Introduction to Isabelle

Lawrence C. Paulson
Computer Laboratory
University of Cambridge
`lcp@cl.cam.ac.uk`

With Contributions by Tobias Nipkow and Markus Wenzel

28 August 1998

Preface

Isabelle [10, 12, 13] is a generic theorem prover. It has been instantiated to support reasoning in several object-logics:

- first-order logic, constructive and classical versions
- higher-order logic, similar to that of Gordon's HOL [2]
- Zermelo-Fraenkel set theory [18]
- an extensional version of Martin-Löf's Type Theory [9]
- the classical first-order sequent calculus, LK
- the modal logics T, $S4$, and $S43$
- the Logic for Computable Functions [11]

A logic's syntax and inference rules are specified declaratively; this allows single-step proof construction. Isabelle provides control structures for expressing search procedures. Isabelle also provides several generic tools, such as simplifiers and classical theorem provers, which can be applied to object-logics.

Isabelle is a large system, but beginners can get by with a small repertoire of commands and a basic knowledge of how Isabelle works. Some knowledge of Standard ML is essential, because ML is Isabelle's user interface. Advanced Isabelle theorem proving can involve writing ML code, possibly with Isabelle's sources at hand. My book on ML [14] covers much material connected with Isabelle, including a simple theorem prover. Users must be familiar with logic as used in computer science; there are many good texts [1, 17].

LCF, developed by Robin Milner and colleagues [3], is an ancestor of HOL, Nuprl, and several other systems. Isabelle borrows ideas from LCF: formulae are ML values; theorems belong to an abstract type; tactics and tacticals support backward proof. But LCF represents object-level rules by functions, while Isabelle represents them by terms. You may find my other writings [11, 15] helpful in understanding the relationship between LCF and Isabelle.

Isabelle was first distributed in 1986. The 1987 version introduced a higher-order meta-logic with an improved treatment of quantifiers. The 1988 version added limited polymorphism and support for natural deduction. The 1989 version included a parser and pretty printer generator. The 1992 version introduced type classes, to support many-sorted and higher-order logics. The current version provides greater support for theories and is much faster. Isabelle is still under development and will continue to change.

Overview

This manual consists of three parts. Part I discusses the Isabelle's foundations. Part II, presents simple on-line sessions, starting with forward proof. It also covers basic tactics and tacticals, and some commands for invoking them. Part III contains further examples for users with a bit of experience. It explains how to derive rules define theories, and concludes with an extended example: a Prolog interpreter.

Isabelle's Reference Manual and Object-Logics manual contain more details. They assume familiarity with the concepts presented here.

Acknowledgements

Tobias Nipkow contributed most of the section on defining theories. Stefan Berghofer and Sara Kalvala suggested improvements.

Tobias Nipkow has made immense contributions to Isabelle, including the parser generator, type classes, and the simplifier. Carsten Clasohm and Markus Wenzel made major contributions; Sonia Mahjoub and Karin Nimmermann also helped. Isabelle was developed using Dave Matthews's Standard ML compiler, Poly/ML. Many people have contributed to Isabelle's standard object-logics, including Martin Coen, Philippe de Groote, Philippe Noël. The research has been funded by the EPSRC (grants GR/G53279, GR/H40570, GR/K57381, GR/K77051) and by ESPRIT (projects 3245: Logical Frameworks, and 6453: Types).

Contents

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

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

Part I
Foundations

The following sections discuss Isabelle's logical foundations in detail: representing logical syntax in the typed λ-calculus; expressing inference rules in Isabelle's meta-logic; combining rules by resolution.

If you wish to use Isabelle immediately, please turn to page 24. You can always read about foundations later, either by returning to this point or by looking up particular items in the index.

1 Formalizing logical syntax in Isabelle

Figure 1 presents intuitionistic first-order logic, including equality. Let us see how to formalize this logic in Isabelle, illustrating the main features of Isabelle's polymorphic meta-logic.

Isabelle represents syntax using the simply typed λ-calculus. We declare a type for each syntactic category of the logic. We declare a constant for each symbol of the logic, giving each n-place operation an n-argument curried function type. Most importantly, λ-abstraction represents variable binding in quantifiers.

Isabelle has ML-style polymorphic types such as $(\alpha)list$, where $list$ is a type constructor and α is a type variable; for example, $(bool)list$ is the type of lists of booleans. Function types have the form $(\sigma,\tau)fun$ or $\sigma \Rightarrow \tau$, where σ and τ are types. Curried function types may be abbreviated:

$$\sigma_1 \Rightarrow (\cdots \sigma_n \Rightarrow \tau \cdots) \quad \text{as} \quad [\sigma_1,\ldots,\sigma_n] \Rightarrow \tau$$

The syntax for terms is summarised below. Note that there are two versions of function application syntax available in Isabelle: either $t\,u$, which is the usual form for higher-order languages, or $t(u)$, trying to look more like first-order. The latter syntax is used throughout the manual.

$t :: \tau$	type constraint, on a term or bound variable
$\lambda x\,.\,t$	abstraction
$\lambda x_1 \ldots x_n\,.\,t$	curried abstraction, $\lambda x_1\,.\,\ldots\lambda x_n\,.\,t$
$t(u)$	application
$t(u_1,\ldots,u_n)$	curried application, $t(u_1)\ldots(u_n)$

1.1 Simple types and constants

The syntactic categories of our logic (Fig. 1) are **formulae** and **terms**. Formulae denote truth values, so (following tradition) let us call their type o. To allow 0

$$\begin{array}{rcl} \neg P & \text{abbreviates} & P \to \bot \\ P \leftrightarrow Q & \text{abbreviates} & (P \to Q) \land (Q \to P) \end{array}$$

$$\frac{P \quad Q}{P \land Q}\,(\land I) \qquad \frac{P \land Q}{P}\,(\land E1) \qquad \frac{P \land Q}{Q}\,(\land E2)$$

$$\frac{P}{P \lor Q}\,(\lor I1) \qquad \frac{Q}{P \lor Q}\,(\lor I2) \qquad \frac{P \lor Q \quad \begin{matrix}[P] \\ \vdots \\ R\end{matrix} \quad \begin{matrix}[Q] \\ \vdots \\ R\end{matrix}}{R}\,(\lor E)$$

$$\frac{\begin{matrix}[P] \\ \vdots \\ Q\end{matrix}}{P \to Q}\,(\to I) \qquad \frac{P \to Q \quad P}{Q}\,(\to E)$$

$$\frac{\bot}{P}\,(\bot E)$$

$$\frac{P}{\forall x\,.\,P}\,(\forall I)* \qquad \frac{\forall x\,.\,P}{P[t/x]}\,(\forall E)$$

$$\frac{P[t/x]}{\exists x\,.\,P}\,(\exists I) \qquad \frac{\exists x\,.\,P \quad \begin{matrix}[P] \\ \vdots \\ Q\end{matrix}}{Q}\,(\exists E)*$$

$$t = t\,(\mathit{refl}) \qquad \frac{t = u \quad P[t/x]}{P[u/x]}\,(\mathit{subst})$$

$*$*Eigenvariable conditions*:
$\forall I$: provided x is not free in the assumptions
$\exists E$: provided x is not free in Q or any assumption except P

Figure 1: Intuitionistic first-order logic

and $Suc(t)$ as terms, let us declare a type *nat* of natural numbers. Later, we shall see how to admit terms of other types.

After declaring the types o and *nat*, we may declare constants for the symbols of our logic. Since $\bot$ denotes a truth value (falsity) and 0 denotes a number, we put

$$\begin{array}{rcl} \bot & :: & o \\ 0 & :: & nat. \end{array}$$

If a symbol requires operands, the corresponding constant must have a function type. In our logic, the successor function (*Suc*) is from natural numbers to natural numbers, negation ($\neg$) is a function from truth values to truth values, and the binary connectives are curried functions taking two truth values as arguments:

$$\begin{array}{rcl} Suc & :: & nat \Rightarrow nat \\ \neg & :: & o \Rightarrow o \\ \wedge, \vee, \rightarrow, \leftrightarrow & :: & [o, o] \Rightarrow o \end{array}$$

The binary connectives can be declared as infixes, with appropriate precedences, so that we write $P \wedge Q \vee R$ instead of $\vee(\wedge(P, Q), R)$.

Section 11 below describes the syntax of Isabelle theory files and illustrates it by extending our logic with mathematical induction.

1.2 Polymorphic types and constants

Which type should we assign to the equality symbol? If we tried $[nat, nat] \Rightarrow o$, then equality would be restricted to the natural numbers; we should have to declare different equality symbols for each type. Isabelle's type system is polymorphic, so we could declare

$$= \;::\; [\alpha, \alpha] \Rightarrow o,$$

where the type variable α ranges over all types. But this is also wrong. The declaration is too polymorphic; α includes types like o and $nat \Rightarrow nat$. Thus, it admits $\bot = \neg(\bot)$ and $Suc = Suc$ as formulae, which is acceptable in higher-order logic but not in first-order logic.

Isabelle's **type classes** control polymorphism [8]. Each type variable belongs to a class, which denotes a set of types. Classes are partially ordered by the subclass relation, which is essentially the subset relation on the sets of types. They closely resemble the classes of the functional language Haskell [4, 5].

Isabelle provides the built-in class *logic*, which consists of the logical types: the ones we want to reason about. Let us declare a class *term*, to consist of all legal types of terms in our logic. The subclass structure is now $term \leq logic$.

We put *nat* in class *term* by declaring $nat{::}term$. We declare the equality constant by

$$= \;::\; [\alpha{::}term, \alpha] \Rightarrow o$$

where $\alpha{::}term$ constrains the type variable α to class *term*. Such type variables resemble Standard ML's equality type variables.

We give o and function types the class *logic* rather than *term*, since they are not legal types for terms. We may introduce new types of class *term* — for instance, type *string* or *real* — at any time. We can even declare type constructors such as *list*, and state that type $(\tau)list$ belongs to class *term* provided τ does; equality applies to lists of natural numbers but not to lists of formulae. We may summarize this paragraph by a set of **arity declarations** for type constructors:

$$\begin{array}{rcl} o & :: & logic \\ fun & :: & (logic, logic)logic \\ nat, string, real & :: & term \\ list & :: & (term)term \end{array}$$

(Recall that *fun* is the type constructor for function types.) In higher-order logic, equality does apply to truth values and functions; this requires the arity declarations $o :: term$ and $fun :: (term, term)term$. The class system can also handle overloading. We could declare *arith* to be the subclass of *term* consisting of the 'arithmetic' types, such as *nat*. Then we could declare the operators

$$+, -, \times, / \;::\; [\alpha{::}arith, \alpha] \Rightarrow \alpha$$

If we declare new types *real* and *complex* of class *arith*, then we in effect have three sets of operators:

$$\begin{array}{rcl} +, -, \times, / & :: & [nat, nat] \Rightarrow nat \\ +, -, \times, / & :: & [real, real] \Rightarrow real \\ +, -, \times, / & :: & [complex, complex] \Rightarrow complex \end{array}$$

Isabelle will regard these as distinct constants, each of which can be defined separately. We could even introduce the type $(\alpha)vector$ and declare its arity as $(arith)arith$. Then we could declare the constant

$$+ \;::\; [(\alpha)vector, (\alpha)vector] \Rightarrow (\alpha)vector$$

and specify it in terms of $+ :: [\alpha, \alpha] \Rightarrow \alpha$.

A type variable may belong to any finite number of classes. Suppose that we had declared yet another class $ord \leq term$, the class of all 'ordered' types, and a constant

$$\leq \;::\; [\alpha{::}ord, \alpha] \Rightarrow o.$$

In this context the variable x in $x \leq (x + x)$ will be assigned type $\alpha{::}\{arith, ord\}$, which means α belongs to both *arith* and *ord*. Semantically the set $\{arith, ord\}$ should be understood as the intersection of the sets of types represented by *arith* and *ord*. Such intersections of classes are called **sorts**. The empty intersection of classes, $\{\}$, contains all types and is thus the **universal sort**.

Even with overloading, each term has a unique, most general type. For this to be possible, the class and type declarations must satisfy certain technical constraints; see Sect. Defining Theories in the *Reference Manual*.

1.3 Higher types and quantifiers

Quantifiers are regarded as operations upon functions. Ignoring polymorphism for the moment, consider the formula $\forall x \,.\, P(x)$, where x ranges over type *nat*. This is true if $P(x)$ is true for all x. Abstracting $P(x)$ into a function, this is the same as saying that $\lambda x \,.\, P(x)$ returns true for all arguments. Thus, the universal quantifier can be represented by a constant

$$\forall \quad :: \quad (nat \Rightarrow o) \Rightarrow o,$$

which is essentially an infinitary truth table. The representation of $\forall x \,.\, P(x)$ is $\forall(\lambda x \,.\, P(x))$.

The existential quantifier is treated in the same way. Other binding operators are also easily handled; for instance, the summation operator $\Sigma_{k=i}^{j} f(k)$ can be represented as $\Sigma(i, j, \lambda k \,.\, f(k))$, where

$$\Sigma \quad :: \quad [nat, nat, nat \Rightarrow nat] \Rightarrow nat.$$

Quantifiers may be polymorphic. We may define $\forall$ and $\exists$ over all legal types of terms, not just the natural numbers, and allow summations over all arithmetic types:

$$\begin{aligned} \forall, \exists \quad &:: \quad (\alpha{::}term \Rightarrow o) \Rightarrow o \\ \Sigma \quad &:: \quad [nat, nat, nat \Rightarrow \alpha{::}arith] \Rightarrow \alpha \end{aligned}$$

Observe that the index variables still have type *nat*, while the values being summed may belong to any arithmetic type.

2 Formalizing logical rules in Isabelle

Object-logics are formalized by extending Isabelle's meta-logic [12], which is intuitionistic higher-order logic. The meta-level connectives are **implication**, the **universal quantifier**, 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', for expressing **definitions** (see §10.2). Equalities left over from the unification process, so called **flex-flex constraints**, are written $a \stackrel{?}{\equiv} b$. The two equality symbols have the same logical meaning.

The syntax of the meta-logic is formalized in the same manner as object-logics, using the simply typed λ-calculus. Analogous to type o above, there is a built-in type *prop* of meta-level truth values. Meta-level formulae will have this type. Type *prop* belongs to class *logic*; also, $\sigma \Rightarrow \tau$ belongs to *logic* provided σ and τ do. Here are the types of the built-in connectives:

$$\begin{array}{rcl}
\Longrightarrow & :: & [\mathit{prop}, \mathit{prop}] \Rightarrow \mathit{prop} \\
\bigwedge & :: & (\alpha{::}\mathit{logic} \Rightarrow \mathit{prop}) \Rightarrow \mathit{prop} \\
\equiv & :: & [\alpha{::}\{\}, \alpha] \Rightarrow \mathit{prop} \\
\stackrel{?}{\equiv} & :: & [\alpha{::}\{\}, \alpha] \Rightarrow \mathit{prop}
\end{array}$$

The polymorphism in $\bigwedge$ is restricted to class *logic* to exclude certain types, those used just for parsing. The type variable $\alpha{::}\{\}$ ranges over the universal sort.

In our formalization of first-order logic, we declared a type o of object-level truth values, rather than using *prop* for this purpose. If we declared the object-level connectives to have types such as $\neg :: \mathit{prop} \Rightarrow \mathit{prop}$, then these connectives would be applicable to meta-level formulae. Keeping *prop* and o as separate types maintains the distinction between the meta-level and the object-level. To formalize the inference rules, we shall need to relate the two levels; accordingly, we declare the constant

$$\mathit{Trueprop} \quad :: \quad o \Rightarrow \mathit{prop}.$$

We may regard *Trueprop* as a meta-level predicate, reading $\mathit{Trueprop}(P)$ as 'P is true at the object-level.' Put another way, *Trueprop* is a coercion from o to *prop*.

2.1 Expressing propositional rules

We shall illustrate the use of the meta-logic by formalizing the rules of Fig. 1. Each object-level rule is expressed as a meta-level axiom.

One of the simplest rules is $(\wedge E1)$. Making everything explicit, its formalization in the meta-logic is

$$\bigwedge P\ Q \,.\, \mathit{Trueprop}(P \wedge Q) \Longrightarrow \mathit{Trueprop}(P). \qquad (\wedge E1)$$

This may look formidable, but it has an obvious reading: for all object-level truth values P and Q, if $P \land Q$ is true then so is P. The reading is correct because the meta-logic has simple models, where types denote sets and $\bigwedge$ really means 'for all.'

Isabelle adopts notational conventions to ease the writing of rules. We may hide the occurrences of *Trueprop* by making it an implicit coercion. Outer universal quantifiers may be dropped. Finally, the nested implication

$$\phi_1 \Longrightarrow (\cdots \phi_n \Longrightarrow \psi \cdots)$$

may be abbreviated as $[\![\phi_1; \ldots; \phi_n]\!] \Longrightarrow \psi$, which formalizes a rule of n premises.

Using these conventions, the conjunction rules become the following axioms. These fully specify the properties of $\land$:

$$[\![P; Q]\!] \Longrightarrow P \land Q \qquad (\land I)$$

$$P \land Q \Longrightarrow P \qquad P \land Q \Longrightarrow Q \qquad (\land E1, 2)$$

Next, consider the disjunction rules. The discharge of assumption in $(\lor E)$ is expressed using $\Longrightarrow$:

$$P \Longrightarrow P \lor Q \qquad Q \Longrightarrow P \lor Q \qquad (\lor I1, 2)$$

$$[\![P \lor Q; P \Longrightarrow R; Q \Longrightarrow R]\!] \Longrightarrow R \qquad (\lor E)$$

To understand this treatment of assumptions in natural deduction, look at implication. The rule $(\to I)$ is the classic example of natural deduction: to prove that $P \to Q$ is true, assume P is true and show that Q must then be true. More concisely, if P implies Q (at the meta-level), then $P \to Q$ is true (at the object-level). Showing the coercion explicitly, this is formalized as

$$(\mathit{Trueprop}(P) \Longrightarrow \mathit{Trueprop}(Q)) \Longrightarrow \mathit{Trueprop}(P \to Q).$$

The rule $(\to E)$ is straightforward; hiding *Trueprop*, the axioms to specify $\to$ are

$$(P \Longrightarrow Q) \Longrightarrow P \to Q \qquad (\to I)$$

$$[\![P \to Q; P]\!] \Longrightarrow Q. \qquad (\to E)$$

Finally, the intuitionistic contradiction rule is formalized as the axiom

$$\bot \Longrightarrow P. \qquad (\bot E)$$

! Earlier versions of Isabelle, and certain papers [12, 13], use $[\![P]\!]$ to mean $\mathit{Trueprop}(P)$.

2.2 Quantifier rules and substitution

Isabelle expresses variable binding using λ-abstraction; for instance, $\forall x\,.\,P$ is formalized as $\forall(\lambda x\,.\,P)$. Recall that $F(t)$ is Isabelle's syntax for application of the function F to the argument t; it is not a meta-notation for substitution. On the other hand, a substitution will take place if F has the form $\lambda x\,.\,P$; Isabelle transforms $(\lambda x\,.\,P)(t)$ to $P[t/x]$ by β-conversion. Thus, we can express inference rules that involve substitution for bound variables.

A logic may attach provisos to certain of its rules, especially quantifier rules. We cannot hope to formalize arbitrary provisos. Fortunately, those typical of quantifier rules always have the same form, namely 'x not free in ... *(some set of formulae)*,' where x is a variable (called a **parameter** or **eigenvariable**) in some premise. Isabelle treats provisos using $\bigwedge$, its inbuilt notion of 'for all'.

The purpose of the proviso 'x not free in ... ' is to ensure that the premise may not make assumptions about the value of x, and therefore holds for all x. We formalize $(\forall I)$ by

$$\left(\bigwedge x\,.\,\mathit{Trueprop}(P(x))\right) \Longrightarrow \mathit{Trueprop}(\forall x\,.\,P(x)).$$

This means, 'if $P(x)$ is true for all x, then $\forall x\,.\,P(x)$ is true.' The $\forall E$ rule exploits β-conversion. Hiding *Trueprop*, the $\forall$ axioms are

$$\left(\bigwedge x\,.\,P(x)\right) \Longrightarrow \forall x\,.\,P(x) \tag{$\forall I$}$$

$$(\forall x\,.\,P(x)) \Longrightarrow P(t). \tag{$\forall E$}$$

We have defined the object-level universal quantifier ($\forall$) using $\bigwedge$. But we do not require meta-level counterparts of all the connectives of the object-logic! Consider the existential quantifier:

$$P(t) \Longrightarrow \exists x\,.\,P(x) \tag{$\exists I$}$$

$$[\![\exists x\,.\,P(x);\ \bigwedge x\,.\,P(x) \Longrightarrow Q]\!] \Longrightarrow Q \tag{$\exists E$}$$

Let us verify $(\exists E)$ semantically. Suppose that the premises hold; since $\exists x\,.\,P(x)$ is true, we may choose an a such that $P(a)$ is true. Instantiating $\bigwedge x\,.\,P(x) \Longrightarrow Q$ with a yields $P(a) \Longrightarrow Q$, and we obtain the desired conclusion, Q.

The treatment of substitution deserves mention. The rule

$$\frac{t = u \quad P}{P[u/t]}$$

would be hard to formalize in Isabelle. It calls for replacing t by u throughout P, which cannot be expressed using β-conversion. Our rule (*subst*) uses P as a template for substitution, inferring $P[u/x]$ from $P[t/x]$. When we formalize this as an axiom, the template becomes a function variable:

$$[\![t = u; P(t)]\!] \Longrightarrow P(u). \tag{subst}$$

2.3 Signatures and theories

A **signature** contains the information necessary for type checking, parsing and pretty printing a term. It specifies type classes and their relationships, types and their arities, constants and their types, etc. It also contains grammar rules, specified using mixfix declarations.

Two signatures can be merged provided their specifications are compatible — they must not, for example, assign different types to the same constant. Under similar conditions, a signature can be extended. Signatures are managed internally by Isabelle; users seldom encounter them.

A **theory** consists of a signature plus a collection of axioms. The `Pure` theory contains only the meta-logic. Theories can be combined provided their signatures are compatible. A theory definition extends an existing theory with further signature specifications — classes, types, constants and mixfix declarations — plus lists of axioms and definitions etc., expressed as strings to be parsed. A theory can formalize a small piece of mathematics, such as lists and their operations, or an entire logic. A mathematical development typically involves many theories in a hierarchy. For example, the `Pure` theory could be extended to form a theory for Fig. 1; this could be extended in two separate ways to form a theory for natural numbers and a theory for lists; the union of these two could be extended into a theory defining the length of a list:

$$
\begin{tikzcd}
 & \mathtt{Pure} \arrow[d] & \\
 & \mathtt{FOL} \arrow[dl] \arrow[dr] & \\
\mathtt{Nat} \arrow[dr] & & \mathtt{List} \arrow[dl] \\
 & \mathtt{Nat} + \mathtt{List} \arrow[d] & \\
 & \mathtt{Length} &
\end{tikzcd}
$$

Each Isabelle proof typically works within a single theory, which is associated with the proof state. However, many different theories may coexist at the same time, and you may work in each of these during a single session.

! Confusing problems arise if you work in the wrong theory. Each theory defines its own syntax. An identifier may be regarded in one theory as a constant and in another as a variable, for example.

3 Proof construction in Isabelle

I have elsewhere described the meta-logic and demonstrated it by formalizing first-order logic [12]. There is a one-to-one correspondence between meta-level proofs and object-level proofs. To each use of a meta-level axiom, such as $(\forall I)$, there is a use of the corresponding object-level rule. Object-level assumptions and parameters have meta-level counterparts. The meta-level formalization is **faithful**, admitting no incorrect object-level inferences, and **adequate**, admitting all correct object-level inferences. These properties must be demonstrated separately for each object-logic.

The meta-logic is defined by a collection of inference rules, including equational rules for the λ-calculus and logical rules. The rules for $\Longrightarrow$ and $\bigwedge$ resemble those for $\rightarrow$ and $\forall$ in Fig. 1. Proofs performed using the primitive meta-rules would be lengthy; Isabelle proofs normally use certain derived rules. **Resolution**, in particular, is convenient for backward proof.

Unification is central to theorem proving. It supports quantifier reasoning by allowing certain 'unknown' terms to be instantiated later, possibly in stages. When proving that the time required to sort n integers is proportional to n^2, we need not state the constant of proportionality; when proving that a hardware adder will deliver the sum of its inputs, we need not state how many clock ticks will be required. Such quantities often emerge from the proof.

Isabelle provides **schematic variables**, or **unknowns**, for unification. Logically, unknowns are free variables. But while ordinary variables remain fixed, unification may instantiate unknowns. Unknowns are written with a ? prefix and are frequently subscripted: $?a$, $?a_1$, $?a_2$, $\ldots$, $?P$, $?P_1$, $\ldots$.

Recall that an inference rule of the form

$$\frac{\phi_1 \quad \ldots \quad \phi_n}{\phi}$$

is formalized in Isabelle's meta-logic as the axiom $[\![\phi_1; \ldots; \phi_n]\!] \Longrightarrow \phi$. Such axioms resemble Prolog's Horn clauses, and can be combined by resolution — Isabelle's principal proof method. Resolution yields both forward and backward proof. Backward proof works by unifying a goal with the conclusion of a rule, whose premises become new subgoals. Forward proof works by unifying theorems with the premises of a rule, deriving a new theorem.

Isabelle formulae require an extended notion of resolution. They differ from Horn clauses in two major respects:

- They are written in the typed λ-calculus, and therefore must be resolved using higher-order unification.
- The constituents of a clause need not be atomic formulae. Any formula of the form $\mathit{Trueprop}(\cdots)$ is atomic, but axioms such as $\rightarrow I$ and $\forall I$ contain non-atomic formulae.

Isabelle has little in common with classical resolution theorem provers such as Otter [19]. At the meta-level, Isabelle proves theorems in their positive form, not by refutation. However, an object-logic that includes a contradiction rule may employ a refutation proof procedure.

3.1 Higher-order unification

Unification is equation solving. The solution of $f(?x, c) \stackrel{?}{\equiv} f(d, ?y)$ is $?x \equiv d$ and $?y \equiv c$. **Higher-order unification** is equation solving for typed λ-terms. To handle β-conversion, it must reduce $(\lambda x\,.\,t)u$ to $t[u/x]$. That is easy — in the typed λ-calculus, all reduction sequences terminate at a normal form. But it must guess the unknown function $?f$ in order to solve the equation

$$?f(t) \stackrel{?}{\equiv} g(u_1, \ldots, u_k). \tag{1}$$

Huet's [6] search procedure solves equations by imitation and projection. **Imitation** makes $?f$ apply the leading symbol (if a constant) of the right-hand side. To solve equation (1), it guesses

$$?f \equiv \lambda x\,.\,g(?h_1(x), \ldots, ?h_k(x)),$$

where $?h_1$, …, $?h_k$ are new unknowns. Assuming there are no other occurrences of $?f$, equation (1) simplifies to the set of equations

$$?h_1(t) \stackrel{?}{\equiv} u_1 \quad \ldots \quad ?h_k(t) \stackrel{?}{\equiv} u_k.$$

If the procedure solves these equations, instantiating $?h_1$, …, $?h_k$, then it yields an instantiation for $?f$.

Projection makes $?f$ apply one of its arguments. To solve equation (1), if t expects m arguments and delivers a result of suitable type, it guesses

$$?f \equiv \lambda x\,.\,x(?h_1(x), \ldots, ?h_m(x)),$$

where $?h_1$, …, $?h_m$ are new unknowns. Assuming there are no other occurrences of $?f$, equation (1) simplifies to the equation

$$t(?h_1(t), \ldots, ?h_m(t)) \stackrel{?}{\equiv} g(u_1, \ldots, u_k).$$

! Huet's unification procedure is complete. Isabelle's polymorphic version, which solves for type unknowns as well as for term unknowns, is incomplete. The problem is that projection requires type information. In equation (1), if the type of t is unknown, then projections are possible for all $m \geq 0$, and the types of the $?h_i$ will be similarly unconstrained. Therefore, Isabelle never attempts such projections, and may fail to find unifiers where a type unknown turns out to be a function type.

Given $?f(t_1, \ldots, t_n) \stackrel{?}{\equiv} u$, Huet's procedure could make up to $n + 1$ guesses. The search tree and set of unifiers may be infinite. But higher-order unification can work effectively, provided you are careful with **function unknowns**:

- Equations with no function unknowns are solved using first-order unification, extended to treat bound variables. For example, $\lambda x\,.\,x \stackrel{?}{\equiv} \lambda x\,.\,?y$ has no solution because $?y \equiv x$ would capture the free variable x.
- An occurrence of the term $?f(x, y, z)$, where the arguments are distinct bound variables, causes no difficulties. Its projections can only match the corresponding variables.
- Even an equation such as $?f(a) \stackrel{?}{\equiv} a + a$ is all right. It has four solutions, but Isabelle evaluates them lazily, trying projection before imitation. The first solution is usually the one desired:

$$?f \equiv \lambda x\,.\,x + x \quad ?f \equiv \lambda x\,.\,a + x \quad ?f \equiv \lambda x\,.\,x + a \quad ?f \equiv \lambda x\,.\,a + a$$

- Equations such as $?f(?x, ?y) \stackrel{?}{\equiv} t$ and $?f(?g(x)) \stackrel{?}{\equiv} t$ admit vast numbers of unifiers, and must be avoided.

In problematic cases, you may have to instantiate some unknowns before invoking unification.

3.2 Joining rules by resolution

Let $[\![\psi_1; \ldots; \psi_m]\!] \Longrightarrow \psi$ and $[\![\phi_1; \ldots; \phi_n]\!] \Longrightarrow \phi$ be two Isabelle theorems, representing object-level rules. Choosing some i from 1 to n, suppose that ψ and ϕ_i have a higher-order unifier. Writing Xs for the application of substitution s to expression X, this means there is some s such that $\psi s \equiv \phi_i s$. By resolution, we may conclude

$$([\![\phi_1; \ldots; \phi_{i-1}; \psi_1; \ldots; \psi_m; \phi_{i+1}; \ldots; \phi_n]\!] \Longrightarrow \phi)s.$$

The substitution s may instantiate unknowns in both rules. In short, resolution is the following rule:

$$\frac{[\![\psi_1; \ldots; \psi_m]\!] \Longrightarrow \psi \qquad [\![\phi_1; \ldots; \phi_n]\!] \Longrightarrow \phi}{([\![\phi_1; \ldots; \phi_{i-1}; \psi_1; \ldots; \psi_m; \phi_{i+1}; \ldots; \phi_n]\!] \Longrightarrow \phi)s} \ (\psi s \equiv \phi_i s)$$

It operates at the meta-level, on Isabelle theorems, and is justified by the properties of $\Longrightarrow$ and $\bigwedge$. It takes the number i (for $1 \le i \le n$) as a parameter and may yield infinitely many conclusions, one for each unifier of ψ with ϕ_i. Isabelle returns these conclusions as a sequence (lazy list).

Resolution expects the rules to have no outer quantifiers ($\bigwedge$). It may rename or instantiate any schematic variables, but leaves free variables unchanged. When constructing a theory, Isabelle puts the rules into a standard form with all free variables converted into schematic ones; for instance, $(\rightarrow E)$ becomes

$$[\![?P \rightarrow ?Q; ?P]\!] \Longrightarrow ?Q.$$

When resolving two rules, the unknowns in the first rule are renamed, by subscripting, to make them distinct from the unknowns in the second rule. To resolve $(\rightarrow E)$ with itself, the first copy of the rule becomes

$$[\![?P_1 \rightarrow ?Q_1; ?P_1]\!] \Longrightarrow ?Q_1.$$

Resolving this with $(\rightarrow E)$ in the first premise, unifying $?Q_1$ with $?P \rightarrow ?Q$, is the meta-level inference

$$\frac{[\![?P_1 \rightarrow ?Q_1; ?P_1]\!] \Longrightarrow ?Q_1 \qquad [\![?P \rightarrow ?Q; ?P]\!] \Longrightarrow ?Q}{[\![?P_1 \rightarrow (?P \rightarrow ?Q); ?P_1; ?P]\!] \Longrightarrow ?Q.}$$

Renaming the unknowns in the resolvent, we have derived the object-level rule

$$\frac{R \rightarrow (P \rightarrow Q) \quad R \quad P}{Q.}$$

Joining rules in this fashion is a simple way of proving theorems. The derived rules are conservative extensions of the object-logic, and may permit simpler proofs. Let us consider another example. Suppose we have the axiom

$$\forall x\, y\, .\, Suc(x) = Suc(y) \rightarrow x = y. \qquad (inject)$$

The standard form of $(\forall E)$ is $\forall x\, .\, ?P(x) \Longrightarrow ?P(?t)$. Resolving $(inject)$ with $(\forall E)$ replaces $?P$ by $\lambda x\, .\, \forall y\, .\, Suc(x) = Suc(y) \rightarrow x = y$ and leaves $?t$ unchanged, yielding

$$\forall y\, .\, Suc(?t) = Suc(y) \rightarrow ?t = y.$$

Resolving this with $(\forall E)$ puts a subscript on $?t$ and yields

$$Suc(?t_1) = Suc(?t) \rightarrow ?t_1 = ?t.$$

Resolving this with $(\rightarrow E)$ increases the subscripts and yields

$$Suc(?t_2) = Suc(?t_1) \Longrightarrow ?t_2 = ?t_1.$$

We have derived the rule

$$\frac{Suc(m) = Suc(n)}{m = n,}$$

which goes directly from $Suc(m) = Suc(n)$ to $m = n$. It is handy for simplifying an equation like $Suc(Suc(Suc(m))) = Suc(Suc(Suc(0)))$.

4 Lifting a rule into a context

The rules $(\to I)$ and $(\forall I)$ may seem unsuitable for resolution. They have non-atomic premises, namely $P \Longrightarrow Q$ and $\bigwedge x\,.\,P(x)$, while the conclusions of all the rules are atomic (they have the form $\mathit{Trueprop}(\cdots)$). Isabelle gets round the problem through a meta-inference called **lifting**. Let us consider how to construct proofs such as

$$\dfrac{\dfrac{\begin{matrix}[P, Q] \\ \vdots \\ R\end{matrix}}{Q \to R}\,(\to I)}{P \to (Q \to R)}\,(\to I) \qquad \dfrac{\dfrac{P(x,y)}{\forall y\,.\,P(x,y)}\,(\forall I)}{\forall x\,y\,.\,P(x,y)}\,(\forall I)$$

4.1 Lifting over assumptions

Lifting over $\theta \Longrightarrow$ is the following meta-inference rule:

$$\frac{[\![\phi_1; \ldots; \phi_n]\!] \Longrightarrow \phi}{[\![\theta \Longrightarrow \phi_1; \ldots; \theta \Longrightarrow \phi_n]\!] \Longrightarrow (\theta \Longrightarrow \phi)}$$

This is clearly sound: if $[\![\phi_1; \ldots; \phi_n]\!] \Longrightarrow \phi$ is true and $\theta \Longrightarrow \phi_1, \ldots, \theta \Longrightarrow \phi_n$ and θ are all true then ϕ must be true. Iterated lifting over a series of meta-formulae $\theta_k, \ldots, \theta_1$ yields an object-rule whose conclusion is $[\![\theta_1; \ldots; \theta_k]\!] \Longrightarrow \phi$. Typically the θ_i are the assumptions in a natural deduction proof; lifting copies them into a rule's premises and conclusion.

When resolving two rules, Isabelle lifts the first one if necessary. The standard form of $(\to I)$ is

$$(?P \Longrightarrow ?Q) \Longrightarrow ?P \to ?Q.$$

To resolve this rule with itself, Isabelle modifies one copy as follows: it renames the unknowns to $?P_1$ and $?Q_1$, then lifts the rule over $?P \Longrightarrow$ to obtain

$$(?P \Longrightarrow (?P_1 \Longrightarrow ?Q_1)) \Longrightarrow (?P \Longrightarrow (?P_1 \to ?Q_1)).$$

Using the $[\![\cdots]\!]$ abbreviation, this can be written as

$$[\![[\![?P; ?P_1]\!] \Longrightarrow ?Q_1; ?P]\!] \Longrightarrow ?P_1 \to ?Q_1.$$

Unifying $?P \Longrightarrow ?P_1 \to ?Q_1$ with $?P \Longrightarrow ?Q$ instantiates $?Q$ to $?P_1 \to ?Q_1$. Resolution yields

$$([\![?P; ?P_1]\!] \Longrightarrow ?Q_1) \Longrightarrow ?P \to (?P_1 \to ?Q_1).$$

This represents the derived rule

$$\frac{\begin{matrix}[P, Q] \\ \vdots \\ R\end{matrix}}{P \to (Q \to R).}$$

4.2 Lifting over parameters

An analogous form of lifting handles premises of the form $\bigwedge x \ldots$. Here, lifting prefixes an object-rule's premises and conclusion with $\bigwedge x$. At the same time, lifting introduces a dependence upon x. It replaces each unknown $?a$ in the rule by $?a'(x)$, where $?a'$ is a new unknown (by subscripting) of suitable type — necessarily a function type. In short, lifting is the meta-inference

$$\frac{[\![\phi_1; \ldots; \phi_n]\!] \Longrightarrow \phi}{[\![\bigwedge x . \phi_1^x; \ldots; \bigwedge x . \phi_n^x]\!] \Longrightarrow \bigwedge x . \phi^x},$$

where ϕ^x stands for the result of lifting unknowns over x in ϕ. It is not hard to verify that this meta-inference is sound. If $\phi \Longrightarrow \psi$ then $\phi^x \Longrightarrow \psi^x$ for all x; so if ϕ^x is true for all x then so is ψ^x. Thus, from $\phi \Longrightarrow \psi$ we conclude $(\bigwedge x . \phi^x) \Longrightarrow (\bigwedge x . \psi^x)$.

For example, $(\vee I)$ might be lifted to

$$(\bigwedge x . ?P_1(x)) \Longrightarrow (\bigwedge x . ?P_1(x) \vee ?Q_1(x))$$

and $(\forall I)$ to

$$(\bigwedge x\, y . ?P_1(x, y)) \Longrightarrow (\bigwedge x . \forall y . ?P_1(x, y)).$$

Isabelle has renamed a bound variable in $(\forall I)$ from x to y, avoiding a clash. Resolving the above with $(\forall I)$ is the meta-inference

$$\frac{(\bigwedge x\, y . ?P_1(x, y)) \Longrightarrow (\bigwedge x . \forall y . ?P_1(x, y)) \quad (\bigwedge x . ?P(x)) \Longrightarrow (\forall x . ?P(x))}{\bigwedge x\, y . ?P_1(x, y)) \Longrightarrow \forall x\, y . ?P_1(x, y))}$$

Here, $?P$ is replaced by $\lambda x . \forall y . ?P_1(x, y)$; the resolvent expresses the derived rule

$$\frac{Q(x, y)}{\forall x\, y . Q(x, y)} \quad \text{provided } x,\ y \text{ not free in the assumptions}$$

I discuss lifting and parameters at length elsewhere [12]. Miller goes into even greater detail [7].

5 Backward proof by resolution

Resolution is convenient for deriving simple rules and for reasoning forward from facts. It can also support backward proof, where we start with a goal and refine it to progressively simpler subgoals until all have been solved. LCF and its descendants HOL and Nuprl provide tactics and tacticals, which constitute a sophisticated language for expressing proof searches. **Tactics** refine subgoals while **tacticals** combine tactics.

Isabelle's tactics and tacticals work differently from LCF's. An Isabelle rule is bidirectional: there is no distinction between inputs and outputs. LCF has a separate tactic for each rule; Isabelle performs refinement by any rule in a uniform fashion, using resolution.

Isabelle works with meta-level theorems of the form $[\![\phi_1; \ldots; \phi_n]\!] \Longrightarrow \phi$. We have viewed this as the **rule** with premises $\phi_1, \ldots, \phi_n$ and conclusion ϕ. It can also be viewed as the **proof state** with subgoals $\phi_1, \ldots, \phi_n$ and main goal ϕ.

To prove the formula ϕ, take $\phi \Longrightarrow \phi$ as the initial proof state. This assertion is, trivially, a theorem. At a later stage in the backward proof, a typical proof state is $[\![\phi_1; \ldots; \phi_n]\!] \Longrightarrow \phi$. This proof state is a theorem, ensuring that the subgoals $\phi_1, \ldots, \phi_n$ imply ϕ. If $n = 0$ then we have proved ϕ outright. If ϕ contains unknowns, they may become instantiated during the proof; a proof state may be $[\![\phi_1; \ldots; \phi_n]\!] \Longrightarrow \phi'$, where ϕ' is an instance of ϕ.

5.1 Refinement by resolution

To refine subgoal i of a proof state by a rule, perform the following resolution:

$$\frac{\text{rule} \qquad \text{proof state}}{\text{new proof state}}$$

Suppose the rule is $[\![\psi'_1; \ldots; \psi'_m]\!] \Longrightarrow \psi'$ after lifting over subgoal i's assumptions and parameters. If the proof state is $[\![\phi_1; \ldots; \phi_n]\!] \Longrightarrow \phi$, then the new proof state is (for $1 \leq i \leq n$)

$$([\![\phi_1; \ldots; \phi_{i-1}; \psi'_1; \ldots; \psi'_m; \phi_{i+1}; \ldots; \phi_n]\!] \Longrightarrow \phi)s.$$

Substitution s unifies ψ' with ϕ_i. In the proof state, subgoal i is replaced by m new subgoals, the rule's instantiated premises. If some of the rule's unknowns are left un-instantiated, they become new unknowns in the proof state. Refinement by $(\exists I)$, namely

$$?P(?t) \Longrightarrow \exists x \,.\, ?P(x),$$

inserts a new unknown derived from $?t$ by subscripting and lifting. We do not have to specify an 'existential witness' when applying $(\exists I)$. Further resolutions may instantiate unknowns in the proof state.

5.2 Proof by assumption

In the course of a natural deduction proof, parameters $x_1, \ldots, x_l$ and assumptions $\theta_1, \ldots, \theta_k$ accumulate, forming a context for each subgoal. Repeated lifting steps can lift a rule into any context. To aid readability, Isabelle puts contexts into a normal form, gathering the parameters at the front:

$$\bigwedge x_1 \ldots x_l \,.\, [\![\theta_1; \ldots; \theta_k]\!] \Longrightarrow \theta. \tag{2}$$

Under the usual reading of the connectives, this expresses that θ follows from $\theta_1, \ldots \theta_k$ for arbitrary $x_1, \ldots, x_l$. It is trivially true if θ equals any of $\theta_1, \ldots \theta_k$, or is unifiable with any of them. This models proof by assumption in natural deduction.

Isabelle automates the meta-inference for proof by assumption. Its arguments are the meta-theorem $[\![\phi_1; \ldots; \phi_n]\!] \Longrightarrow \phi$, and some i from 1 to n, where ϕ_i has the form (2). Its results are meta-theorems of the form

$$([\![\phi_1; \ldots; \phi_{i-1}; \phi_{i+1}; \phi_n]\!] \Longrightarrow \phi)s$$

for each s and j such that s unifies $\lambda x_1 \ldots x_l . \theta_j$ with $\lambda x_1 \ldots x_l . \theta$. Isabelle supplies the parameters $x_1, \ldots, x_l$ to higher-order unification as bound variables, which regards them as unique constants with a limited scope — this enforces parameter provisos [12].

The premise represents a proof state with n subgoals, of which the ith is to be solved by assumption. Isabelle searches the subgoal's context for an assumption θ_j that can solve it. For each unifier, the meta-inference returns an instantiated proof state from which the ith subgoal has been removed. Isabelle searches for a unifying assumption; for readability and robustness, proofs do not refer to assumptions by number.

Consider the proof state

$$([\![P(a); P(b)]\!] \Longrightarrow P(?x)) \Longrightarrow Q(?x).$$

Proof by assumption (with $i = 1$, the only possibility) yields two results:

- $Q(a)$, instantiating $?x \equiv a$
- $Q(b)$, instantiating $?x \equiv b$

Here, proof by assumption affects the main goal. It could also affect other subgoals; if we also had the subgoal $[\![P(b); P(c)]\!] \Longrightarrow P(?x)$, then $?x \equiv a$ would transform it to $[\![P(b); P(c)]\!] \Longrightarrow P(a)$, which might be unprovable.

5.3 A propositional proof

Our first example avoids quantifiers. Given the main goal $P \lor P \rightarrow P$, Isabelle creates the initial state

$$(P \lor P \rightarrow P) \Longrightarrow (P \lor P \rightarrow P).$$

Bear in mind that every proof state we derive will be a meta-theorem, expressing that the subgoals imply the main goal. Our aim is to reach the state $P \lor P \rightarrow P$; this meta-theorem is the desired result.

The first step is to refine subgoal 1 by $(\rightarrow I)$, creating a new state where $P \lor P$ is an assumption:

$$(P \lor P \Longrightarrow P) \Longrightarrow (P \lor P \rightarrow P)$$

The next step is $(\vee E)$, which replaces subgoal 1 by three new subgoals. Because of lifting, each subgoal contains a copy of the context — the assumption $P \vee P$. (In fact, this assumption is now redundant; we shall shortly see how to get rid of it!) The new proof state is the following meta-theorem, laid out for clarity:

$$\begin{array}{ll} [\![\; P \vee P \Longrightarrow ?P_1 \vee ?Q_1; & \text{(subgoal 1)} \\ \quad [\![P \vee P; ?P_1]\!] \Longrightarrow P; & \text{(subgoal 2)} \\ \quad [\![P \vee P; ?Q_1]\!] \Longrightarrow P & \text{(subgoal 3)} \\ ]\!] \Longrightarrow (P \vee P \rightarrow P) & \text{(main goal)} \end{array}$$

Notice the unknowns in the proof state. Because we have applied $(\vee E)$, we must prove some disjunction, $?P_1 \vee ?Q_1$. Of course, subgoal 1 is provable by assumption. This instantiates both $?P_1$ and $?Q_1$ to P throughout the proof state:

$$\begin{array}{ll} [\![\; [\![P \vee P; P]\!] \Longrightarrow P; & \text{(subgoal 1)} \\ \quad [\![P \vee P; P]\!] \Longrightarrow P & \text{(subgoal 2)} \\ ]\!] \Longrightarrow (P \vee P \rightarrow P) & \text{(main goal)} \end{array}$$

Both of the remaining subgoals can be proved by assumption. After two such steps, the proof state is $P \vee P \rightarrow P$.

5.4 A quantifier proof

To illustrate quantifiers and $\bigwedge$-lifting, let us prove $(\exists x \,.\, P(f(x))) \rightarrow (\exists x \,.\, P(x))$. The initial proof state is the trivial meta-theorem

$$(\exists x \,.\, P(f(x))) \rightarrow (\exists x \,.\, P(x)) \Longrightarrow (\exists x \,.\, P(f(x))) \rightarrow (\exists x \,.\, P(x)).$$

As above, the first step is refinement by $(\rightarrow I)$:

$$(\exists x \,.\, P(f(x)) \Longrightarrow \exists x \,.\, P(x)) \Longrightarrow (\exists x \,.\, P(f(x))) \rightarrow (\exists x \,.\, P(x))$$

The next step is $(\exists E)$, which replaces subgoal 1 by two new subgoals. Both have the assumption $\exists x \,.\, P(f(x))$. The new proof state is the meta-theorem

$$\begin{array}{ll} [\![\; \exists x \,.\, P(f(x)) \Longrightarrow \exists x \,.\, ?P_1(x); & \text{(subgoal 1)} \\ \quad \bigwedge x \,.\, [\![\exists x \,.\, P(f(x)); ?P_1(x)]\!] \Longrightarrow \exists x \,.\, P(x) & \text{(subgoal 2)} \\ ]\!] \Longrightarrow (\exists x \,.\, P(f(x))) \rightarrow (\exists x \,.\, P(x)) & \text{(main goal)} \end{array}$$

The unknown $?P_1$ appears in both subgoals. Because we have applied $(\exists E)$, we must prove $\exists x \,.\, ?P_1(x)$, where $?P_1(x)$ may become any formula possibly containing x. Proving subgoal 1 by assumption instantiates $?P_1$ to $\lambda x \,.\, P(f(x))$:

$$\left(\bigwedge x \,.\, [\![\exists x \,.\, P(f(x)); P(f(x))]\!] \Longrightarrow \exists x \,.\, P(x)\right) \Longrightarrow (\exists x \,.\, P(f(x))) \rightarrow (\exists x \,.\, P(x))$$

The next step is refinement by $(\exists I)$. The rule is lifted into the context of the parameter x and the assumption $P(f(x))$. This copies the context to the subgoal and allows the existential witness to depend upon x:

$$\left(\bigwedge x \,.\, [\![\exists x \,.\, P(f(x)); P(f(x))]\!] \Longrightarrow P(?x_2(x))\right) \Longrightarrow (\exists x \,.\, P(f(x))) \to (\exists x \,.\, P(x))$$

The existential witness, $?x_2(x)$, consists of an unknown applied to a parameter. Proof by assumption unifies $\lambda x \,.\, P(f(x))$ with $\lambda x \,.\, P(?x_2(x))$, instantiating $?x_2$ to f. The final proof state contains no subgoals: $(\exists x \,.\, P(f(x))) \to (\exists x \,.\, P(x))$.

5.5 Tactics and tacticals

Tactics perform backward proof. Isabelle tactics differ from those of LCF, HOL and Nuprl by operating on entire proof states, rather than on individual subgoals. An Isabelle tactic is a function that takes a proof state and returns a sequence (lazy list) of possible successor states. Lazy lists are coded in ML as functions, a standard technique [14]. Isabelle represents proof states by theorems.

Basic tactics execute the meta-rules described above, operating on a given subgoal. The **resolution tactics** take a list of rules and return next states for each combination of rule and unifier. The **assumption tactic** examines the subgoal's assumptions and returns next states for each combination of assumption and unifier. Lazy lists are essential because higher-order resolution may return infinitely many unifiers. If there are no matching rules or assumptions then no next states are generated; a tactic application that returns an empty list is said to **fail**.

Sequences realize their full potential with **tacticals** — operators for combining tactics. Depth-first search, breadth-first search and best-first search (where a heuristic function selects the best state to explore) return their outcomes as a sequence. Isabelle provides such procedures in the form of tacticals. Simpler procedures can be expressed directly using the basic tacticals `THEN`, `ORELSE` and `REPEAT`:

*tac*1 `THEN` *tac*2 is a tactic for sequential composition. Applied to a proof state, it returns all states reachable in two steps by applying *tac*1 followed by *tac*2.

*tac*1 `ORELSE` *tac*2 is a choice tactic. Applied to a state, it tries *tac*1 and returns the result if non-empty; otherwise, it uses *tac*2.

`REPEAT` *tac* is a repetition tactic. Applied to a state, it returns all states reachable by applying *tac* as long as possible — until it would fail.

For instance, this tactic repeatedly applies *tac*1 and *tac*2, giving *tac*1 priority:

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

6 Variations on resolution

In principle, resolution and proof by assumption suffice to prove all theorems. However, specialized forms of resolution are helpful for working with elimination rules. Elim-resolution applies an elimination rule to an assumption; destruct-resolution is similar, but applies a rule in a forward style.

The last part of the section shows how the techniques for proving theorems can also serve to derive rules.

6.1 Elim-resolution

Consider proving the theorem $((R \lor R) \lor R) \lor R \to R$. By $(\to I)$, we prove R from the assumption $((R \lor R) \lor R) \lor R$. Applying $(\lor E)$ to this assumption yields two subgoals, one that assumes R (and is therefore trivial) and one that assumes $(R \lor R) \lor R$. This subgoal admits another application of $(\lor E)$. Since natural deduction never discards assumptions, we eventually generate a subgoal containing much that is redundant:

$$[\![((R \lor R) \lor R) \lor R; (R \lor R) \lor R; R \lor R; R]\!] \Longrightarrow R.$$

In general, using $(\lor E)$ on the assumption $P \lor Q$ creates two new subgoals with the additional assumption P or Q. In these subgoals, $P \lor Q$ is redundant. Other elimination rules behave similarly. In first-order logic, only universally quantified assumptions are sometimes needed more than once — say, to prove $P(f(f(a)))$ from the assumptions $\forall x \,.\, P(x) \to P(f(x))$ and $P(a)$.

Many logics can be formulated as sequent calculi that delete redundant assumptions after use. The rule $(\lor E)$ might become

$$\frac{\Gamma, P, \Delta \vdash \Theta \qquad \Gamma, Q, \Delta \vdash \Theta}{\Gamma, P \lor Q, \Delta \vdash \Theta}\ \lor\text{-left}$$

In backward proof, a goal containing $P \lor Q$ on the left of the $\vdash$ (that is, as an assumption) splits into two subgoals, replacing $P \lor Q$ by P or Q. But the sequent calculus, with its explicit handling of assumptions, can be tiresome to use.

Elim-resolution is Isabelle's way of getting sequent calculus behaviour from natural deduction rules. It lets an elimination rule consume an assumption. Elim-resolution combines two meta-theorems:

- a rule $[\![\psi_1; \ldots; \psi_m]\!] \Longrightarrow \psi$
- a proof state $[\![\phi_1; \ldots; \phi_n]\!] \Longrightarrow \phi$

The rule must have at least one premise, thus $m > 0$. Write the rule's lifted form as $[\![\psi'_1; \ldots; \psi'_m]\!] \Longrightarrow \psi'$. Suppose we wish to change subgoal number i.

Ordinary resolution would attempt to reduce ϕ_i, replacing subgoal i by m new ones. Elim-resolution tries simultaneously to reduce ϕ_i and to solve ψ'_1 by assumption; it returns a sequence of next states. Each of these replaces subgoal i by instances of $\psi'_2, \ldots, \psi'_m$ from which the selected assumption has been deleted. Suppose ϕ_i has the parameter x and assumptions $\theta_1, \ldots, \theta_k$. Then ψ'_1, the rule's first premise after lifting, will be $\bigwedge x \,.\, [\![\theta_1; \ldots; \theta_k]\!] \Longrightarrow \psi_1^x$. Elim-resolution tries to unify $\psi' \stackrel{?}{\equiv} \phi_i$ and $\lambda x \,.\, \theta_j \stackrel{?}{\equiv} \lambda x \,.\, \psi_1^x$ simultaneously, for $j = 1, \ldots, k$.

Let us redo the example from §5.3. The elimination rule is $(\vee E)$,

$$[\![?P \vee ?Q;\ ?P \Longrightarrow ?R;\ ?Q \Longrightarrow ?R]\!] \Longrightarrow ?R$$

and the proof state is $(P \vee P \Longrightarrow P) \Longrightarrow (P \vee P \rightarrow P)$. The lifted rule is

$$\begin{array}{l} [\![\ P \vee P \Longrightarrow ?P_1 \vee ?Q_1; \\ \quad [\![P \vee P;\ ?P_1]\!] \Longrightarrow ?R_1; \\ \quad [\![P \vee P;\ ?Q_1]\!] \Longrightarrow ?R_1 \\ ]\!] \Longrightarrow (P \vee P \Longrightarrow ?R_1) \end{array}$$

Unification takes the simultaneous equations $P \vee P \stackrel{?}{\equiv} ?P_1 \vee ?Q_1$ and $?R_1 \stackrel{?}{\equiv} P$, yielding $?P_1 \equiv ?Q_1 \equiv ?R_1 \equiv P$. The new proof state is simply

$$[\![P \Longrightarrow P;\ P \Longrightarrow P]\!] \Longrightarrow (P \vee P \rightarrow P).$$

Elim-resolution's simultaneous unification gives better control than ordinary resolution. Recall the substitution rule:

$$[\![?t = ?u; ?P(?t)]\!] \Longrightarrow ?P(?u) \qquad (subst)$$

Unsuitable for ordinary resolution because $?P(?u)$ admits many unifiers, $(subst)$ works well with elim-resolution. It deletes some assumption of the form $x = y$ and replaces every y by x in the subgoal formula. The simultaneous unification instantiates $?u$ to y; if y is not an unknown, then $?P(y)$ can easily be unified with another formula.

In logical parlance, the premise containing the connective to be eliminated is called the **major premise**. Elim-resolution expects the major premise to come first. The order of the premises is significant in Isabelle.

6.2 Destruction rules

Looking back to Fig. 1, notice that there are two kinds of elimination rule. The rules $(\wedge E1)$, $(\wedge E2)$, $(\rightarrow E)$ and $(\forall E)$ extract the conclusion from the major premise. In Isabelle parlance, such rules are called **destruction rules**; they are readable and easy to use in forward proof. The rules $(\vee E)$, $(\bot E)$ and $(\exists E)$ work by discharging assumptions; they support backward proof in a style reminiscent of the sequent calculus.

The latter style is the most general form of elimination rule. In natural deduction, there is no way to recast $(\vee E)$, $(\bot E)$ or $(\exists E)$ as destruction rules. But we can write general elimination rules for $\land$, $\rightarrow$ and $\forall$:

$$\frac{P \land Q \qquad \begin{matrix}[P,Q]\\ \vdots\\ R\end{matrix}}{R} \qquad \frac{P \rightarrow Q \quad P \quad \begin{matrix}[Q]\\ \vdots\\ R\end{matrix}}{R} \qquad \frac{\forall x \,.\, P \qquad \begin{matrix}[P[t/x]]\\ \vdots\\ Q\end{matrix}}{Q}$$

Because they are concise, destruction rules are simpler to derive than the corresponding elimination rules. To facilitate their use in backward proof, Isabelle provides a means of transforming a destruction rule such as

$$\frac{P_1 \quad \ldots \quad P_m}{Q} \quad \text{to the elimination rule} \quad \frac{P_1 \quad \ldots \quad P_m \quad \begin{matrix}[Q]\\ \vdots\\ R\end{matrix}}{R.}$$

Destruct-resolution combines this transformation with elim-resolution. It applies a destruction rule to some assumption of a subgoal. Given the rule above, it replaces the assumption P_1 by Q, with new subgoals of showing instances of P_2, …, P_m. Destruct-resolution works forward from a subgoal's assumptions. Ordinary resolution performs forward reasoning from theorems, as illustrated in §3.2.

6.3 Deriving rules by resolution

The meta-logic, itself a form of the predicate calculus, is defined by a system of natural deduction rules. Each theorem may depend upon meta-assumptions. The theorem that ϕ follows from the assumptions ϕ_1, …, ϕ_n is written

$$\phi \quad [\phi_1, \ldots, \phi_n].$$

A more conventional notation might be $\phi_1, \ldots, \phi_n \vdash \phi$, but Isabelle's notation is more readable with large formulae.

Meta-level natural deduction provides a convenient mechanism for deriving new object-level rules. To derive the rule

$$\frac{\theta_1 \quad \ldots \quad \theta_k}{\phi,}$$

assume the premises θ_1, …, θ_k at the meta-level. Then prove ϕ, possibly using these assumptions. Starting with a proof state $\phi \Longrightarrow \phi$, assumptions may accumulate, reaching a final state such as

$$\phi \quad [\theta_1, \ldots, \theta_k].$$

The meta-rule for $\Longrightarrow$ introduction discharges an assumption. Discharging them in the order $\theta_k, \ldots, \theta_1$ yields the meta-theorem $[\![\theta_1; \ldots; \theta_k]\!] \Longrightarrow \phi$, with no assumptions. This represents the desired rule. Let us derive the general $\wedge$ elimination rule:

$$\frac{P \wedge Q \qquad \begin{array}{c}[P, Q] \\ \vdots \\ R\end{array}}{R} \qquad (\wedge E)$$

We assume $P \wedge Q$ and $[\![P; Q]\!] \Longrightarrow R$, and commence backward proof in the state $R \Longrightarrow R$. Resolving this with the second assumption yields the state

$$[\![P; Q]\!] \Longrightarrow R \quad [\,[\![P; Q]\!] \Longrightarrow R\,].$$

Resolving subgoals 1 and 2 with $(\wedge E1)$ and $(\wedge E2)$, respectively, yields the state

$$[\![P \wedge ?Q_1;\ ?P_2 \wedge Q]\!] \Longrightarrow R \quad [\,[\![P; Q]\!] \Longrightarrow R\,].$$

The unknowns $?Q_1$ and $?P_2$ arise from unconstrained subformulae in the premises of $(\wedge E1)$ and $(\wedge E2)$. Resolving both subgoals with the assumption $P \wedge Q$ instantiates the unknowns to yield

$$R \quad [\,[\![P; Q]\!] \Longrightarrow R, P \wedge Q\,].$$

The proof may use the meta-assumptions in any order, and as often as necessary; when finished, we discharge them in the correct order to obtain the desired form:

$$[\![P \wedge Q;\ [\![P; Q]\!] \Longrightarrow R]\!] \Longrightarrow R$$

We have derived the rule using free variables, which prevents their premature instantiation during the proof; we may now replace them by schematic variables.

! Schematic variables are not allowed in meta-assumptions, for a variety of reasons. Meta-assumptions remain fixed throughout a proof.

Part II
Getting Started with Isabelle

Let us consider how to perform simple proofs using Isabelle. At present, Isabelle's user interface is ML. Proofs are conducted by applying certain ML functions, which update a stored proof state. Basically, all syntax must be expressed using plain ASCII characters. There are also mechanisms built into Isabelle that support alternative syntaxes, for example using mathematical symbols from a special screen font. The meta-logic and major object-logics already provide such fancy output as an option.

Object-logics are built upon Pure Isabelle, which implements the meta-logic and provides certain fundamental data structures: types, terms, signatures, theorems and theories, tactics and tacticals. These data structures have the corresponding ML types `typ`, `term`, `Sign.sg`, `thm`, `theory` and `tactic`; tacticals have function types such as `tactic->tactic`. Isabelle users can operate on these data structures by writing ML programs.

7 Forward proof

This section describes the concrete syntax for types, terms and theorems, and demonstrates forward proof. The examples are set in first-order logic. The command to start Isabelle running first-order logic is

```
isabelle FOL
```

Note that just typing `isabelle` usually brings up higher-order logic (`HOL`) by default.

7.1 Lexical matters

An **identifier** is a string of letters, digits, underscores (`_`) and single quotes (`'`), beginning with a letter. Single quotes are regarded as primes; for instance `x'` is read as x'. Identifiers are separated by white space and special characters. **Reserved words** are identifiers that appear in Isabelle syntax definitions.

An Isabelle theory can declare symbols composed of special characters, such as `=`, `==`, `=>` and `==>`. (The latter three are part of the syntax of the meta-logic.) Such symbols may be run together; thus if `}` and `{` are used for set brackets then `{{a},{a,b}}` is valid notation for a set of sets — but only if `}}` and `{{` have not been declared as symbols! The parser resolves any ambiguity by taking the longest possible symbol that has been declared. Thus the string `==>` is read as a single symbol. But `= =>` is read as two symbols.

Identifiers that are not reserved words may serve as free variables or constants. A **type identifier** consists of an identifier prefixed by a prime, for example `'a` and `'hello`. Type identifiers stand for (free) type variables, which remain fixed during a proof.

An **unknown** (or type unknown) consists of a question mark, an identifier (or type identifier), and a subscript. The subscript, a non-negative integer, allows the renaming of unknowns prior to unification.[1]

7.2 Syntax of types and terms

Classes are denoted by identifiers; the built-in class `logic` contains the 'logical' types. Sorts are lists of classes enclosed in braces } and {; singleton sorts may be abbreviated by dropping the braces.

Types are written with a syntax like ML's. The built-in type *prop* is the type of propositions. Type variables can be constrained to particular classes or sorts, for example `'a::term` and `?'b::{ord, arith}`.

ASCII Notation for Types

α`::`C	class constraint
α`::{`C_1`,` ... `,` C_n`}`	sort constraint
σ `=>` τ	function type $\sigma \Rightarrow \tau$
`[`σ_1`,` ... `,`σ_n`] =>` τ	n-argument function type
`(`τ_1`,` ... `,`τ_n`)` *tycon*	type construction

Terms are those of the typed λ-calculus.

ASCII Notation for Terms

t`::`σ	type constraint
`%`x `.` t	abstraction $\lambda x \,.\, t$
`%`$x_1 \ldots x_n$ `.` t	abstraction over several arguments
t`(`u_1`,` ... `,` u_n`)`	application to several arguments (FOL and ZF)
$t\ u_1 \ldots\ u_n$	application to several arguments (HOL)

Note that `HOL` uses its traditional "higher-order" syntax for application, which differs from that used in `FOL`.

The theorems and rules of an object-logic are represented by theorems in the meta-logic, which are expressed using meta-formulae. Since the meta-logic is

[1] The subscript may appear after the identifier, separated by a dot; this prevents ambiguity when the identifier ends with a digit. Thus `?z6.0` has identifier `"z6"` and subscript 0, while `?a0.5` has identifier `"a0"` and subscript 5. If the identifier does not end with a digit, then no dot appears and a subscript of 0 is omitted; for example, `?hello` has identifier `"hello"` and subscript zero, while `?z6` has identifier `"z"` and subscript 6. The same conventions apply to type unknowns. The question mark is *not* part of the identifier!

higher-order, meta-formulae ϕ, ψ, θ, ... are just terms of type `prop`.

ASCII Notation for Meta-Formulae

a `==` b	$a \equiv b$	meta-equality
a `=?=` b	$a \stackrel{?}{\equiv} b$	flex-flex constraint
ϕ `==>` ψ	$\phi \Longrightarrow \psi$	meta-implication
`[|`ϕ_1`;`...`;`ϕ_n`|]` `==>` ψ	$[\![\phi_1; \ldots; \phi_n]\!] \Longrightarrow \psi$	nested implication
`!!`x`.`ϕ	$\bigwedge x\,.\,\phi$	meta-quantification
`!!`$x_1 \ldots x_n$`.`ϕ	$\bigwedge x_1\,.\,\ldots x_n\,.\,\phi$	nested quantification

Flex-flex constraints are meta-equalities arising from unification; they require special treatment. See §7.4.

Most logics define the implicit coercion *Trueprop* from object-formulae to propositions. This could cause an ambiguity: in $P \Longrightarrow Q$, do the variables P and Q stand for meta-formulae or object-formulae? If the latter, $P \Longrightarrow Q$ really abbreviates $\mathit{Trueprop}(P) \Longrightarrow \mathit{Trueprop}(Q)$. To prevent such ambiguities, Isabelle's syntax does not allow a meta-formula to consist of a variable. Variables of type *prop* are seldom useful, but you can make a variable stand for a meta-formula by prefixing it with the symbol `PROP`:

```
PROP ?psi ==> PROP ?theta
```

Symbols of object-logics are typically rendered into ASCII as follows:

`True`	$\top$	true
`False`	$\bot$	false
P `&` Q	$P \land Q$	conjunction
P `|` Q	$P \lor Q$	disjunction
`~` P	$\neg P$	negation
P `-->` Q	$P \to Q$	implication
P `<->` Q	$P \leftrightarrow Q$	bi-implication
`ALL` $x\,y\,z$ `.` P	$\forall x\,y\,z\,.\,P$	for all
`EX` $x\,y\,z$ `.` P	$\exists x\,y\,z\,.\,P$	there exists

To illustrate the notation, consider two axioms for first-order logic:

$$[\![P; Q]\!] \Longrightarrow P \land Q \qquad (\land I)$$

$$[\![\exists x\,.\,P(x); \bigwedge x\,.\,P(x) \to Q]\!] \Longrightarrow Q \qquad (\exists E)$$

$(\land I)$ translates into ASCII characters as

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

The schematic variables let unification instantiate the rule. To avoid cluttering logic definitions with question marks, Isabelle converts any free variables in a rule to schematic variables; we normally declare $(\land I)$ as

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

This variables convention agrees with the treatment of variables in goals. Free variables in a goal remain fixed throughout the proof. After the proof is finished, Isabelle converts them to scheme variables in the resulting theorem. Scheme variables in a goal may be replaced by terms during the proof, supporting answer extraction, program synthesis, and so forth.

For a final example, the rule $(\exists E)$ is rendered in ASCII as

```
[| EX x.P(x);  !!x. P(x) ==> Q |] ==> Q
```

7.3 Basic operations on theorems

Meta-level theorems have the ML type `thm`. They represent the theorems and inference rules of object-logics. Isabelle's meta-logic is implemented using the LCF approach: each meta-level inference rule is represented by a function from theorems to theorems. Object-level rules are taken as axioms.

The main theorem printing commands are `prth`, `prths` and `prthq`. Of the other operations on theorems, most useful are `RS` and `RSN`, which perform resolution.

`prth` *thm*; pretty-prints *thm* at the terminal.

`prths` *thms*; pretty-prints *thms*, a list of theorems.

`prthq` *thmq*; pretty-prints *thmq*, a sequence of theorems; this is useful for inspecting the output of a tactic.

*thm*1 `RS` *thm*2 resolves the conclusion of *thm*1 with the first premise of *thm*2.

*thm*1 `RSN` (*i*, *thm*2) resolves the conclusion of *thm*1 with the *i*th premise of *thm*2.

`standard` *thm* puts *thm* into a standard format. It also renames schematic variables to have subscript zero, improving readability and reducing subscript growth.

The rules of a theory are normally bound to ML identifiers. Suppose we are running an Isabelle session containing theory `FOL`, natural deduction first-order logic.[2] Let us try an example given in §3.2. We first print `mp`, which is the

[2]For a listing of the `FOL` rules and their ML names, turn to *Isabelle's Object-Logics*.

rule ($\rightarrow E$), then resolve it with itself.

```
prth mp;
  [| ?P --> ?Q; ?P |] ==> ?Q
  val it = "[| ?P --> ?Q; ?P |] ==> ?Q" : thm
prth (mp RS mp);
  [| ?P1 --> ?P --> ?Q; ?P1; ?P |] ==> ?Q
  val it = "[| ?P1 --> ?P --> ?Q; ?P1; ?P |] ==> ?Q" : thm
```

User input appears in `typewriter characters`, and output appears in *`slanted typewriter characters`*. ML's response *`val`* ... is compiler-dependent and will sometimes be suppressed. This session illustrates two formats for the display of theorems. Isabelle's top-level displays theorems as ML values, enclosed in quotes. Printing commands like `prth` omit the quotes and the surrounding `val ...: thm`. Ignoring their side-effects, the commands are identity functions.

To contrast `RS` with `RSN`, we resolve `conjunct1`, which stands for ($\wedge E1$), with `mp`.

```
conjunct1 RS mp;
  val it = "[| (?P --> ?Q) & ?Q1; ?P |] ==> ?Q" : thm
conjunct1 RSN (2,mp);
  val it = "[| ?P --> ?Q; ?P & ?Q1 |] ==> ?Q" : thm
```

These correspond to the following proofs:

$$\frac{\dfrac{(P \rightarrow Q) \wedge Q_1}{P \rightarrow Q}\,(\wedge E1) \qquad P}{Q}\,(\rightarrow E) \qquad\qquad \frac{P \rightarrow Q \qquad \dfrac{P \wedge Q_1}{P}\,(\wedge E1)}{Q}\,(\rightarrow E)$$

Rules can be derived by pasting other rules together. Let us join `spec`, which stands for ($\forall E$), with `mp` and `conjunct1`. In ML, the identifier `it` denotes the value just printed.

```
spec;
  val it = "ALL x. ?P(x) ==> ?P(?x)" : thm
it RS mp;
  val it = "[| ALL x. ?P3(x) --> ?Q2(x); ?P3(?x1) |] ==>
           ?Q2(?x1)" : thm
it RS conjunct1;
  val it = "[| ALL x. ?P4(x) --> ?P6(x) & ?Q5(x); ?P4(?x2) |] ==>
           ?P6(?x2)" : thm
standard it;
  val it = "[| ALL x. ?P(x) --> ?Pa(x) & ?Q(x); ?P(?x) |] ==>
           ?Pa(?x)" : thm
```

By resolving ($\forall E$) with ($\rightarrow E$) and ($\wedge E1$), we have derived a destruction rule for formulae of the form $\forall x \,.\, P(x) \rightarrow (Q(x) \wedge R(x))$. Used with destruct-resolution, such specialized rules provide a way of referring to particular assumptions.

7.4 *Flex-flex constraints

In higher-order unification, **flex-flex** equations are those where both sides begin with a function unknown, such as $?f(0) \stackrel{?}{\equiv} ?g(0)$. They admit a trivial unifier, here $?f \equiv \lambda x . ?a$ and $?g \equiv \lambda y . ?a$, where $?a$ is a new unknown. They admit many other unifiers, such as $?f \equiv \lambda x . ?g(0)$ and $\{?f \equiv \lambda x . x, ?g \equiv \lambda x . 0\}$. Huet's procedure does not enumerate the unifiers; instead, it retains flex-flex equations as constraints on future unifications. Flex-flex constraints occasionally become attached to a proof state; more frequently, they appear during use of `RS` and `RSN`:

```
refl;
  val it = "?a = ?a" : thm
exI;
  val it = "?P(?x) ==> EX x. ?P(x)" : thm
refl RS exI;
  val it = "?a3(?x) =?= ?a2(?x) ==> EX x. ?a3(x) = ?a2(x)" : thm
```

Renaming variables, this is $\exists x . ?f(x) = ?g(x)$ with the constraint $?f(?u) \stackrel{?}{\equiv} ?g(?u)$. Instances satisfying the constraint include $\exists x . ?f(x) = ?f(x)$ and $\exists x . x = ?u$. Calling `flexflex_rule` removes all constraints by applying the trivial unifier:

```
prthq (flexflex_rule it);
  EX x. ?a4 = ?a4
```

Isabelle simplifies flex-flex equations to eliminate redundant bound variables. In $\lambda x\, y . ?f(k(y), x) \stackrel{?}{\equiv} \lambda x\, y . ?g(y)$, there is no bound occurrence of x on the right side; thus, there will be none on the left in a common instance of these terms. Choosing a new variable $?h$, Isabelle assigns $?f \equiv \lambda u\, v . ?h(u)$, simplifying the left side to $\lambda x\, y . ?h(k(y))$. Dropping x from the equation leaves $\lambda y . ?h(k(y)) \stackrel{?}{\equiv} \lambda y . ?g(y)$. By η-conversion, this simplifies to the assignment $?g \equiv \lambda y . ?h(k(y))$.

! `RS` and `RSN` fail (by raising exception `THM`) unless the resolution delivers **exactly one** resolvent. For multiple results, use `RL` and `RLN`, which operate on theorem lists. The following example uses `read_instantiate` to create an instance of `refl` containing no schematic variables.

```
val reflk = read_instantiate [("a","k")] refl;
  val reflk = "k = k" : thm
```

A flex-flex constraint is no longer possible; resolution does not find a unique unifier:

```
reflk RS exI;
  uncaught exception THM
```

Using `RL` this time, we discover that there are four unifiers, and four resolvents:

```
[reflk] RL [exI];
  val it = ["EX x. x = x", "EX x. k = x",
            "EX x. x = k", "EX x. k = k"] : thm list
```

8 Backward proof

Although `RS` and `RSN` are fine for simple forward reasoning, large proofs require tactics. Isabelle provides a suite of commands for conducting a backward proof using tactics.

8.1 The basic tactics

The tactics `assume_tac`, `resolve_tac`, `eresolve_tac`, and `dresolve_tac` suffice for most single-step proofs. Although `eresolve_tac` and `dresolve_tac` are not strictly necessary, they simplify proofs involving elimination and destruction rules. All the tactics act on a subgoal designated by a positive integer i, failing if i is out of range. The resolution tactics try their list of theorems in left-to-right order.

`assume_tac` i is the tactic that attempts to solve subgoal i by assumption. Proof by assumption is not a trivial step; it can falsify other subgoals by instantiating shared variables. There may be several ways of solving the subgoal by assumption.

`resolve_tac` *thms* i is the basic resolution tactic, used for most proof steps. The *thms* represent object-rules, which are resolved against subgoal i of the proof state. For each rule, resolution forms next states by unifying the conclusion with the subgoal and inserting instantiated premises in its place. A rule can admit many higher-order unifiers. The tactic fails if none of the rules generates next states.

`eresolve_tac` *thms* i performs elim-resolution. Like `resolve_tac` *thms* i followed by `assume_tac` i, it applies a rule then solves its first premise by assumption. But `eresolve_tac` additionally deletes that assumption from any subgoals arising from the resolution.

`dresolve_tac` *thms* i performs destruct-resolution with the *thms*, as described in §6.2. It is useful for forward reasoning from the assumptions.

8.2 Commands for backward proof

Tactics are normally applied using the subgoal module, which maintains a proof state and manages the proof construction. It allows interactive backtracking through the proof space, going away to prove lemmas, etc.; of its many commands, most important are the following:

`Goal` *formula*`;` begins a new proof, where *theory* is usually an ML identifier and the *formula* is written as an ML string.

`by` *tactic*; applies the *tactic* to the current proof state, raising an exception if the tactic fails.

`undo();` reverts to the previous proof state. Undo can be repeated but cannot be undone. Do not omit the parentheses; typing `undo;` merely causes ML to echo the value of that function.

`result();` returns the theorem just proved, in a standard format. It fails if unproved subgoals are left, etc.

`qed` *name*; is the usual way of ending a proof. It gets the theorem using `result`, stores it in Isabelle's theorem database and binds it to an ML identifier.

The commands and tactics given above are cumbersome for interactive use. Although our examples will use the full commands, you may prefer Isabelle's short-cuts:

`ba` *i*;	abbreviates	`by (assume_tac` *i*`);`
`br` *thm i*;	abbreviates	`by (resolve_tac [`*thm*`]` *i*`);`
`be` *thm i*;	abbreviates	`by (eresolve_tac [`*thm*`]` *i*`);`
`bd` *thm i*;	abbreviates	`by (dresolve_tac [`*thm*`]` *i*`);`

8.3 A trivial example in propositional logic

Directory `FOL` of the Isabelle distribution defines the theory of first-order logic. Let us try the example from §5.3, entering the goal $P \lor P \to P$ in that theory.[3]

```
Goal "P|P --> P";
  Level 0
  P | P --> P
  1. P | P --> P
```

Isabelle responds by printing the initial proof state, which has $P \lor P \to P$ as the main goal and the only subgoal. The **level** of the state is the number of `by` commands that have been applied to reach it. We now use `resolve_tac` to apply the rule `impI`, or $(\to I)$, to subgoal 1:

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

In the new proof state, subgoal 1 is P under the assumption $P \lor P$. (The meta-implication `==>` indicates assumptions.) We apply `disjE`, or $(\lor E)$, to that

[3]To run these examples, see the file `FOL/ex/intro.ML`.

subgoal:

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

At Level 2 there are three subgoals, each provable by assumption. We deviate from §5.3 by tackling subgoal 3 first, using `assume_tac`. This affects subgoal 1, updating `?Q1` to `P`.

```
by (assume_tac 3);
  Level 3
  P | P --> P
  1. P | P ==> ?P1 | P
  2. [| P | P; ?P1 |] ==> P
```

Next we tackle subgoal 2, instantiating `?P1` to `P` in subgoal 1.

```
by (assume_tac 2);
  Level 4
  P | P --> P
  1. P | P ==> P | P
```

Lastly we prove the remaining subgoal by assumption:

```
by (assume_tac 1);
  Level 5
  P | P --> P
  No subgoals!
```

Isabelle tells us that there are no longer any subgoals: the proof is complete. Calling `qed` stores the theorem.

```
qed "mythm";
  val mythm = "?P | ?P --> ?P" : thm
```

Isabelle has replaced the free variable `P` by the scheme variable `?P`. Free variables in the proof state remain fixed throughout the proof. Isabelle finally converts them to scheme variables so that the resulting theorem can be instantiated with any formula.

As an exercise, try doing the proof as in §5.3, observing how instantiations affect the proof state.

8.4 Part of a distributive law

To demonstrate the tactics `eresolve_tac`, `dresolve_tac` and the tactical `REPEAT`, let us prove part of the distributive law

$$(P \land Q) \lor R \leftrightarrow (P \lor R) \land (Q \lor R).$$

We begin by stating the goal to Isabelle and applying $(\to I)$ to it:

```
Goal "(P & Q) | R  --> (P | R)";
  Level 0
  P & Q | R --> P | R
   1. P & Q | R --> P | R
by (resolve_tac [impI] 1);
  Level 1
  P & Q | R --> P | R
   1. P & Q | R ==> P | R
```

Previously we applied $(\lor E)$ using `resolve_tac`, but `eresolve_tac` deletes the assumption after use. The resulting proof state is simpler.

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

Using `dresolve_tac`, we can apply $(\land E1)$ to subgoal 1, replacing the assumption $P \land Q$ by P. Normally we should apply the rule $(\land E)$, given in §6.2. That is an elimination rule and requires `eresolve_tac`; it would replace $P \land Q$ by the two assumptions P and Q. Because the present example does not need Q, we may try out `dresolve_tac`.

```
by (dresolve_tac [conjunct1] 1);
  Level 3
  P & Q | R --> P | R
   1. P ==> P | R
   2. R ==> P | R
```

The next two steps apply $(\lor I1)$ and $(\lor I2)$ in an obvious manner.

```
by (resolve_tac [disjI1] 1);
  Level 4
  P & Q | R --> P | R
   1. P ==> P
   2. R ==> P | R
by (resolve_tac [disjI2] 2);
  Level 5
  P & Q | R --> P | R
   1. P ==> P
   2. R ==> R
```

Two calls of `assume_tac` can finish the proof. The tactical `REPEAT` here expresses a tactic that calls `assume_tac 1` as many times as possible. We can restrict attention to subgoal 1 because the other subgoals move up after subgoal 1 disappears.

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

9 Quantifier reasoning

This section illustrates how Isabelle enforces quantifier provisos. Suppose that x, y and z are parameters of a subgoal. Quantifier rules create terms such as $?f(x,z)$, where $?f$ is a function unknown. Instantiating $?f$ to $\lambda x\, z\,.\, t$ has the effect of replacing $?f(x,z)$ by t, where the term t may contain free occurrences of x and z. On the other hand, no instantiation of $?f$ can replace $?f(x,z)$ by a term containing free occurrences of y, since parameters are bound variables.

9.1 Two quantifier proofs: a success and a failure

Let us contrast a proof of the theorem $\forall x\,.\,\exists y\,.\,x = y$ with an attempted proof of the non-theorem $\exists y\,.\,\forall x\,.\,x = y$. The former proof succeeds, and the latter fails, because of the scope of quantified variables [12]. 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 happens automatically.

The successful proof. The proof of $\forall x\,.\,\exists y\,.\,x = y$ demonstrates the introduction rules $(\forall I)$ and $(\exists I)$. We state the goal and apply $(\forall I)$:

```
Goal "ALL x. EX y. x=y";
  Level 0
  ALL x. EX y. x = y
   1. ALL x. EX y. x = y
by (resolve_tac [allI] 1);
  Level 1
  ALL x. EX y. x = y
   1. !!x. EX y. x = y
```

The variable `x` is no longer universally quantified, but is a parameter in the subgoal; thus, it is universally quantified at the meta-level. The subgoal must be proved for all possible values of `x`.

To remove the existential quantifier, we apply the rule $(\exists I)$:

```
by (resolve_tac [exI] 1);
  Level 2
  ALL x. EX y. x = y
   1. !!x. x = ?y1(x)
```

The bound variable `y` has become `?y1(x)`. This term consists of the function unknown `?y1` applied to the parameter `x`. Instances of `?y1(x)` may or may not contain `x`. We resolve the subgoal with the reflexivity axiom.

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

Let us consider what has happened in detail. The reflexivity axiom is lifted over x to become $\bigwedge x\,.\,?f(x) = ?f(x)$, which is unified with $\bigwedge x\,.\,x = ?y_1(x)$. The function unknowns $?f$ and $?y_1$ are both instantiated to the identity function, and $x = ?y_1(x)$ collapses to $x = x$ by β-reduction.

The unsuccessful proof. We state the goal $\exists y\,.\,\forall x\,.\,x = y$, which is not a theorem, and try $(\exists I)$:

```
Goal "EX y. ALL x. x=y";
  Level 0
  EX y. ALL x. x = y
   1. EX y. ALL x. x = y
by (resolve_tac [exI] 1);
  Level 1
  EX y. ALL x. x = y
   1. ALL x. x = ?y
```

The unknown `?y` may be replaced by any term, but this can never introduce another bound occurrence of `x`. We now apply $(\forall I)$:

```
by (resolve_tac [allI] 1);
  Level 2
  EX y. ALL x. x = y
   1. !!x. x = ?y
```

Compare our position with the previous Level 2. Instead of `?y1(x)` we have `?y`, whose instances may not contain the bound variable `x`. The reflexivity axiom does not unify with subgoal 1.

```
by (resolve_tac [refl] 1);
  by: tactic failed
```

There can be no proof of $\exists y.\forall x.x = y$ by the soundness of first-order logic. I have elsewhere proved the faithfulness of Isabelle's encoding of first-order logic [12];

there could, of course, be faults in the implementation.

9.2 Nested quantifiers

Multiple quantifiers create complex terms. Proving

$$(\forall x\, y\, .\, P(x, y)) \rightarrow (\forall z\, w\, .\, P(w, z))$$

will demonstrate how parameters and unknowns develop. If they appear in the wrong order, the proof will fail.

This section concludes with a demonstration of `REPEAT` and `ORELSE`.

```
Goal "(ALL x y.P(x,y))  -->  (ALL z w.P(w,z))";
  Level 0
  (ALL x y. P(x,y)) --> (ALL z w. P(w,z))
   1. (ALL x y. P(x,y)) --> (ALL z w. P(w,z))
by (resolve_tac [impI] 1);
  Level 1
  (ALL x y. P(x,y)) --> (ALL z w. P(w,z))
   1. ALL x y. P(x,y) ==> ALL z w. P(w,z)
```

The wrong approach. Using `dresolve_tac`, we apply the rule $(\forall E)$, bound to the ML identifier `spec`. Then we apply $(\forall I)$.

```
by (dresolve_tac [spec] 1);
  Level 2
  (ALL x y. P(x,y)) --> (ALL z w. P(w,z))
   1. ALL y. P(?x1,y) ==> ALL z w. P(w,z)
by (resolve_tac [allI] 1);
  Level 3
  (ALL x y. P(x,y)) --> (ALL z w. P(w,z))
   1. !!z. ALL y. P(?x1,y) ==> ALL w. P(w,z)
```

The unknown `?x1` and the parameter `z` have appeared. We again apply $(\forall E)$ and $(\forall I)$.

```
by (dresolve_tac [spec] 1);
  Level 4
  (ALL x y. P(x,y)) --> (ALL z w. P(w,z))
   1. !!z. P(?x1,?y3(z)) ==> ALL w. P(w,z)
by (resolve_tac [allI] 1);
  Level 5
  (ALL x y. P(x,y)) --> (ALL z w. P(w,z))
   1. !!z w. P(?x1,?y3(z)) ==> P(w,z)
```

The unknown `?y3` and the parameter `w` have appeared. Each unknown is applied to the parameters existing at the time of its creation; instances of `?x1` cannot contain `z` or `w`, while instances of `?y3(z)` can only contain `z`. Due to the restriction on `?x1`, proof by assumption will fail.

```
by (assume_tac 1);
  by: tactic failed
  uncaught exception ERROR
```

The right approach. To do this proof, the rules must be applied in the correct order. Parameters should be created before unknowns. The `choplev` command returns to an earlier stage of the proof; let us return to the result of applying $(\rightarrow I)$:

```
choplev 1;
  Level 1
  (ALL x y. P(x,y)) --> (ALL z w. P(w,z))
   1. ALL x y. P(x,y) ==> ALL z w. P(w,z)
```

Previously we made the mistake of applying $(\forall E)$ before $(\forall I)$.

```
by (resolve_tac [allI] 1);
  Level 2
  (ALL x y. P(x,y)) --> (ALL z w. P(w,z))
   1. !!z. ALL x y. P(x,y) ==> ALL w. P(w,z)
by (resolve_tac [allI] 1);
  Level 3
  (ALL x y. P(x,y)) --> (ALL z w. P(w,z))
   1. !!z w. ALL x y. P(x,y) ==> P(w,z)
```

The parameters `z` and `w` have appeared. We now create the unknowns:

```
by (dresolve_tac [spec] 1);
  Level 4
  (ALL x y. P(x,y)) --> (ALL z w. P(w,z))
   1. !!z w. ALL y. P(?x3(z,w),y) ==> P(w,z)
by (dresolve_tac [spec] 1);
  Level 5
  (ALL x y. P(x,y)) --> (ALL z w. P(w,z))
   1. !!z w. P(?x3(z,w),?y4(z,w)) ==> P(w,z)
```

Both `?x3(z,w)` and `?y4(z,w)` could become any terms containing `z` and `w`:

```
by (assume_tac 1);
  Level 6
  (ALL x y. P(x,y)) --> (ALL z w. P(w,z))
  No subgoals!
```

A one-step proof using tacticals. Repeated application of rules can be effective, but the rules should be attempted in the correct order. Let us return to the original goal using `choplev`:

```
choplev 0;
  Level 0
  (ALL x y. P(x,y)) --> (ALL z w. P(w,z))
   1. (ALL x y. P(x,y)) --> (ALL z w. P(w,z))
```

As we have just seen, `allI` should be attempted before `spec`, while `assume_tac` generally can be attempted first. Such priorities can easily be expressed using `ORELSE`, and repeated using `REPEAT`.

```
by (REPEAT (assume_tac 1 ORELSE resolve_tac [impI,allI] 1
     ORELSE dresolve_tac [spec] 1));
  Level 1
  (ALL x y. P(x,y)) --> (ALL z w. P(w,z))
  No subgoals!
```

9.3 A realistic quantifier proof

To see the practical use of parameters and unknowns, let us prove half of the equivalence

$$(\forall x\,.\,P(x) \to Q) \leftrightarrow ((\exists x\,.\,P(x)) \to Q).$$

We state the left-to-right half to Isabelle in the normal way. Since $\to$ is nested to the right, $(\to I)$ can be applied twice; we use `REPEAT`:

```
Goal "(ALL x.P(x) --> Q) --> (EX x.P(x)) --> Q";
  Level 0
  (ALL x. P(x) --> Q) --> (EX x. P(x)) --> Q
   1. (ALL x. P(x) --> Q) --> (EX x. P(x)) --> Q
by (REPEAT (resolve_tac [impI] 1));
  Level 1
  (ALL x. P(x) --> Q) --> (EX x. P(x)) --> Q
   1. [| ALL x. P(x) --> Q; EX x. P(x) |] ==> Q
```

We can eliminate the universal or the existential quantifier. The existential quantifier should be eliminated first, since this creates a parameter. The rule $(\exists E)$ is bound to the identifier `exE`.

```
by (eresolve_tac [exE] 1);
  Level 2
  (ALL x. P(x) --> Q) --> (EX x. P(x)) --> Q
   1. !!x. [| ALL x. P(x) --> Q; P(x) |] ==> Q
```

The only possibility now is $(\forall E)$, a destruction rule. We use `dresolve_tac`, which discards the quantified assumption; it is only needed once.

```
by (dresolve_tac [spec] 1);
  Level 3
  (ALL x. P(x) --> Q) --> (EX x. P(x)) --> Q
   1. !!x. [| P(x); P(?x3(x)) --> Q |] ==> Q
```

Because we applied $(\exists E)$ before $(\forall E)$, the unknown term `?x3(x)` may depend upon the parameter `x`.

Although $(\rightarrow E)$ is a destruction rule, it works with `eresolve_tac` to perform backward chaining. This technique is frequently useful.

```
by (eresolve_tac [mp] 1);
  Level 4
  (ALL x. P(x) --> Q) --> (EX x. P(x)) --> Q
   1. !!x. P(x) ==> P(?x3(x))
```

The tactic has reduced `Q` to `P(?x3(x))`, deleting the implication. The final step is trivial, thanks to the occurrence of `x`.

```
by (assume_tac 1);
  Level 5
  (ALL x. P(x) --> Q) --> (EX x. P(x)) --> Q
  No subgoals!
```

9.4 The classical reasoner

Although Isabelle cannot compete with fully automatic theorem provers, it provides enough automation to tackle substantial examples. The classical reasoner can be set up for any classical natural deduction logic; see the *Reference Manual*.

Rules are packaged into **classical sets**. The classical reasoner provides several tactics, which apply rules using naive algorithms. Unification handles quantifiers as shown above. The most useful tactic is `Blast_tac`.

Let us solve problems 40 and 60 of Pelletier [16]. (The backslashes `\`...`\` are an ML string escape sequence, to break the long string over two lines.)

```
Goal "(EX y. ALL x. J(y,x) <-> ~J(x,x))  \
\       -->  ~ (ALL x. EX y. ALL z. J(z,y) <-> ~ J(z,x))";
  Level 0
  (EX y. ALL x. J(y,x) <-> ~J(x,x)) -->
  ~(ALL x. EX y. ALL z. J(z,y) <-> ~J(z,x))
   1. (EX y. ALL x. J(y,x) <-> ~J(x,x)) -->
      ~(ALL x. EX y. ALL z. J(z,y) <-> ~J(z,x))
```

`Blast_tac` proves subgoal 1 at a stroke.

```
by (Blast_tac 1);
  Depth = 0
  Depth = 1
  Level 1
  (EX y. ALL x. J(y,x) <-> ~J(x,x)) -->
  ~(ALL x. EX y. ALL z. J(z,y) <-> ~J(z,x))
  No subgoals!
```

Sceptics may examine the proof by calling the package's single-step tactics, such as `step_tac`. This would take up much space, however, so let us proceed to the next example:

```
Goal "ALL x. P(x,f(x)) <-> \
\       (EX y. (ALL z. P(z,y) --> P(z,f(x))) & P(x,y))";
  Level 0
  ALL x. P(x,f(x)) <-> (EX y. (ALL z. P(z,y) --> P(z,f(x))) & P(x,y))
   1. ALL x. P(x,f(x)) <->
      (EX y. (ALL z. P(z,y) --> P(z,f(x))) & P(x,y))
```

Again, subgoal 1 succumbs immediately.

```
by (Blast_tac 1);
  Depth = 0
  Depth = 1
  Level 1
  ALL x. P(x,f(x)) <-> (EX y. (ALL z. P(z,y) --> P(z,f(x))) & P(x,y))
  No subgoals!
```

The classical reasoner is not restricted to the usual logical connectives. The natural deduction rules for unions and intersections resemble those for disjunction and conjunction. The rules for infinite unions and intersections resemble those for quantifiers. Given such rules, the classical reasoner is effective for reasoning in set theory.

Part III
Advanced Methods

Before continuing, it might be wise to try some of your own examples in Isabelle, reinforcing your knowledge of the basic functions.

Look through *Isabelle's Object-Logics* and try proving some simple theorems. You probably should begin with first-order logic (`FOL` or `LK`). Try working some of the examples provided, and others from the literature. Set theory (`ZF`) and Constructive Type Theory (`CTT`) form a richer world for mathematical reasoning and, again, many examples are in the literature. Higher-order logic (`HOL`) is Isabelle's most elaborate logic. Its types and functions are identified with those of the meta-logic.

Choose a logic that you already understand. Isabelle is a proof tool, not a teaching tool; if you do not know how to do a particular proof on paper, then you certainly will not be able to do it on the machine. Even experienced users plan large proofs on paper.

We have covered only the bare essentials of Isabelle, but enough to perform substantial proofs. By occasionally dipping into the *Reference Manual*, you can learn additional tactics, subgoal commands and tacticals.

10 Deriving rules in Isabelle

A mathematical development goes through a progression of stages. Each stage defines some concepts and derives rules about them. We shall see how to derive rules, perhaps involving definitions, using Isabelle. The following section will explain how to declare types, constants, rules and definitions.

10.1 Deriving a rule using tactics and meta-level assumptions

The subgoal module supports the derivation of rules, as discussed in §6.3. When the `Goal` command is supplied a formula of the form $[\![\theta_1; \ldots; \theta_k]\!] \Longrightarrow \phi$, there are two possibilities:

- If all of the premises $\theta_1, \ldots, \theta_k$ are simple formulae (they do not involve the meta-connectives $\bigwedge$ or $\Longrightarrow$) then the command sets the goal to be $[\![\theta_1; \ldots; \theta_k]\!] \Longrightarrow \phi$ and returns the empty list.
- If one or more premises involves the meta-connectives $\bigwedge$ or $\Longrightarrow$, then the command sets the goal to be ϕ and returns a list consisting of the theorems

θ_i $[\theta_i]$, for $i = 1, \ldots, k$. These meta-assumptions are also recorded internally, allowing `result` (which is called by `qed`) to discharge them in the original order.

Rules that discharge assumptions or introduce eigenvariables have complex premises, and the second case applies.

Let us derive $\wedge$ elimination. Until now, calling `Goal` has returned an empty list, which we have ignored. In this example, the list contains the two premises of the rule, since one of them involves the $\Longrightarrow$ connective. We bind them to the ML identifiers `major` and `minor`:[4]

```
val [major,minor] = Goal
    "[| P&Q;  [| P; Q |] ==> R |] ==> R";
  Level 0
  R
   1. R
  val major = "P & Q  [P & Q]" : thm
  val minor = "[| P; Q |] ==> R  [[| P; Q |] ==> R]" : thm
```

Look at the minor premise, recalling that meta-level assumptions are shown in brackets. Using `minor`, we reduce R to the subgoals P and Q:

```
by (resolve_tac [minor] 1);
  Level 1
  R
   1. P
   2. Q
```

Deviating from §6.3, we apply $(\wedge E1)$ forwards from the assumption $P \wedge Q$ to obtain the theorem P $[P \wedge Q]$.

```
major RS conjunct1;
  val it = "P  [P & Q]" : thm

by (resolve_tac [major RS conjunct1] 1);
  Level 2
  R
   1. Q
```

Similarly, we solve the subgoal involving Q.

```
major RS conjunct2;
  val it = "Q  [P & Q]" : thm
by (resolve_tac [major RS conjunct2] 1);
  Level 3
  R
  No subgoals!
```

[4]Some ML compilers will print a message such as *binding not exhaustive*. This warns that `Goal` must return a 2-element list. Otherwise, the pattern-match will fail; ML will raise exception `Match`.

Calling `topthm` returns the current proof state as a theorem. Note that it contains assumptions. Calling `qed` discharges the assumptions — both occurrences of $P \wedge Q$ are discharged as one — and makes the variables schematic.

```
topthm();
  val it = "R  [P & Q, P & Q, [| P; Q |] ==> R]" : thm
qed "conjE";
  val conjE = "[| ?P & ?Q; [| ?P; ?Q |] ==> ?R |] ==> ?R" : thm
```

10.2 Definitions and derived rules

Definitions are expressed as meta-level equalities. Let us define negation and the if-and-only-if connective:

$$\begin{aligned} \neg ?P &\equiv ?P \rightarrow \bot \\ ?P \leftrightarrow ?Q &\equiv (?P \rightarrow ?Q) \wedge (?Q \rightarrow ?P) \end{aligned}$$

Isabelle permits **meta-level rewriting** using definitions such as these. **Unfolding** replaces every instance of $\neg ?P$ by the corresponding instance of $?P \rightarrow \bot$. For example, $\forall x \,.\, \neg(P(x) \wedge \neg R(x,0))$ unfolds to

$$\forall x \,.\, (P(x) \wedge R(x,0) \rightarrow \bot) \rightarrow \bot.$$

Folding a definition replaces occurrences of the right-hand side by the left. The occurrences need not be free in the entire formula.

When you define new concepts, you should derive rules asserting their abstract properties, and then forget their definitions. This supports modularity: if you later change the definitions without affecting their abstract properties, then most of your proofs will carry through without change. Indiscriminate unfolding makes a subgoal grow exponentially, becoming unreadable.

Taking this point of view, Isabelle does not unfold definitions automatically during proofs. Rewriting must be explicit and selective. Isabelle provides tactics and meta-rules for rewriting, and a version of the `Goal` command that unfolds the conclusion and premises of the rule being derived.

For example, the intuitionistic definition of negation given above may seem peculiar. Using Isabelle, we shall derive pleasanter negation rules:

$$\frac{\begin{matrix}[P] \\ \vdots \\ \bot\end{matrix}}{\neg P}\,(\neg I) \qquad \frac{\neg P \quad P}{Q}\,(\neg E)$$

This requires proving the following meta-formulae:

$$(P \Longrightarrow \bot) \Longrightarrow \neg P \qquad (\neg I)$$

$$[\![\neg P; P]\!] \Longrightarrow Q. \qquad (\neg E)$$

10.3 Deriving the ¬ introduction rule

To derive $(\neg I)$, we may call `Goal` with the appropriate formula. Again, the rule's premises involve a meta-connective, and `Goal` returns one-element list. We bind this list to the ML identifier `prems`.

```
val prems = Goal "(P ==> False) ==> ~P";
  Level 0
  ~P
   1. ~P
  val prems = ["P ==> False  [P ==> False]"] : thm list
```

Calling `rewrite_goals_tac` with `not_def`, which is the definition of negation, unfolds that definition in the subgoals. It leaves the main goal alone.

```
not_def;
  val it = "~?P == ?P --> False" : thm
by (rewrite_goals_tac [not_def]);
  Level 1
  ~P
   1. P --> False
```

Using `impI` and the premise, we reduce subgoal 1 to a triviality:

```
by (resolve_tac [impI] 1);
  Level 2
  ~P
   1. P ==> False
by (resolve_tac prems 1);
  Level 3
  ~P
   1. P ==> P
```

The rest of the proof is routine. Note the form of the final result.

```
by (assume_tac 1);
  Level 4
  ~P
  No subgoals!
qed "notI";
  val notI = "(?P ==> False) ==> ~?P" : thm
```

There is a simpler way of conducting this proof. The `Goalw` command starts a backward proof, as does `Goal`, but it also unfolds definitions. Thus there is no need to call `rewrite_goals_tac`:

```
val prems = Goalw [not_def]
    "(P ==> False) ==> ~P";
  Level 0
  ~P
   1. P --> False
  val prems = ["P ==> False  [P ==> False]"] : thm list
```

10.4 Deriving the ¬ elimination rule

Let us derive the rule $(\neg E)$. The proof follows that of `conjE` above, with an additional step to unfold negation in the major premise. The `Goalw` command is best for this: it unfolds definitions not only in the conclusion but the premises.

```
Goalw [not_def] "[| ~P;  P |] ==> R";
  Level 0
  [| ~ P; P |] ==> R
   1. [| P --> False; P |] ==> R
```

As the first step, we apply `FalseE`:

```
by (resolve_tac [FalseE] 1);
  Level 1
  [| ~ P; P |] ==> R
   1. [| P --> False; P |] ==> False
```

Everything follows from falsity. And we can prove falsity using the premises and Modus Ponens:

```
by (eresolve_tac [mp] 1);
  Level 2
  [| ~ P; P |] ==> R
   1. P ==> P
by (assume_tac 1);
  Level 3
  [| ~ P; P |] ==> R
  No subgoals!
qed "notE";
  val notE = "[| ~?P; ?P |] ==> ?R" : thm
```

`Goalw` unfolds definitions in the premises even when it has to return them as a list. Another way of unfolding definitions in a theorem is by applying the function `rewrite_rule`.

11 Defining theories

Isabelle makes no distinction between simple extensions of a logic — like specifying a type *bool* with constants *true* and *false* — and defining an entire logic. A theory definition has a form like

T `=` S_1 `+` $\cdots$ `+` S_n `+`	
`classes`	*class declarations*
`default`	*sort*
`types`	*type declarations and synonyms*
`arities`	*type arity declarations*
`consts`	*constant declarations*
`syntax`	*syntactic constant declarations*
`translations`	*ast translation rules*
`defs`	*meta-logical definitions*
`rules`	*rule declarations*
`end`	
`ML`	*ML code*

This declares the theory T to extend the existing theories S_1, …, S_n. It may introduce new classes, types, arities (of existing types), constants and rules; it can specify the default sort for type variables. A constant declaration can specify an associated concrete syntax. The translations section specifies rewrite rules on abstract syntax trees, handling notations and abbreviations. The `ML` section may contain code to perform arbitrary syntactic transformations. The main declaration forms are discussed below. There are some more sections not presented here, the full syntax can be found in an appendix of the *Reference Manual*. Also note that object-logics may add further theory sections, for example `typedef`, `datatype` in `HOL`.

All the declaration parts can be omitted or repeated and may appear in any order, except that the ML section must be last (after the `end` keyword). In the simplest case, T is just the union of S_1, …, S_n. New theories always extend one or more other theories, inheriting their types, constants, syntax, etc. The theory `Pure` contains nothing but Isabelle's meta-logic. The variant `CPure` offers the more usual higher-order function application syntax $t\,u_1 \ldots u_n$ instead of $t(u_1, \ldots, u_n)$ in `Pure`.

Each theory definition must reside in a separate file, whose name is the theory's with `.thy` appended. Calling `use_thy "`T`"` reads the definition from T`.thy`, writes a corresponding file of ML code `.`T`.thy.ML`, reads the latter file, and deletes it if no errors occurred. This declares the ML structure T, which contains a component `thy` denoting the new theory, a component for each rule, and everything declared in *ML code*.

Errors may arise during the translation to ML (say, a misspelled keyword) or during creation of the new theory (say, a type error in a rule). But if all goes well, `use_thy` will finally read the file T`.ML` (if it exists). This file typically contains proofs that refer to the components of T. The structure is automatically opened, so its components may be referred to by unqualified names, e.g. just `thy` instead of T`.thy`.

`use_thy` automatically loads a theory's parents before loading the theory itself. When a theory file is modified, many theories may have to be reloaded. Isabelle records the modification times and dependencies of theory files. See the *Reference Manual* for more details.

11.1 Declaring constants, definitions and rules

Most theories simply declare constants, definitions and rules. The **constant declaration part** has the form

$$\begin{array}{ll} \texttt{consts} & c_1 \texttt{ :: } \tau_1 \\ & \vdots \\ & c_n \texttt{ :: } \tau_n \end{array}$$

where c_1, ..., c_n are constants and τ_1, ..., τ_n are types. The types must be enclosed in quotation marks if they contain user-declared infix type constructors like `*`. Each constant must be enclosed in quotation marks unless it is a valid identifier. To declare c_1, ..., c_n as constants of type τ, the n declarations may be abbreviated to a single line:

$$c_1\texttt{, ..., } c_n \texttt{ :: } \tau$$

The **rule declaration part** has the form

$$\begin{array}{ll} \texttt{rules} & id_1 \texttt{ "}rule_1\texttt{"} \\ & \vdots \\ & id_n \texttt{ "}rule_n\texttt{"} \end{array}$$

where id_1, ..., id_n are ML identifiers and $rule_1$, ..., $rule_n$ are expressions of type *prop*. Each rule *must* be enclosed in quotation marks.

The **definition part** is similar, but with the keyword `defs` instead of `rules`. **Definitions** are rules of the form $s \equiv t$, and should serve only as abbreviations. The simplest form of a definition is $f \equiv t$, where f is a constant. Also allowed are η-equivalent forms of this, where the arguments of f appear applied on the left-hand side of the equation instead of abstracted on the right-hand side.

Isabelle checks for common errors in definitions, such as extra variables on the right-hand side, but currently does not a complete test of well-formedness. Thus determined users can write non-conservative 'definitions' by using mutual recursion, for example; the consequences of such actions are their responsibility.

This example theory extends first-order logic by declaring and defining two constants, *nand* and *xor*:

```
Gate = FOL +
consts  nand,xor :: [o,o] => o
defs    nand_def "nand(P,Q) == ~(P & Q)"
        xor_def  "xor(P,Q)  == P & ~Q | ~P & Q"
end
```

Declaring and defining constants can be combined:

```
Gate = FOL +
constdefs  nand :: [o,o] => o
           "nand(P,Q) == ~(P & Q)"
           xor  :: [o,o] => o
           "xor(P,Q)  == P & ~Q | ~P & Q"
end
```

`constdefs` generates the names `nand_def` and `xor_def` automatically, which is why it is restricted to alphanumeric identifiers. In general it has the form

constdefs id_1 :: τ_1
"$id_1 \equiv \ldots$"
$\vdots$
id_n :: τ_n
"$id_n \equiv \ldots$"

! A common mistake when writing definitions is to introduce extra free variables on the right-hand side as in the following fictitious definition:

```
defs  prime_def "prime(p) == (m divides p) --> (m=1 | m=p)"
```

Isabelle rejects this "definition" because of the extra `m` on the right-hand side, which would introduce an inconsistency. What you should have written is

```
defs  prime_def "prime(p) == ALL m. (m divides p) --> (m=1 | m=p)"
```

11.2 Declaring type constructors

Types are composed of type variables and **type constructors**. Each type constructor takes a fixed number of arguments. They are declared with an ML-like syntax. If *list* takes one type argument, *tree* takes two arguments and *nat* takes no arguments, then these type constructors can be declared by

```
types 'a list
      ('a,'b) tree
      nat
```

The **type declaration part** has the general form

$$\begin{array}{ll} \texttt{types} & \mathit{tids}_1 \;\; \mathit{id}_1 \\ & \vdots \\ & \mathit{tids}_n \;\; \mathit{id}_n \end{array}$$

where id_1, …, id_n are identifiers and $tids_1$, …, $tids_n$ are type argument lists as shown in the example above. It declares each id_i as a type constructor with the specified number of argument places.

The **arity declaration part** has the form

$$\begin{array}{ll} \texttt{arities} & \mathit{tycon}_1 \;\texttt{::}\; \mathit{arity}_1 \\ & \vdots \\ & \mathit{tycon}_n \;\texttt{::}\; \mathit{arity}_n \end{array}$$

where $tycon_1$, …, $tycon_n$ are identifiers and $arity_1$, …, $arity_n$ are arities. Arity declarations add arities to existing types; they do not declare the types themselves. In the simplest case, for an 0-place type constructor, an arity is simply the type's class. Let us declare a type *bool* of class *term*, with constants *tt* and *ff*. (In first-order logic, booleans are distinct from formulae, which have type $o :: logic$.)

```
Bool = FOL +
types   bool
arities bool    :: term
consts  tt,ff   :: bool
end
```

A k-place type constructor may have arities of the form $(s_1, \ldots, s_k)c$, where $s_1, \ldots, s_n$ are sorts and c is a class. Each sort specifies a type argument; it has the form $\{c_1, \ldots, c_m\}$, where c_1, …, c_m are classes. Mostly we deal with singleton sorts, and may abbreviate them by dropping the braces. The arity $(term)term$ is short for $(\{term\})term$. Recall the discussion in §1.2.

A type constructor may be overloaded (subject to certain conditions) by appearing in several arity declarations. For instance, the function type constructor *fun* has the arity $(logic, logic)logic$; in higher-order logic, it is declared also to have arity $(term, term)term$.

Theory `List` declares the 1-place type constructor *list*, gives it the arity (*term*)*term*, and declares constants *Nil* and *Cons* with polymorphic types:[5]

```
List = FOL +
types   'a list
arities list    :: (term)term
consts  Nil     :: 'a list
        Cons    :: ['a, 'a list] => 'a list
end
```

Multiple arity declarations may be abbreviated to a single line:

`arities` $tycon_1$ `, ...,` $tycon_n$ `::` *arity*

11.3 Type synonyms

Isabelle supports **type synonyms** (**abbreviations**) which are similar to those found in ML. Such synonyms are defined in the type declaration part and are fairly self explanatory:

```
types gate       = [o,o] => o
      'a pred    = 'a => o
      ('a,'b)nuf = 'b => 'a
```

Type declarations and synonyms can be mixed arbitrarily:

```
types nat
      'a stream = nat => 'a
      signal    = nat stream
      'a list
```

A synonym is merely an abbreviation for some existing type expression. Hence synonyms may not be recursive! Internally all synonyms are fully expanded. As a consequence Isabelle output never contains synonyms. Their main purpose is to improve the readability of theory definitions. Synonyms can be used just like any other type:

```
consts and,or :: gate
       negate :: signal => signal
```

[5]In the `consts` part, type variable `'a` has the default sort, which is `term`. See the *Reference Manual* for more information.

11.4 Infix and mixfix operators

Infix or mixfix syntax may be attached to constants. Consider the following theory:

```
Gate2 = FOL +
consts  "~&"     :: [o,o] => o         (infixl 35)
        "#"      :: [o,o] => o         (infixl 30)
defs    nand_def "P ~& Q == ~(P & Q)"
        xor_def  "P # Q  == P & ~Q | ~P & Q"
end
```

The constant declaration part declares two left-associating infix operators with their priorities, or precedences; they are $\neg\&$ of priority 35 and # of priority 30. Hence $P \# Q \# R$ is parsed as $(P \# Q) \# R$ and $P \# Q \neg\& R$ as $P \# (Q \neg\& R)$. Note the quotation marks in `"~&"` and `"#"`.

The constants `op ~&` and `op #` are declared automatically, just as in ML. Hence you may write propositions like `op #(True) == op ~&(True)`, which asserts that the functions $\lambda Q\,.\,\mathit{True} \# Q$ and $\lambda Q\,.\,\mathit{True} \neg\& Q$ are identical.

Infix syntax and constant names may be also specified independently. For example, consider this version of $\neg\&$:

```
consts  nand     :: [o,o] => o         (infixl "~&" 35)
```

Mixfix operators may have arbitrary context-free syntaxes. Let us add a line to the constant declaration part:

```
        If :: [o,o,o] => o       ("if _ then _ else _")
```

This declares a constant *If* of type $[o, o, o] \Rightarrow o$ with concrete syntax `if` P `then` Q `else` R as well as `If(`P`,`Q`,`R`)`. Underscores denote argument positions.

The declaration above does not allow the `if-then-else` construct to be printed split across several lines, even if it is too long to fit on one line. Pretty-printing information can be added to specify the layout of mixfix operators. For details, see the *Reference Manual*, chapter 'Defining Logics'.

Mixfix declarations can be annotated with priorities, just like infixes. The example above is just a shorthand for

```
        If :: [o,o,o] => o       ("if _ then _ else _" [0,0,0] 1000)
```

The numeric components determine priorities. The list of integers defines, for each argument position, the minimal priority an expression at that position must have. The final integer is the priority of the construct itself. In the example above, any argument expression is acceptable because priorities are non-negative, and conditionals may appear everywhere because 1000 is the highest priority. On the other hand, the declaration

```
If :: [o,o,o] => o       ("if _ then _ else _" [100,0,0] 99)
```

defines concrete syntax for a conditional whose first argument cannot have the form `if` P `then` Q `else` R because it must have a priority of at least 100. We may of course write

`if (if` P `then` Q `else` R`) then` S `else` T

because expressions in parentheses have maximal priority.

Binary type constructors, like products and sums, may also be declared as infixes. The type declaration below introduces a type constructor $*$ with infix notation $\alpha * \beta$, together with the mixfix notation <_,_> for pairs. We also see a rule declaration part.

```
Prod = FOL +
types   ('a,'b) "*"                       (infixl 20)
arities "*"     :: (term,term)term
consts  fst     :: "'a * 'b => 'a"
        snd     :: "'a * 'b => 'b"
        Pair    :: "['a,'b] => 'a * 'b"   ("(1<_,/_>)")
rules   fst     "fst(<a,b>) = a"
        snd     "snd(<a,b>) = b"
end
```

! The name of the type constructor is `*` and not `op *`, as it would be in the case of an infix constant. Only infix type constructors can have symbolic names like `*`. General mixfix syntax for types may be introduced via appropriate `syntax` declarations.

11.5 Overloading

The **class declaration part** has the form

`classes` id_1 `<` c_1
$\vdots$
id_n `<` c_n

where id_1, …, id_n are identifiers and c_1, …, c_n are existing classes. It declares each id_i as a new class, a subclass of c_i. In the general case, an identifier may be declared to be a subclass of k existing classes:

id `<` c_1`,` `...`, c_k

Type classes allow constants to be overloaded. As suggested in §1.2, let us define the class *arith* of arithmetic types with the constants $+ :: [\alpha, \alpha] \Rightarrow \alpha$ and

$0, 1{::}\alpha$, for $\alpha{::}arith$. We introduce *arith* as a subclass of *term* and add the three polymorphic constants of this class.

```
Arith = FOL +
classes arith < term
consts  "0"     :: 'a::arith                  ("0")
        "1"     :: 'a::arith                  ("1")
        "+"     :: ['a::arith,'a] => 'a       (infixl 60)
end
```

No rules are declared for these constants: we merely introduce their names without specifying properties. On the other hand, classes with rules make it possible to prove **generic** theorems. Such theorems hold for all instances, all types in that class.

We can now obtain distinct versions of the constants of *arith* by declaring certain types to be of class *arith*. For example, let us declare the 0-place type constructors *bool* and *nat*:

```
BoolNat = Arith +
types   bool  nat
arities bool, nat   :: arith
consts  Suc         :: nat=>nat
rules   add0        "0 + n = n::nat"
        addS        "Suc(m)+n = Suc(m+n)"
        nat1        "1 = Suc(0)"
        or0l        "0 + x = x::bool"
        or0r        "x + 0 = x::bool"
        or1l        "1 + x = 1::bool"
        or1r        "x + 1 = 1::bool"
end
```

Because *nat* and *bool* have class *arith*, we can use 0, 1 and + at either type. The type constraints in the axioms are vital. Without constraints, the x in $1 + x = x$ would have type $\alpha{::}arith$ and the axiom would hold for any type of class *arith*. This would collapse *nat* to a trivial type:

$$Suc(1) = Suc(0 + 1) = Suc(0) + 1 = 1 + 1 = 1!$$

12 Theory example: the natural numbers

We shall now work through a small example of formalized mathematics demonstrating many of the theory extension features.

12.1 Extending first-order logic with the natural numbers

Section 1 has formalized a first-order logic, including a type *nat* and the constants $0 :: nat$ and $Suc :: nat \Rightarrow nat$. Let us introduce the Peano axioms for mathematical induction and the freeness of 0 and *Suc*:

$$\frac{P[0/x] \quad \begin{matrix}[P] \\ \vdots \\ P[Suc(x)/x]\end{matrix}}{P[n/x]}\ (induct) \qquad \text{provided } x \text{ is not free in any assumption except } P$$

$$\frac{Suc(m) = Suc(n)}{m = n}\ (Suc\_inject) \qquad \frac{Suc(m) = 0}{R}\ (Suc\_neq\_0)$$

Mathematical induction asserts that $P(n)$ is true, for any $n :: nat$, provided $P(0)$ holds and that $P(x)$ implies $P(Suc(x))$ for all x. Some authors express the induction step as $\forall x\,.\,P(x) \to P(Suc(x))$. To avoid making induction require the presence of other connectives, we formalize mathematical induction as

$$[\![P(0); \bigwedge x\,.\,P(x) \Longrightarrow P(Suc(x))]\!] \Longrightarrow P(n). \qquad (induct)$$

Similarly, to avoid expressing the other rules using $\forall$, $\to$ and $\neg$, we take advantage of the meta-logic;[6] (Suc_neq_0) is an elimination rule for $Suc(m) = 0$:

$$Suc(m) = Suc(n) \Longrightarrow m = n \qquad (Suc\_inject)$$

$$Suc(m) = 0 \Longrightarrow R \qquad (Suc\_neq\_0)$$

We shall also define a primitive recursion operator, *rec*. Traditionally, primitive recursion takes a natural number a and a 2-place function f, and obeys the equations

$$\begin{array}{rcl} rec(0, a, f) & = & a \\ rec(Suc(m), a, f) & = & f(m, rec(m, a, f)) \end{array}$$

Addition, defined by $m + n \equiv rec(m, n, \lambda x\, y\,.\, Suc(y))$, should satisfy

$$\begin{array}{rcl} 0 + n & = & n \\ Suc(m) + n & = & Suc(m + n) \end{array}$$

Primitive recursion appears to pose difficulties: first-order logic has no function-valued expressions. We again take advantage of the meta-logic, which does have functions. We also generalise primitive recursion to be polymorphic over any type of class *term*, and declare the addition function:

$$\begin{array}{rcl} rec & :: & [nat, \alpha{::}term, [nat, \alpha] \Rightarrow \alpha] \Rightarrow \alpha \\ + & :: & [nat, nat] \Rightarrow nat \end{array}$$

[6] On the other hand, the axioms $Suc(m) = Suc(n) \leftrightarrow m = n$ and $\neg(Suc(m) = 0)$ are logically equivalent to those given, and work better with Isabelle's simplifier.

12.2 Declaring the theory to Isabelle

Let us create the theory Nat starting from theory FOL, which contains only classical logic with no natural numbers. We declare the 0-place type constructor *nat* and the associated constants. Note that the constant 0 requires a mixfix annotation because 0 is not a legal identifier, and could not otherwise be written in terms:

```
Nat = FOL +
types   nat
arities nat         :: term
consts  "0"         :: nat                              ("0")
        Suc         :: nat=>nat
        rec         :: [nat, 'a, [nat,'a]=>'a] => 'a
        "+"         :: [nat, nat] => nat                (infixl 60)
rules   Suc_inject  "Suc(m)=Suc(n) ==> m=n"
        Suc_neq_0   "Suc(m)=0      ==> R"
        induct      "[| P(0);  !!x. P(x) ==> P(Suc(x)) |]  ==> P(n)"
        rec_0       "rec(0,a,f) = a"
        rec_Suc     "rec(Suc(m), a, f) = f(m, rec(m,a,f))"
        add_def     "m+n == rec(m, n, %x y. Suc(y))"
end
```

In axiom add_def, recall that % stands for λ. Loading this theory file creates the ML structure Nat, which contains the theory and axioms.

12.3 Proving some recursion equations

Theory FOL/ex/Nat contains proofs involving this theory of the natural numbers. As a trivial example, let us derive recursion equations for +. Here is the zero case:

```
Goalw [add_def] "0+n = n";
  Level 0
  0 + n = n
   1. rec(0,n,%x y. Suc(y)) = n

by (resolve_tac [rec_0] 1);
  Level 1
  0 + n = n
  No subgoals!
qed "add_0";
```

And here is the successor case:

```
Goalw [add_def] "Suc(m)+n = Suc(m+n)";
  Level 0
  Suc(m) + n = Suc(m + n)
   1. rec(Suc(m),n,%x y. Suc(y)) = Suc(rec(m,n,%x y. Suc(y)))
```

```
by (resolve_tac [rec_Suc] 1);
  Level 1
  Suc(m) + n = Suc(m + n)
  No subgoals!
qed "add_Suc";
```

The induction rule raises some complications, which are discussed next.

13 Refinement with explicit instantiation

In order to employ mathematical induction, we need to refine a subgoal by the rule (*induct*). The conclusion of this rule is $?P(?n)$, which is highly ambiguous in higher-order unification. It matches every way that a formula can be regarded as depending on a subterm of type *nat*. To get round this problem, we could make the induction rule conclude $\forall n \,.\, ?P(n)$ — but putting a subgoal into this form requires refinement by $(\forall E)$, which is equally hard!

The tactic `res_inst_tac`, like `resolve_tac`, refines a subgoal by a rule. But it also accepts explicit instantiations for the rule's schematic variables.

`res_inst_tac` *insts thm i* instantiates the rule *thm* with the instantiations *insts*, and then performs resolution on subgoal i.

`eres_inst_tac` and `dres_inst_tac` are similar, but perform elim-resolution and destruct-resolution, respectively.

The list *insts* consists of pairs $[(v_1, e_1), \ldots, (v_n, e_n)]$, where $v_1, \ldots, v_n$ are names of schematic variables in the rule — with no leading question marks! — and $e_1, \ldots, e_n$ are expressions giving their instantiations. The expressions are type-checked in the context of a particular subgoal: free variables receive the same types as they have in the subgoal, and parameters may appear. Type variable instantiations may appear in *insts*, but they are seldom required: `res_inst_tac` instantiates type variables automatically whenever the type of e_i is an instance of the type of $?v_i$.

13.1 A simple proof by induction

Let us prove that no natural number k equals its own successor. To use (*induct*), we instantiate $?n$ to k; Isabelle finds a good instantiation for $?P$.

```
Goal "~ (Suc(k) = k)";
  Level 0
  Suc(k) ~= k
   1. Suc(k) ~= k
```

```
by (res_inst_tac [("n","k")] induct 1);
  Level 1
  Suc(k) ~= k
   1. Suc(0) ~= 0
   2. !!x. Suc(x) ~= x ==> Suc(Suc(x)) ~= Suc(x)
```

We should check that Isabelle has correctly applied induction. Subgoal 1 is the base case, with k replaced by 0. Subgoal 2 is the inductive step, with k replaced by $Suc(x)$ and with an induction hypothesis for x. The rest of the proof demonstrates `notI`, `notE` and the other rules of theory `Nat`. The base case holds by `Suc_neq_0`:

```
by (resolve_tac [notI] 1);
  Level 2
  Suc(k) ~= k
   1. Suc(0) = 0 ==> False
   2. !!x. Suc(x) ~= x ==> Suc(Suc(x)) ~= Suc(x)
by (eresolve_tac [Suc_neq_0] 1);
  Level 3
  Suc(k) ~= k
   1. !!x. Suc(x) ~= x ==> Suc(Suc(x)) ~= Suc(x)
```

The inductive step holds by the contrapositive of `Suc_inject`. Negation rules transform the subgoal into that of proving $Suc(x) = x$ from $Suc(Suc(x)) = Suc(x)$:

```
by (resolve_tac [notI] 1);
  Level 4
  Suc(k) ~= k
   1. !!x. [| Suc(x) ~= x; Suc(Suc(x)) = Suc(x) |] ==> False
by (eresolve_tac [notE] 1);
  Level 5
  Suc(k) ~= k
   1. !!x. Suc(Suc(x)) = Suc(x) ==> Suc(x) = x
by (eresolve_tac [Suc_inject] 1);
  Level 6
  Suc(k) ~= k
  No subgoals!
```

13.2 An example of ambiguity in `resolve_tac`

If you try the example above, you may observe that `res_inst_tac` is not actually needed. Almost by chance, `resolve_tac` finds the right instantiation for (*induct*) to yield the desired next state. With more complex formulae, our luck fails.

```
Goal "(k+m)+n = k+(m+n)";
  Level 0
  k + m + n = k + (m + n)
   1. k + m + n = k + (m + n)
by (resolve_tac [induct] 1);
  Level 1
  k + m + n = k + (m + n)
   1. k + m + n = 0
   2. !!x. k + m + n = x ==> k + m + n = Suc(x)
```

This proof requires induction on k. The occurrence of 0 in subgoal 1 indicates that induction has been applied to the term $k+(m+n)$; this application is sound but will not lead to a proof here. Fortunately, Isabelle can (lazily!) generate all the valid applications of induction. The `back` command causes backtracking to an alternative outcome of the tactic.

```
back();
  Level 1
  k + m + n = k + (m + n)
   1. k + m + n = k + 0
   2. !!x. k + m + n = k + x ==> k + m + n = k + Suc(x)
```

Now induction has been applied to $m + n$. This is equally useless. Let us call `back` again.

```
back();
  Level 1
  k + m + n = k + (m + n)
   1. k + m + 0 = k + (m + 0)
   2. !!x. k + m + x = k + (m + x) ==>
           k + m + Suc(x) = k + (m + Suc(x))
```

Now induction has been applied to n. What is the next alternative?

```
back();
  Level 1
  k + m + n = k + (m + n)
   1. k + m + n = k + (m + 0)
   2. !!x. k + m + n = k + (m + x) ==> k + m + n = k + (m + Suc(x))
```

Inspecting subgoal 1 reveals that induction has been applied to just the second occurrence of n. This perfectly legitimate induction is useless here.

The main goal admits fourteen different applications of induction. The number is exponential in the size of the formula.

13.3 Proving that addition is associative

Let us invoke the induction rule properly, using `res_inst_tac`. At the same time, we shall have a glimpse at Isabelle's simplification tactics, which are described in the *Reference Manual*.

Isabelle's simplification tactics repeatedly apply equations to a subgoal, perhaps proving it. For efficiency, the rewrite rules must be packaged into a **simplification set**, or **simpset**. We augment the implicit simpset of `FOL` with the equations proved in the previous section, namely $0 + n = n$ and $\texttt{Suc}(m) + n = \texttt{Suc}(m + n)$:

```
Addsimps [add_0, add_Suc];
```

We state the goal for associativity of addition, and use `res_inst_tac` to invoke induction on k:

```
Goal "(k+m)+n = k+(m+n)";
  Level 0
  k + m + n = k + (m + n)
   1. k + m + n = k + (m + n)
by (res_inst_tac [("n","k")] induct 1);
  Level 1
  k + m + n = k + (m + n)
   1. 0 + m + n = 0 + (m + n)
   2. !!x. x + m + n = x + (m + n) ==>
           Suc(x) + m + n = Suc(x) + (m + n)
```

The base case holds easily; both sides reduce to $m + n$. The tactic `Simp_tac` rewrites with respect to the current simplification set, applying the rewrite rules for addition:

```
by (Simp_tac 1);
  Level 2
  k + m + n = k + (m + n)
   1. !!x. x + m + n = x + (m + n) ==>
           Suc(x) + m + n = Suc(x) + (m + n)
```

The inductive step requires rewriting by the equations for addition together the induction hypothesis, which is also an equation. The tactic `Asm_simp_tac` rewrites using the implicit simplification set and any useful assumptions:

```
by (Asm_simp_tac 1);
  Level 3
  k + m + n = k + (m + n)
  No subgoals!
```

14 A Prolog interpreter

To demonstrate the power of tacticals, let us construct a Prolog interpreter and execute programs involving lists.[7] The Prolog program consists of a theory. We declare a type constructor for lists, with an arity declaration to say that $(\tau)list$ is of class *term* provided τ is:

$$list \quad :: \quad (term)term$$

We declare four constants: the empty list *Nil*; the infix list constructor :; the list concatenation predicate *app*; the list reverse predicate *rev*. (In Prolog, functions on lists are expressed as predicates.)

$$\begin{aligned} Nil &:: \alpha list \\ : &:: [\alpha, \alpha list] \Rightarrow \alpha list \\ app &:: [\alpha list, \alpha list, \alpha list] \Rightarrow o \\ rev &:: [\alpha list, \alpha list] \Rightarrow o \end{aligned}$$

The predicate *app* should satisfy the Prolog-style rules

$$app(Nil, ys, ys) \qquad \frac{app(xs, ys, zs)}{app(x : xs, ys, x : zs)}$$

We define the naive version of *rev*, which calls *app*:

$$rev(Nil, Nil) \qquad \frac{rev(xs, ys) \quad app(ys, x : Nil, zs)}{rev(x : xs, zs)}$$

Theory `Prolog` extends first-order logic in order to make use of the class *term* and the type o. The interpreter does not use the rules of `FOL`.

```
Prolog = FOL +
types   'a list
arities list    :: (term)term
consts  Nil     :: 'a list
        ":"     :: ['a, 'a list]=> 'a list            (infixr 60)
        app     :: ['a list, 'a list, 'a list] => o
        rev     :: ['a list, 'a list] => o
rules   appNil  "app(Nil,ys,ys)"
        appCons "app(xs,ys,zs) ==> app(x:xs, ys, x:zs)"
        revNil  "rev(Nil,Nil)"
        revCons "[| rev(xs,ys); app(ys,x:Nil,zs) |] ==> rev(x:xs,zs)"
end
```

[7]To run these examples, see the file `FOL/ex/Prolog.ML`.

14.1 Simple executions

Repeated application of the rules solves Prolog goals. Let us append the lists $[a, b, c]$ and $[d, e]$. As the rules are applied, the answer builds up in `?x`.

```
Goal "app(a:b:c:Nil, d:e:Nil, ?x)";
  Level 0
  app(a : b : c : Nil, d : e : Nil, ?x)
   1. app(a : b : c : Nil, d : e : Nil, ?x)
by (resolve_tac [appNil,appCons] 1);
  Level 1
  app(a : b : c : Nil, d : e : Nil, a : ?zs1)
   1. app(b : c : Nil, d : e : Nil, ?zs1)
by (resolve_tac [appNil,appCons] 1);
  Level 2
  app(a : b : c : Nil, d : e : Nil, a : b : ?zs2)
   1. app(c : Nil, d : e : Nil, ?zs2)
```

At this point, the first two elements of the result are a and b.

```
by (resolve_tac [appNil,appCons] 1);
  Level 3
  app(a : b : c : Nil, d : e : Nil, a : b : c : ?zs3)
   1. app(Nil, d : e : Nil, ?zs3)
by (resolve_tac [appNil,appCons] 1);
  Level 4
  app(a : b : c : Nil, d : e : Nil, a : b : c : d : e : Nil)
  No subgoals!
```

Prolog can run functions backwards. Which list can be appended with $[c, d]$ to produce $[a, b, c, d]$? Using `REPEAT`, we find the answer at once, $[a, b]$:

```
Goal "app(?x, c:d:Nil, a:b:c:d:Nil)";
  Level 0
  app(?x, c : d : Nil, a : b : c : d : Nil)
   1. app(?x, c : d : Nil, a : b : c : d : Nil)
by (REPEAT (resolve_tac [appNil,appCons] 1));
  Level 1
  app(a : b : Nil, c : d : Nil, a : b : c : d : Nil)
  No subgoals!
```

14.2 Backtracking

Prolog backtracking can answer questions that have multiple solutions. Which lists x and y can be appended to form the list $[a, b, c, d]$? This question has five solutions. Using `REPEAT` to apply the rules, we quickly find the first solution, namely $x = []$ and $y = [a, b, c, d]$:

```
Goal "app(?x, ?y, a:b:c:d:Nil)";
  Level 0
  app(?x, ?y, a : b : c : d : Nil)
   1. app(?x, ?y, a : b : c : d : Nil)
by (REPEAT (resolve_tac [appNil,appCons] 1));
  Level 1
  app(Nil, a : b : c : d : Nil, a : b : c : d : Nil)
  No subgoals!
```

Isabelle can lazily generate all the possibilities. The `back` command returns the tactic's next outcome, namely $x = [a]$ and $y = [b, c, d]$:

```
back();
  Level 1
  app(a : Nil, b : c : d : Nil, a : b : c : d : Nil)
  No subgoals!
```

The other solutions are generated similarly.

```
back();
  Level 1
  app(a : b : Nil, c : d : Nil, a : b : c : d : Nil)
  No subgoals!
back();
  Level 1
  app(a : b : c : Nil, d : Nil, a : b : c : d : Nil)
  No subgoals!
back();
  Level 1
  app(a : b : c : d : Nil, Nil, a : b : c : d : Nil)
  No subgoals!
```

14.3 Depth-first search

Now let us try *rev*, reversing a list. Bundle the rules together as the ML identifier `rules`. Naive reverse requires 120 inferences for this 14-element list, but the tactic terminates in a few seconds.

```
Goal "rev(a:b:c:d:e:f:g:h:i:j:k:l:m:n:Nil, ?w)";
  Level 0
  rev(a : b : c : d : e : f : g : h : i : j : k : l : m : n : Nil, ?w)
   1. rev(a : b : c : d : e : f : g : h : i : j : k : l : m : n : Nil,
          ?w)
val rules = [appNil,appCons,revNil,revCons];
```

```
by (REPEAT (resolve_tac rules 1));
  Level 1
  rev(a : b : c : d : e : f : g : h : i : j : k : l : m : n : Nil,
      n : m : l : k : j : i : h : g : f : e : d : c : b : a : Nil)
  No subgoals!
```

We may execute *rev* backwards. This, too, should reverse a list. What is the reverse of $[a, b, c]$?

```
Goal "rev(?x, a:b:c:Nil)";
  Level 0
  rev(?x, a : b : c : Nil)
   1. rev(?x, a : b : c : Nil)
by (REPEAT (resolve_tac rules 1));
  Level 1
  rev(?x1 : Nil, a : b : c : Nil)
   1. app(Nil, ?x1 : Nil, a : b : c : Nil)
```

The tactic has failed to find a solution! It reached a dead end at subgoal 1: there is no $?x_1$ such that [] appended with $[?x_1]$ equals $[a, b, c]$. Backtracking explores other outcomes.

```
back();
  Level 1
  rev(?x1 : a : Nil, a : b : c : Nil)
   1. app(Nil, ?x1 : Nil, b : c : Nil)
```

This too is a dead end, but the next outcome is successful.

```
back();
  Level 1
  rev(c : b : a : Nil, a : b : c : Nil)
  No subgoals!
```

`REPEAT` goes wrong because it is only a repetition tactical, not a search tactical. `REPEAT` stops when it cannot continue, regardless of which state is reached. The tactical `DEPTH_FIRST` searches for a satisfactory state, as specified by an ML predicate. Below, `has_fewer_prems` specifies that the proof state should have no subgoals.

```
val prolog_tac = DEPTH_FIRST (has_fewer_prems 1)
                             (resolve_tac rules 1);
```

Since Prolog uses depth-first search, this tactic is a (slow!) Prolog interpreter. We return to the start of the proof using `choplev`, and apply `prolog_tac`:

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

Let us try `prolog_tac` on one more example, containing four unknowns:

```
Goal "rev(a:?x:c:?y:Nil, d:?z:b:?u)";
  Level 0
  rev(a : ?x : c : ?y : Nil, d : ?z : b : ?u)
   1. rev(a : ?x : c : ?y : Nil, d : ?z : b : ?u)
by prolog_tac;
  Level 1
  rev(a : b : c : d : Nil, d : c : b : a : Nil)
  No subgoals!
```

Although Isabelle is much slower than a Prolog system, Isabelle tactics can exploit logic programming techniques.

References

[1] Antony Galton. *Logic for Information Technology.* Wiley, 1990.

[2] M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, 1993.

[3] Michael J. C. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation.* LNCS 78. Springer, 1979.

[4] Paul Hudak and Joseph H. Fasel. A gentle introduction to Haskell. *SIGPLAN Notices*, 27(5), May 1992.

[5] Paul Hudak, Simon Peyton Jones, and Philip Wadler. Report on the programming language Haskell: A non-strict, purely functional language. *SIGPLAN Notices*, 27(5), May 1992. Version 1.2.

[6] G. P. Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[7] Dale Miller. Unification under a mixed prefix. *Journal of Symbolic Computation*, 14(4):321–358, 1992.

[8] Tobias Nipkow and Christian Prehofer. Type reconstruction for type classes. *Journal of Functional Programming*, 5(2):201–224, 1995.

[9] Bengt Nordström, Kent Petersson, and Jan Smith. *Programming in Martin-Löf's Type Theory. An Introduction.* Oxford University Press, 1990.

[10] Lawrence C. Paulson. Natural deduction as higher-order resolution. *Journal of Logic Programming*, 3:237–258, 1986.

[11] Lawrence C. Paulson. *Logic and Computation: Interactive proof with Cambridge LCF.* Cambridge University Press, 1987.

[12] Lawrence C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5(3):363–397, 1989.

[13] Lawrence C. Paulson. Isabelle: The next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–386. Academic Press, 1990.

[14] Lawrence C. Paulson. *ML for the Working Programmer.* Cambridge University Press, 1991.

[15] Lawrence C. Paulson. Designing a theorem prover. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, pages 415–475. Oxford University Press, 1992.

[16] F. J. Pelletier. Seventy-five problems for testing automatic theorem provers. *Journal of Automated Reasoning*, 2:191–216, 1986. Errata, JAR 4 (1988), 235–236 and JAR 18 (1997), 135.

[17] Steve Reeves and Michael Clarke. *Logic for Computer Science*. Addison-Wesley, 1990.

[18] Patrick Suppes. *Axiomatic Set Theory*. Dover, 1972.

[19] Larry Wos. Automated reasoning and Bledsoe's dream for the field. In Robert S. Boyer, editor, *Automated Reasoning: Essays in Honor of Woody Bledsoe*, pages 297–342. Kluwer Academic Publishers, 1991.

Index