)];
val impl_syn = extend pure {logical_types=[Aform], mixfix=mixfix,
                  parse_translation=[], print_translation=[]};
val impl_thy = enrich_theory pure_thy "Impl"
      (["form"], constants mixfix, impl_syn)
      [("K",  "P --> Q --> P"),
       ("S",  "(P --> Q --> R) --> (P --> Q) --> P --> R"),
       ("MP", "[| P --> Q; P |] ==> Q")];
```

You can now start to prove theorems in this logic:

```
> goal impl_thy "P --> P";
Level 0
P --> P
 1. P --> P
> by(resolve_tac [get_axiom impl_thy "MP"] 1);
Level 1
P --> P
 1. ?P --> P --> P
 2. ?P
> by(resolve_tac [get_axiom impl_thy "MP"] 1);
Level 2
P --> P
 1. ?P1 --> ?P --> P --> P
 2. ?P1
 3. ?P
> by(resolve_tac [get_axiom impl_thy "S"] 1);
Level 3
P --> P
 1. P --> ?Q2 --> P
 2. P --> ?Q2
> by(resolve_tac [get_axiom impl_thy "K"] 1);
Level 4
P --> P
 1. P --> ?Q2
> by(resolve_tac [get_axiom impl_thy "K"] 1);
Level 5
P --> P
No subgoals!
```

As you can see, this Hilbert-style formulation of minimal logic is easy to define but difficult to use. The following natural deduction formulation is far preferable:

```
val Nimpl_thy = enrich_theory pure_thy "Impl"
      (["form"], constants impl_ext, impl_syn)
      [("I-I",  "[| P ==> Q |] ==> P-->Q"),
       ("I-E", "[| P --> Q; P |] ==> Q")];
```

Note, however, that although the two systems are equivalent, this fact cannot be proved within Isabelle: `"S"` and `"K"` can be derived in `Nimpl_thy` (exercise!), but `"I-I"` cannot be derived in `impl_thy`. The reason is that `"I-I"` is only an *admissible* rule in `impl_thy`, something that can only be shown by induction over all possible proofs in `impl_thy`.

It is a very simple matter to extend minimal logic with falsity because no new syntax is involved:

```
val NF_thy = extend_theory Nimpl_thy "F Extension"
      ([], [(["False"], Aform)])
      [("F-E", "False ==> P")];
```

On the other hand, we may wish to introduce conjunction only:

```
val mixfix = [Mixfix("_", Aform --> Aprop, "Prop", [], 5),
              Infixr("&", [Aform,Aform]--->Aform, 30)];
val conj_syn = extend pure {logical_types=[Aform], mixfix=mixfix,
                  parse_translation=[], print_translation=[]};
val Nconj_thy = enrich_theory pure_thy "Conj"
      (["form"], constants mixfix, conj_syn)
      [("C-I",  "[| P; Q |] ==> P & Q"),
       ("C-El",  "P & Q ==> P"), ("C-Er",  "P & Q ==> Q")];
```

And if we want to have all three connectives together, we define:

```
val thy = merge_theories (NF_thy,Nconj_thy);
```

Now we can prove mixed theorems like

```
goal thy "P & False --> Q";
by(resolve_tac  [get_axiom thy "I-I"]  1);
by(resolve_tac  [get_axiom thy "F-E"]  1);
by(eresolve_tac [get_axiom thy "C-Er"] 1);
```

Try this as an exercise!

Appendix A

Internals and Obscurities

This Appendix is a tour through the Isabelle sources themselves, listing functions (and other identifiers) with their types and purpose.

A.1 Types and terms

Isabelle types and terms are those of the typed λ-calculus.

A.1.1 Basic declarations

Exceptions in Isabelle are mainly used to signal errors. An exception includes a string (the error message) and other data to identify the error.

```
exception TYPE of string * typ list * term list
```

Signals errors involving types and terms.

```
exception TERM_ERROR of string * term list
```

Signals errors involving terms.

The following ML identifiers denote the symbols of the meta-logic.
The type of propositions is `Aprop`.
The implication symbol is `implies`.
The term `all T` is the universal quantifier for type `T`.
The term `equals T` is the equality predicate for type `T`.

A.1.2 Operations

There are a number of basic functions on terms and types.

```
op ---> : typ list * typ -> typ
```

Given types $[\tau_1, \ldots, \tau_n]$ and τ, it forms the type $\tau_1 \to \cdots \to (\tau_n \to \tau)$.

```
loose_bnos: term -> int list
```

Calling `loose_bnos t` returns the list of all 'loose' bound variable references. In particular, `Bound 0` is loose unless it is enclosed in an abstraction. Similarly `Bound 1` is loose unless it is enclosed in at least two abstractions; if enclosed in just one, the list will contain the number 0. A well-formed term does not contain any loose variables.

Calling `type_of` t computes the type of the term t. Raises exception `TYPE` unless applications are well-typed.

```
op aconv: term*term -> bool
```

Calling t `aconv` u tests whether terms t and u are α-convertible: identical up to renaming of bound variables.

- Two constants, `Free`s, or `Var`s are α-convertible just when their names and types are equal. (Variables having the same name but different types are thus distinct. This confusing situation should be avoided!)
- Two bound variables are α-convertible just when they have the same number.
- Two abstractions are α-convertible just when their bodies are, and their bound variables have the same type.
- Two applications are α-convertible just when the corresponding subterms are.

```
incr_boundvars: int -> term -> term
```

This increments a term's 'loose' bound variables by a given offset, required when moving a subterm into a context where it is enclosed by a different number of lambdas.

```
abstract_over: term*term -> term
```

For abstracting a term over a subterm v: replaces every occurrence of v by a `Bound` variable with the correct index.

Calling `subst_bounds`$([u_{n-1}, \ldots, u_0],\ t)$ substitutes the u_i for loose bound variables in t. This achieves β-reduction of $u_{n-1} \cdots u_0$ into t, replacing `Bound i` with u_i. For $(\lambda xy.t)(u, v)$, the bound variable indices in t are $x : 1$ and $y : 0$. The appropriate call is `subst_bounds([v,u],t)`. Loose bound variables $\geq n$ are reduced by n to compensate for the disappearance of n lambdas.

```
subst_term: (term*term)list -> term -> term
```

Simultaneous substitution for atomic terms in a term. An atomic term is a constant or any kind of variable.

```
maxidx_of_term: term -> int
```

Computes the maximum index of all the `Var`s in a term. If there are no `Var`s, the result is -1.

```
term_match: (term*term)list * term*term -> (term*term)list
```

Calling `term_match(vts,t,u)` instantiates `Var`s in `t` to match it with `u`. The resulting list of variable/term pairs extends `vts`, which is typically empty. First-order pattern matching is used to implement meta-level rewriting.

A.1.3 The representation of object-rules

The module `Logic` contains operations concerned with inference — especially, for constructing and destructing terms that represent object-rules.

```
op occs: term*term -> bool
```

Does one term occur in the other? (This is a reflexive relation.)

```
add_term_vars: term*term list -> term list
```

Accumulates the `Var`s in the term, suppressing duplicates. The second argument should be the list of `Var`s found so far.

```
add_term_frees: term*term list -> term list
```

Accumulates the `Free`s in the term, suppressing duplicates. The second argument should be the list of `Free`s found so far.

```
mk_equals: term*term -> term
```

Given t and u makes the term $t \equiv u$.

```
dest_equals: term -> term*term
```

Given $t \equiv u$ returns the pair (t, u).

```
list_implies: term list * term -> term
```

Given the pair $([\phi_1, \ldots, \phi_m], \phi)$ makes the term $[\phi_1; \ldots; \phi_m] \Longrightarrow \phi$.

```
strip_imp_prems: term -> term list
```

Given $[\phi_1; \ldots; \phi_m] \Longrightarrow \phi$ returns the list $[\phi_1, \ldots, \phi_m]$.

```
strip_imp_concl: term -> term
```

Given $[\phi_1; \ldots; \phi_m] \Longrightarrow \phi$ returns the term ϕ.

```
list_equals: (term*term)list * term -> term
```

For adding flex-flex constraints to an object-rule. Given $([(t_1, u_1), \ldots, (t_k, u_k)], \phi)$, makes the term $[t_1 \equiv u_1; \ldots; t_k \equiv u_k] \Longrightarrow \phi$.

```
strip_equals: term -> (term*term) list * term
```

Given $[t_1 \equiv u_1; \ldots; t_k \equiv u_k] \Longrightarrow \phi$, returns $([(t_1, u_1), \ldots, (t_k, u_k)], \phi)$.

```
rule_of: (term*term)list * term list * term -> term
```

Makes an object-rule: given the triple

$$([(t_1, u_1), \ldots, (t_k, u_k)], [\phi_1, \ldots, \phi_m], \phi)$$

returns the term $[t_1 \equiv u_1; \ldots; t_k \equiv u_k; \phi_1; \ldots; \phi_m] \Longrightarrow \phi$

```
strip_horn: term -> (term*term)list * term list * term
```

Breaks an object-rule into its parts: given

$$[t_1 \equiv u_1; \ldots; t_k \equiv u_k; \phi_1; \ldots; \phi_m] \Longrightarrow \phi$$

returns the triple $([(t_k, u_k), \ldots, (t_1, u_1)], [\phi_1, \ldots, \phi_m], \phi)$.

```
strip_assums: term -> (term*int) list * (string*typ) list * term
```

Strips premises of a rule allowing a more general form, where $\bigwedge$ and $\Longrightarrow$ may be intermixed. This is typical of assumptions of a subgoal in natural deduction. Returns additional information about the number, names, and types of quantified variables. For more discussion of assumptions, see Section A.4.2.

```
strip_prems: int * term list * term -> term list * term
```

For finding premise (or subgoal) i: given the triple $(i, [], \phi_1; \ldots \phi_i \Longrightarrow \phi)$ it returns another triple, $(\phi_i, [\phi_{i-1}, \ldots, \phi_1], \phi)$, where ϕ need not be atomic. Raises an exception if i is out of range.

A.2 Higher-order unification

Unification is used in the resolution of object-rules. Since logics are formalized in the typed λ-calculus, Isabelle uses Huet's higher-order unification algorithm [4].

A.2.1 Sequences

The module `Sequence` declares a type of unbounded sequences by the usual closure idea [8, page 118]. Sequences are defined in terms of the type `option`, declared in Isabelle's basic library, which handles the possible presence of a value.

```
datatype 'a option = None  |  Some of 'a;
```

Operations on the type `'a seq` include conversion between lists and sequences (with truncation), concatenation, and mapping a function over a sequence. Sequences are used in unification and tactics. The module `Sequence`, which is normally closed, declares the following.

```
type 'a seq
```

The type of (possibly unbounded) sequences of type `'a`.

```
seqof: (unit -> ('a * 'a seq) option) -> 'a seq
```

Calling `seqof (fn()=> Some(x,s))` constructs the sequence with head `x` and tail `s`, neither of which is evaluated.

```
null: 'a seq
```

This is `seqof (fn()=> None)`, the empty sequence.

```
single: 'a -> 'a seq
```

This is `seqof (fn()=> Some(x,null))`; makes a 1-element sequence.

```
pull: 'a seq -> ('a * 'a seq) option
```

Calling `pull s` returns `None` if the sequence is empty and `Some(x,s')` if the sequence has head `x` and tail `s'`. Only now is `x` evaluated. (Calling `pull s` again will *recompute* this value! It is not stored!)

```
append: 'a seq * 'a seq -> 'a seq
```

Concatenates two sequences.

```
flats: 'a seq seq -> 'a seq
```

Concatenates a sequence of sequences.

```
maps: ('a -> 'b) -> 'a seq -> 'b seq
```

Applies a function to every element of a sequence, producing a new sequence.

A.2.2 Environments

The module `Envir` (which is normally closed) declares a type of environments. An environment holds variable assignments and the next index to use when generating a variable.

```
datatype env = Envir of {asol: term xolist, maxidx: int}
```

The operations of lookup, update, and generation of variables are used during unification.

```
empty: int->env
```

Creates the environment with no assignments and the given index.

```
lookup: env * indexname -> term option
```

Looks up a variable, specified by its indexname, and returns `None` or `Some` as appropriate.

```
update: (indexname * term) * env -> env
```

Given a variable, term, and environment, produces *a new environment* where the variable has been updated. This has no side effect on the given environment.

```
genvar: env * typ -> env * term
```

Generates a variable of the given type and returns it, paired with a new environment (with incremented `maxidx` field).

```
alist_of: env -> (indexname * term) list
```

Converts an environment into an association list containing the assignments.

```
norm_term: env -> term -> term
```

Copies a term, following assignments in the environment, and performing all possible β-reductions.

```
rewrite: (env * (term*term)list) -> term -> term
```

Rewrites a term using the given term pairs as rewrite rules. Assignments are ignored; the environment is used only with `genvar`, to generate unique `Var`s as placeholders for bound variables.

A.2.3 The unification functions

The module `Unify` implements unification itself. It uses depth-first search with a depth limit that can be set. You can also switch tracing on and off, and specify a print function for tracing.

```
search_bound: int ref
```

Default 20, holds the depth limit for the unification search. The message

```
***Unification bound exceeded
```

appears whenever the search is cut off. This usually means the search would otherwise run forever, but a few proofs require increasing the default value of `search_bound`.

```
printer: (term->unit) ref
```

This function is used to print terms during tracing. It should be set to an object-logic's function `prin`. The default is a dummy that prints nothing.

```
trace_bound: int ref
```

Default 10, tracing information is printed whenever the search depth exceeds this bound.

```
trace_simp: bool ref
```

Default false, controls whether tracing information should include the SIMPL phase of unification. Otherwise only MATCH is traced.

```
unifiers: env * ((term*term)list) -> (env * (term*term)list) seq
```

This is the main unification function. Given an environment and a list of disagreement pairs, it returns a sequence of outcomes. Each outcome consists of an updated environment and a list of flex-flex pairs (these are discussed below).

```
smash_unifiers: env * (term*term)list -> env seq
```

This unification function maps an environment and a list of disagreement pairs to a sequence of updated environments. The function obliterates flex-flex pairs by choosing the obvious unifier. It may be used to tidy up any flex-flex pairs remaining at the end of a proof.

Flexible-flexible disagreement pairs

A *flexible-flexible* disagreement pair is one where the heads of both terms are variables. Every set of flex-flex pairs has an obvious unifier and usually many others. The function `unifiers` returns the flex-flex pairs to constrain later unifications; `smash_unifiers` uses the obvious unifier to eliminate flex-flex pairs.

For example, the many unifiers of $?f(0) \equiv ?g(0)$ include $?f \mapsto \lambda x.?g(0)$ and $\{?f \mapsto \lambda x.x,\ ?g \mapsto \lambda x.0\}$. The trivial unifier, which introduces a new variable $?a$, is $\{?f \mapsto \lambda x.?a,\ ?g \mapsto \lambda x.?a\}$. Of these three unifiers, none is an instance of another.

Flex-flex pairs are simplified to eliminate redundant bound variables, as shown in the following example:

$$\lambda xy.?f(k(y), l(x)) \equiv \lambda xy.?g(y)$$

The bound variable x is not used in the right-hand term. Any unifier of these terms must delete all occurrences of x on the left. Choosing a new variable $?h$, the assignment $?f \mapsto \lambda uv.?h(u)$ reduces the disagreement pair to

$$\lambda xy.?h(k(y)) \equiv \lambda xy.?g(y)$$

without losing any unifiers. Now x can be dropped on both sides (adjusting bound variable indices) to leave

$$\lambda y.?h(k(y)) \equiv \lambda y.?g(y)$$

Assigning $?g \mapsto \lambda y.?h(k(y))$ eliminates $?g$ and unifies both terms to $\lambda y.?h(k(y))$.

Multiple unifiers

Higher-order unification can generate an unbounded sequence of unifiers. Multiple unifiers indicate ambiguity; usually the source of the ambiguity is obvious. Some unifiers are more natural than others. In solving $?f(a) \equiv a + b - a$, the solution $?f \mapsto \lambda x.x+b-x$ is better than $?f \mapsto \lambda x.a+b-a$ because it reveals the dependence of $a + b - a$ on a. There are four unifiers in this case. Isabelle generates the better ones first by preferring *projection* over *imitation*.

The unification procedure performs Huet's MATCH operation [4] in big steps. It solves $?f(t_1, \ldots, t_p) \equiv u$ for $?f$ by finding all ways of copying u, first trying projection on the arguments t_i. It never copies below any variable in u; instead it returns a new variable, resulting in a flex-flex disagreement pair. If it encounters $?f$ in u, it allows projection only. This prevents looping in some obvious cases, but can be fooled by cycles involving several disagreement pairs. It is also incomplete.

Associative unification

Associative unification comes for free: encoded through function composition, an associative operation [5, page 37]. To represent lists, let C be a new constant. The empty list is $\lambda x.x$, while the list $[t_1, t_2, \ldots, t_n]$ is represented by the term

$$\lambda x.C(t_1, C(t_2, \ldots, C(t_n, x)))$$

The unifiers of this with $\lambda x.?f(?g(x))$ give all the ways of expressing $[t_1, t_2, \ldots, t_n]$ as the concatenation of two lists.

Unlike standard associative unification, this technique can represent certain infinite sets of unifiers as finite sets containing flex-flex disagreement pairs. But $\lambda x.C(t, ?a)$ does not represent any list. Such garbage terms may appear in flex-flex pairs and accumulate dramatically.

Associative unification handles sequent calculus rules, where the comma is the associative operator:

$$\frac{\Gamma, A, B, \Delta \vdash \Theta}{\Gamma, A \,\&\, B, \Delta \vdash \Theta}$$

Multiple unifiers occur whenever this is resolved against a goal containing more than one conjunction on the left. Note that we do not really need associative unification, only associative *matching*.

A.3 Terms valid under a signature

The module `Sign` declares the abstract types of signatures and checked terms. A signature contains the syntactic information needed for building a theory. A checked term is simply a term that has been checked to conform with a given signature, and is packaged together with its type, etc.

A.3.1 The ML type sg

A signature lists all *ground types* that may appear in terms in the theory. The *lexical symbol table* declares each constant, infix, variable, and keyword. The *syntax* contains the theory's notation in some internal format; this concerns users but has no logical meaning.

```
type sg =
  {gnd_types: string list, (*ground types*)
   const_tab: typ Symtab.table, (*types of constants*)
   ident_tab: typ Symtab.table, (*default types of identifiers*)
   syn: Syntax.syntax, (*Parsing and printing operations*)
   stamps: string ref list (*unique theory identifier*)  };
```

The *stamps* identify the theory. Each primitive theory has a single stamp. When the union of theories is taken, the lists of stamps are merged. References are used as the unique identifiers. The references are compared, not their contents.

Two signatures can be combined into a new one provided their constants are compatible. If an identifier is used as a constant in both signatures, it must be the same kind of symbol and have the same type in both signatures. This union operation should be idempotent, commutative, and associative. You can build signatures that ought to be the same but have different syntax functions, since functions cannot be compared.

A.3.2 The ML type cterm

A term t is *valid* under a signature provided every type in t is declared in the signature and every constant in t is declared as a constant or infix of the same type in the signature. It must be well-typed and monomorphic and must not have loose bound variables. Note that a subterm of a valid term need *not* be valid: it may contain loose bound variables. Even if $\lambda x.x$ is valid, its subterm x is a loose bound variable.

A checked term is stored with its signature, type, and maximum index of its `Var`s. This information is computed during the checks.

```
datatype cterm = Cterm of {sign: sg,
                           t: term,
                           T: typ,
                           maxidx: int};
```

The inference rules maintain that the terms that make up a theorem are valid under the theorem's signature. Rules (like specialization) that operate on terms take them as `cterm`s rather than taking raw terms and checking them. It is possible to obtain cterms from theorems, saving the effort of checking the terms again.

A.3.3 Declarations

Here are the most important declarations of the module `Sign`. (This module is normally closed.)

```
type sg
```

The type of signatures, this is an abstract type: the constructor is not exported. A signature can only be created by calling `new`, typically via a call to `prim_theory`.

```
type cterm
```

The abstract type of checked terms. A cterm can be created by calling `cterm_of` or `read_cterm`.

```
rep_sg: sg -> {gnd_types: string list, lextab: lexsy table...}
```

The representation function for type `sg`, this returns the underlying record.

```
new: string -> ... -> sg
```

Calling `new signame (gnd_types,lextab,parser,printer)` creates a new signature named `signame` from the given type names, lexical table, parser, and printing functions.

```
rep_cterm: cterm -> {T: typ, maxidx: int, sign: sg, t: term}
```

The representation function for type `cterm`.

```
term_of: cterm -> term
```

Maps a cterm to the underlying term.

```
cterm_of: sg -> term -> cterm
```

Given a signature and term, checks that the term is valid in the signature and produces the corresponding cterm. Raises exception `TERM_ERROR` with the message 'type error in term' or 'term not in signature' if appropriate.

```
read_cterm: sg -> string*typ -> cterm
```

Reads a string as a term using the parsing information in the signature. It checks that this term is valid to produce a cterm. Note that a type must be supplied: this aids type inference considerably. Intended for interactive use, `read_cterm` catches the various exceptions that could arise and prints error messages. Commands like `goal` call `read_cterm`.

```
print_cterm: cterm -> unit
```

Prints the cterm using the printing function in its signature.

```
print_term: sg -> term -> unit
```

Prints the term using the printing function in the given signature.

```
type_assign: cterm -> cterm
```

Produces a cterm by updating the signature of its argument to include all variable/type assignments. Type inference under the resulting signature will assume the same type assignments as in the argument. This is used in the goal package to give persistence to type assignments within each proof. (Contrast with LCF's sticky types [8, page 148].)

A.4 Meta-inference

Theorems and theories are mutually recursive. Each theorem is associated with a theory; each theory contains axioms, which are theorems. To avoid circularity, a theorem contains a signature rather than a theory.

The module `Thm` declares theorems, theories, and all meta-rules. Together with `Sign` this module is critical to Isabelle's correctness: all other modules call on them to construct valid terms and theorems.

A.4.1 Theorems

The natural deduction system for the meta-logic [9] is represented by the obvious sequent calculus. Theorems have the form $\Phi \vdash \psi$, where Φ is the set of hypotheses and ψ is a proposition. Each meta-theorem has a signature and stores the maximum index of all the `Var`s in ψ.

```
datatype thm = Thm of
    {sign: Sign.sg,
     maxidx: int,
     hyps: term list,
     prop: term};
```

The proof state with subgoals $\phi_1, \ldots, \phi_m$ and main goal ϕ is represented by the object-rule $\phi_1 \ldots \phi_m / \phi$, which in turn is represented by the meta-theorem

$$\Phi \vdash [t_1 \equiv u_1; \ldots; t_k \equiv u_k; \phi_1; \ldots; \phi_m] \Longrightarrow \phi \qquad \text{(A.1)}$$

The field `hyps` holds Φ, the set of meta-level assumptions. The field `prop` holds the entire proposition, $[t_1 \equiv u_1; \ldots; \phi_m] \Longrightarrow \phi$, which can be further broken down. The function `tpairs_of` returns the (t, u) pairs, while `prems_of` returns the ϕ_i and `concl_of` returns ϕ.

```
exception THM of string * int * thm list
```

Signals incorrect arguments to meta-rules. The tuple consists of a message, a premise number, and the premises.

```
rep_thm: thm -> {prop: term, hyps: term list, ...}
```

This function returns the representation of a theorem, the underlying record.

```
tpairs_of: thm -> (term*term)list
```

Maps the theorem (A.1) to the list of flex-flex constraints, $[(t_1, u_1), \ldots, (t_k, u_k)]$.

```
prems_of: thm -> term list
```

Maps the theorem (A.1) to the premises, $[\phi_1, \ldots, \phi_m]$.

```
concl_of: thm -> term
```

Maps the theorem (A.1) to the conclusion, ϕ.

Meta-rules

All of the meta-rules are implemented (though not all are used). They raise exception `THM` to signal malformed premises, incompatible signatures and similar errors.

```
assume: Sign.cterm -> thm
```

Makes the sequent $\psi \vdash \psi$, checking that ψ contains no `Var`s. Recall that `Var`s are only allowed in the conclusion.

```
implies_intr: Sign.cterm -> thm -> thm
```

This is $\Longrightarrow$-introduction.

```
implies_elim: thm -> thm -> thm
```

This is $\Longrightarrow$-elimination.

```
forall_intr: Sign.cterm -> thm -> thm
```

The $\bigwedge$-introduction rule generalizes over a variable, either `Free` or `Var`. The variable must not be free in the hypotheses; if it is a `Var` then there is nothing to check.

```
forall_elim: Sign.cterm -> thm -> thm
```

This is $\bigwedge$-elimination.

```
reflexive: Sign.cterm -> thm
```

Reflexivity of equality.

```
symmetric: thm -> thm
```

Symmetry of equality.

```
transitive: thm -> thm -> thm
```

Transitivity of equality.

```
instantiate: (Sign.cterm*Sign.cterm) list -> thm -> thm
```

Simultaneous substitution of terms for distinct `Var`s. The result is not normalized.

Definitions

An axiom of the form $C \equiv t$ defines the constant C as the term t. Rewriting with the axiom $C \equiv t$ *unfolds* the constant C: replaces C by t. Rewriting with the theorem $t \equiv C$ (obtained by the rule `symmetric`) *folds* the constant C: replaces t by C. Several rules are concerned with definitions.

```
rewrite_rule: thm list -> thm -> thm
```

This uses a list of equality theorems to rewrite another theorem. Rewriting is left-to-right and continues until no rewrites are applicable to any subterm.

```
rewrite_goals_rule: thm list -> thm -> thm
```

This uses a list of equality theorems to rewrite just the antecedents of another theorem — typically a proof state. This unfolds definitions in the subgoals but not in the main goal.

Unfolding should only be needed for proving basic theorems about a defined symbol. Later proofs should treat the symbol as a primitive. For example, in first-order logic, bi-implication is defined in terms of implication and conjunction:

```
P<->Q == (P-->Q) & (Q-->P)
```

After deriving basic rules for this connective, we can forget its definition.

This treatment of definitions should be contrasted with many other theorem provers, where defined symbols are automatically unfolded.

A.4.2 Derived meta-rules for backwards proof

The following rules, coded directly in ML for efficiency, handle backwards proof. They typically involve a proof state

$$[\psi_1; \ldots; \psi_i; \ldots; \psi_n] \Longrightarrow \psi$$

Subgoal i, namely ψ_i, might have the form

$$\bigwedge x_1.\theta_1 \Longrightarrow \cdots (\bigwedge x_k.\theta_k \Longrightarrow \theta)$$

Each θ_j may be preceded by zero or more quantifiers, whose scope extends to θ. The θ_j represent the *assumptions* of the subgoal; the x_j represent the *parameters*.

The object-rule $[\phi_1; \ldots; \phi_m] \Longrightarrow \phi$ is lifted over the assumptions and parameters of the subgoal and renumbered [9]; write the lifted object-rule as

$$[\widetilde{\phi}_1; \ldots; \widetilde{\phi}_m] \Longrightarrow \widetilde{\phi}$$

Recall that, for example, $\widetilde{\phi}$ has the form

$$\bigwedge x_1.\theta_1; \ldots; (\bigwedge x_k.\theta_k \Longrightarrow \phi')$$

where ϕ' is obtained from ϕ by replacing its free variables by certain terms.

Each rule raises exception `THM` if subgoal i does not exist.

```
resolution: thm * int * thm list -> thm Sequence.seq
```

Calling `resolution(state,i,rules)` performs higher-order resolution of a theorem in `rules`, typically an object-rule, with subgoal i of the proof state held in the theorem `state`. The sequence of theorems contains the result of each successful unification of $\widetilde{\phi} \equiv \psi_i$, replacing ψ_i by $\widetilde{\phi}_1$, ..., $\widetilde{\phi}_m$ and instantiating variables in `state`. The rules are used in order.

```
assumption: thm * int -> thm Sequence.seq
```

Calling `assumption(state,i)` attempts to solve subgoal i by assumption in natural deduction. The call tries each unification of the form $\theta_j \equiv \theta$ for $j = 1, \ldots, k$. The sequence of theorems contains the outcome of each successful unification, where ψ_i has been deleted and variables may have been instantiated elsewhere in the state.

```
biresolution: thm * int * (bool*thm)list -> thm Sequence.seq
```

Calling `biresolution(state,i,brules)` is like calling `resolution`, except that pairs of the form `(true,rule)` involve an implicit call of `assumption`. This permits using natural deduction object-rules in a sequent style, where the 'principal formula' is deleted after use. Write the lifted object-rule as

$$[\widetilde{\phi}_1; \widetilde{\phi}_2; \ldots; \widetilde{\phi}_m] \Longrightarrow \widetilde{\phi}$$

The rule is interpreted as an elimination rule with ϕ_1 as the major premise, and `biresolution` will insist on finding ϕ_1 among the assumptions of subgoal i. The call tries each unification of the form $\{\phi'_1 \equiv \theta_j, \widetilde{\phi} \equiv \psi_i\}$ for $j = 1, \ldots, k$. The sequence of theorems contains the result of each successful unification, replacing ψ_i by the $m-1$ subgoals $\widetilde{\phi}_2$, ..., $\widetilde{\phi}_m$ and instantiating variables in `state`. The relevant assumption is deleted in the subgoals: if unification involved θ_j, then the occurrence of θ_j is deleted in each of $\widetilde{\phi}_2$, ..., $\widetilde{\phi}_m$.

Pairs of the form `(false,rule)` are treated exactly as in `resolution`. The pairs are used in order.

```
trivial: Sign.cterm -> thm
```

Makes the theorem $\psi \Longrightarrow \psi$, used as the initial state in a backwards proof. The proposition ψ may contain `Var`s.

A.5 Tactics and tacticals

Here are some of the more obscure tactics, tacticals, derived meta-rules, and other items that are available.

A.5.1 Derived rules

The following derived rules are implemented using primitive rules.

```
forall_intr_frees: thm -> thm
```

Generalizes a meta-theorem over all `Free` variables not free in hypotheses.

```
forall_elim_vars: int -> thm -> thm
```

Replaces all outermost quantified variables by `Var`s of a given index.

```
zero_var_indexes: thm -> thm
```

Replaces all `Var`s by `Var`s having index 0, preserving distinctness by renaming when necessary.

```
standard: thm -> thm
```

Puts a meta-theorem into standard form: no hypotheses, `Free` variables, or outer quantifiers. All generality is expressed by `Var`s having index 0.

```
resolve: thm * int * thm list -> thm
```

Calling `resolve (rlb,i,rules)` tries each of the `rules`, in turn, resolving them with premise i of `rlb`. Raises exception `THM` unless resolution produces exactly one result. This function can be used to paste object-rules together, making simple derived rules.

```
op RES: thm * thm -> thm
```

Calling `(rule2 RES rule1)` is equivalent to `resolve(rule2,1,[rule1])`; it resolves the conclusion of `rule1` with premise 1 in `rule2`. Raises exception `THM` unless there is exactly one unifier.

A.5.2 Tactics

The following tactics are mainly intended for use in proof procedures that process lots of rules. However, `res_inst_tac` can be used even in single-step proofs to keep control over unification.

```
biresolve_tac: (bool*thm) list -> int -> tactic
```

This is analogous to `resolve_tac` but calls `biresolution`. It is thus suitable for a mixture of introduction rules (which should be paired with `false`) and elimination rules (which should be paired with `true`).

```
res_inst_tac: (string*string*typ)list -> thm -> int -> tactic
```

For selective instantiation of variables, `res_inst_tac [(v,t,T)] rule i` reads the strings `v` and `t`, obtaining a variable $?v$ and term t of the given type `T`. The term may refer to parameters of subgoal i, for the tactic modifies $?v$ and t to compensate for the parameters, and lifts the `rule` over the assumptions of the subgoal. The tactic replaces $?v$ by t in the rule and finally resolves it against the subgoal.

```
smash_all_ff_tac: tactic
```

Eliminates all flex-flex constraints from the proof state by applying the trivial higher-order unifier.

A.5.3 Filtering of object-rules

Higher-order resolution involving a long list of rules is slow. Filtering techniques can shorten the list of rules given to resolution. A second use of filtering is to detect whether resolving against a given subgoal would cause excessive branching. If too many rules are applicable then another subgoal might be selected.

The module `Stringtree` implements a data structure for fast selection of rules. A term is classified by its *head string*: the string of symbols obtained by following the first argument in function calls until a `Var` is encountered. For instance, the Constructive Type Theory judgement

```
rec(succ(?d),0,?b): N
```

has the head string `["Elem","rec","succ"]` where the constant `Elem` represents the elementhood judgement form $a \in A$.

Two head strings are *compatible* if they are equal or if one is a prefix of the other. If two terms have incompatible head strings, then they are clearly not unifiable. A theorem is classified by the head string of its conclusion, indicating which goals it could resolve with. This method is fast, easy to implement, and powerful.

Head strings can only discriminate terms according to their first arguments. Type Theory introduction rules have conclusions like `0:N` and `inl(?a):?A+?B`. Because the type is the second argument, the head string does not discriminate by types.

```
could_unify: term*term->bool
```

This function quickly detects nonunifiable terms. It assumes all variables are distinct, reporting that `?a=?a` may unify with `0=1`.

```
filt_resolve_tac: thm list -> int -> int -> tactic
```

Calling `filt_resolve_tac rules maxr i` uses `could_resolve` to discover which of the `rules` are applicable to subgoal `i`. If this number exceeds `maxr` then the tactic fails (returning the null sequence). Otherwise it behaves like `resolve_tac`.

```
compat_resolve_tac: thm list -> int -> int -> tactic
```

Calling `compat_resolve_tac rules maxr i` builds a stringtree from the `rules` to discover which of them are applicable to subgoal `i`. If this number exceeds `maxr` then the tactic fails (returning the null sequence). Otherwise it behaves like `resolve_tac`. (To avoid repeated construction of the same stringtree, apply `compat_resolve_tac` to a list of rules and bind the result to an identifier.)

Bibliography

[1] N. G. de Bruijn, Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser Theorem, *Indagationes Mathematicae* **34** (1972), pages 381–392.

[2] K. Futatsugi, J.A. Goguen, J.-P. Jouannaud, J. Meseguer, Principles of OBJ2, *12th ACM Symposium on Principles of Programming Languages* (1985), pages 52–66.

[3] J. H. Gallier, *Logic for Computer Science: Foundations of Automatic Theorem Proving* (Harper & Row, 1986).

[4] G. P. Huet, A unification algorithm for typed λ-calculus, *Theoretical Computer Science* **1** (1975), pages 27–57.

[5] G. P. Huet, B. Lang, Proving and applying program transformations expressed with second-order patterns, *Acta Informatica* **11** (1978), pages 31–55.

[6] P. Martin-Löf, *Intuitionistic Type Theory* (Bibliopolis, 1984).

[7] L. C. Paulson, Natural deduction as higher-order resolution, *Journal of Logic Programming* **3** (1986), pages 237–258.

[8] L. C. Paulson, *Logic and Computation: Interactive Proof with Cambridge LCF* (Cambridge University Press, 1987).

[9] L. C. Paulson, The foundation of a generic theorem prover, *Journal of Automated Reasoning* **5** (1989), 363–397.

[10] L. C. Paulson, Isabelle: The next 700 theorem provers, *In*: P. Odifreddi (editor), *Logic and Computer Science* (Academic Press, 1990, in press), 361–385.

[11] F. J. Pelletier, Seventy-five problems for testing automatic theorem provers, *Journal of Automated Reasoning* **2** (1986), pages 191–216.

[12] P. Suppes, *Axiomatic Set Theory* (Dover Publications, 1972).

[13] G. Takeuti, *Proof Theory* (2nd Edition) (North Holland, 1987).

[14] A. N. Whitehead and B. Russell, *Principia Mathematica* (Paperback edition to *56, Cambridge University Press, 1962).

[15] Å. Wikström, *Functional Programming Using Standard ML* (Prentice-Hall, 1987).

Index