10
A Tactical Theorem Prover

ML was originally designed to serve as the programming language for a theorem prover, Edinburgh LCF. So it is fitting that a book on ML should conclude by describing a theorem prover, called Hal, inspired by LCF.[1] Hal constructs a proof by refinement steps, working backwards from a goal. At its simplest, this is proof checking: at each step, an inference rule is matched to a goal, reducing it to certain subgoals. If we are ever to prove anything significant, we shall require more automation. Hal provides ***tactics*** and ***tacticals***, which constitute a high-level language for expressing search procedures. A few primitive tactics, applied using a tactical for depth-first search, implement a general tactic that can prove many theorems automatically, such as

$$\neg(\exists x\,.\,\forall y\,.\,\phi(x,\,y) \leftrightarrow \neg\phi(y,\,y))$$
$$\exists xy\,.\,\phi(x,\,y) \rightarrow \forall xy\,.\,\phi(x,\,y)$$
$$\exists x\,.\,\forall yz\,.\,(\phi(y) \rightarrow \psi(z)) \rightarrow (\phi(x) \rightarrow \psi(x))$$

For raw power Hal cannot compete with specialized theorem provers. What Hal lacks in power it makes up in flexibility. A typical resolution theorem prover supports pure classical logic with equality, but without induction. Tactical theorem provers allow a mixture of automatic and interactive working, in virtually any logic.

Hal works in classical logic for familiarity's sake, but it can easily be extended to include induction, modal operators, set theory or whatever. Its tactics must be changed to reflect the new inference rules; the tacticals remain the same, ready to express search procedures for the new logic.

Chapter outline

The chapter contains the following sections:

A sequent calculus for first-order logic. The semantics of first-order logic is

[1] Hal is named after King Henry V, who was a master tactician.

sketched and the sequent calculus is described. Quantifier reasoning involves parameters and meta-variables.

Processing terms and formulæ in ML. Hal's representation of first-order logic borrows techniques from previous chapters. A major new technique is unification.

Tactics and the proof state. Hal implements the sequent calculus as a set of transformations upon an abstract type of proof states. Each inference rule is provided as a tactic.

Searching for proofs. A crude user interface allows the tactics to be demonstrated. Tacticals add control structures to tactics, and are used to code automatic tactics for first-order logic.

A sequent calculus for first-order logic

We begin with a quick overview of first-order logic. The syntax of first-order logic has been presented in Section 6.1. ***Propositional logic*** concerns formulæ built by the connectives $\wedge$, $\vee$, $\neg$, $\rightarrow$ and $\leftrightarrow$. First-order logic introduces the quantifiers $\forall$ and $\exists$, with variables and terms. A ***first-order language*** augments the logical symbols with certain constants a, b, $\ldots$, function symbols f, g, $\ldots$ and predicate symbols P, Q, $\ldots$. Let ϕ, ψ, χ, $\ldots$ stand for arbitrary formulæ.

The ***universe*** is a non-empty set containing the possible values of terms. Constants denote elements of the universe; function symbols denote functions over the universe; predicate symbols denote relations over the universe. A ***structure*** defines the semantics of a first-order language by specifying a universe and giving the interpretations of the constants, function symbols and predicate symbols. An ML structure is analogous to a logical structure.

The meaning of a formula depends on the values of its free variables. An ***assignment*** is a mapping from free variables to elements of the universe. Given a structure and an assignment, every formula is either true or false. The formula $\forall x \,.\, \phi$ is true if and only if ϕ is true for every possible value that could be assigned to x (leaving the other variables unchanged). The connectives are defined by truth tables; for instance, $\phi \wedge \psi$ is true if and only if ϕ is true and ψ is true.

A ***valid*** formula is one that is true in all structures and assignments. Since there are infinitely many structures, exhaustive testing can never show that a formula is valid. Instead, we can attempt to show that a formula is valid by formal proof using inference rules, each of which is justified by the semantics of the logic. Each rule accepts zero or more premises and yields a conclusion; a

sound rule must yield a valid conclusion provided its premises are valid. A set of inference rules for a logic is called a ***proof system*** or a ***formalization***.

Of the many proof systems for classical first-order logic, easiest to automate is the ***sequent calculus***. The tableau method, which is sometimes used to automate first-order logic, is a compact notation for the sequent calculus.

10.1 *The sequent calculus for propositional logic*

To keep matters simple, let us temporarily restrict attention to propositional logic. A ***sequent*** has the form

$$\phi_1, \ldots, \phi_m \vdash \psi_1, \ldots, \psi_n$$

where $\phi_1, \ldots, \phi_m$ and $\psi_1, \ldots, \psi_n$ are multisets of formulæ. As discussed above in Chapter 6, a multiset is a collection of elements whose order is insignificant. Traditionally a sequent contains lists of formulæ, and the logic includes rules for exchanging adjacent formulæ; multisets make such rules unnecessary.

Given a structure and an assignment, the sequent above is true if and only if some of the formulæ $\phi_1, \ldots, \phi_m$ are false or some of the formulæ $\psi_1, \ldots, \psi_n$ are true. In other words, the sequent has the same meaning as the formula

$$\phi_1 \wedge \cdots \wedge \phi_m \rightarrow \psi_1 \vee \cdots \vee \psi_n.$$

As a special case, $\vdash \psi$ has the same meaning as ψ. A sequent is not a formula, however; the $\vdash$ symbol (the 'turnstile') is not a logical connective.

For convenience in writing the rules, Γ and Δ will stand for multisets of formulæ. The comma will denote multiset union; thus Γ, Δ stands for the union of Γ and Δ. A formula appearing where a multiset is expected (like ϕ in $\Gamma \vdash \phi$) will stand for a singleton multiset. Thus Γ, ϕ is a multiset containing an occurrence of ϕ, where Γ denotes its other elements.

Validity and basic sequents. A ***valid*** sequent is one that is true under every structure and assignment. The theorems of our sequent calculus will be precisely the valid sequents.

A sequent is called ***basic*** if both sides share a common formula ϕ. This can be formalized as the axiom

$$\phi, \Gamma \vdash \Delta, \phi.$$

In the notation just described, ϕ, Γ and Δ, ϕ are multisets containing ϕ. Such sequents are clearly valid.

The other formulæ, those contained in Γ and Δ, play no part in the inference. The sequent calculus is sometimes formulated such that a basic sequent

must have the form $\phi \vdash \phi$. Then sequents of the form $\phi, \Gamma \vdash \Delta, \phi$ can be derived with the help of 'weakening' rules, which insert arbitrary formulæ into a sequent.

Sequent rules for the connectives. Sequent calculus rules come in pairs, to introduce each connective on the left or right of the $\vdash$ symbol. For example, the rule $\wedge$:left introduces a conjunction on the left, while $\wedge$:right introduces a conjunction on the right. Here is the latter rule in the usual notation, with its premises above the line and its conclusion below:

$$\frac{\Gamma \vdash \Delta, \phi \quad \Gamma \vdash \Delta, \psi}{\Gamma \vdash \Delta, \phi \wedge \psi} \wedge\text{:right}$$

To show that $\wedge$:right is a sound rule, let us assume that its premises are valid and demonstrate that its conclusion is valid. Suppose that, under some structure and assignment, every formula in Γ is true; we must demonstrate that some formula in $\Delta, \phi \wedge \psi$ is true. If no formula in Δ is true, then both ϕ and ψ are true by the premises. Therefore $\phi \wedge \psi$ is true.

Now let us justify the rule $\wedge$:left.

$$\frac{\phi, \psi, \Gamma \vdash \Delta}{\phi \wedge \psi, \Gamma \vdash \Delta} \wedge\text{:left}$$

To show that this rule is sound, we proceed as above. Suppose that every formula in $\Gamma, \phi \wedge \psi$ is true. Then both ϕ and ψ are true. Assuming that the premise is valid, some formula of Δ must be true, and this establishes the conclusion.

Figure 10.1 presents the rules for the propositional connectives $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$ and $\neg$. All the rules are justified similarly.

Exercise 10.1 Which formula is equivalent to $\phi_1, \ldots, \phi_m \vdash$, a sequent whose right side is empty?

Exercise 10.2 Justify the rules $\vee$:left and $\vee$:right.

Exercise 10.3 Justify the rules $\leftrightarrow$:left and $\leftrightarrow$:right.

10.2 *Proving theorems in the sequent calculus*

Inference rules are often viewed in a forward direction, from premises to conclusion. Thus, $\wedge$:right accepts premises $\Gamma \vdash \Delta, \phi$ and $\Gamma \vdash \Delta, \psi$, yielding the conclusion $\Gamma \vdash \Delta, \phi \wedge \psi$. Applying another rule to this sequent yields another conclusion, and so forth. A formal proof is a tree constructed by applying

Figure 10.1 *Sequent rules for the propositional connectives*

:left	:right
$\dfrac{\phi, \psi, \Gamma \vdash \Delta}{\phi \wedge \psi, \Gamma \vdash \Delta}$	$\dfrac{\Gamma \vdash \Delta, \phi \quad \Gamma \vdash \Delta, \psi}{\Gamma \vdash \Delta, \phi \wedge \psi}$
$\dfrac{\phi, \Gamma \vdash \Delta \quad \psi, \Gamma \vdash \Delta}{\phi \vee \psi, \Gamma \vdash \Delta}$	$\dfrac{\Gamma \vdash \Delta, \phi, \psi}{\Gamma \vdash \Delta, \phi \vee \psi}$
$\dfrac{\Gamma \vdash \Delta, \phi \quad \psi, \Gamma \vdash \Delta}{\phi \rightarrow \psi, \Gamma \vdash \Delta}$	$\dfrac{\phi, \Gamma \vdash \Delta, \psi}{\Gamma \vdash \Delta, \phi \rightarrow \psi}$
$\dfrac{\phi, \psi, \Gamma \vdash \Delta \quad \Gamma \vdash \Delta, \phi, \psi}{\phi \leftrightarrow \psi, \Gamma \vdash \Delta}$	$\dfrac{\phi, \Gamma \vdash \Delta, \psi \quad \psi, \Gamma \vdash \Delta, \phi}{\Gamma \vdash \Delta, \phi \leftrightarrow \psi}$
$\dfrac{\Gamma \vdash \Delta, \phi}{\neg\phi, \Gamma \vdash \Delta}$	$\dfrac{\phi, \Gamma \vdash \Delta}{\Gamma \vdash \Delta, \neg\phi}$

inference rules. Here is a proof of the sequent $\phi \wedge \psi \vdash \psi \wedge \phi$:

$$\dfrac{\dfrac{\phi, \psi \vdash \psi \qquad \phi, \psi \vdash \phi}{\phi, \psi \vdash \psi \wedge \phi}\ \wedge\text{:right}}{\phi \wedge \psi \vdash \psi \wedge \phi}\ \wedge\text{:left} \qquad (*)$$

Viewed in the forward direction, two basic sequents are combined by $\wedge$:right and the result transformed by $\wedge$:left. However, the forward reading does not help us find a proof of a given sequent.

For the purpose of finding proofs, rules should be viewed in the backward direction, from a goal to subgoals. Thus, $\wedge$:right accepts the goal $\Gamma \vdash \Delta, \phi \wedge \psi$ and returns the subgoals $\Gamma \vdash \Delta, \phi$ and $\Gamma \vdash \Delta, \psi$. If these subgoals can be proved as theorems, then so can the goal. The subgoals are refined by further rule applications until all the remaining subgoals are basic sequents, which are immediately valid. The proof tree is constructed from the root upwards; the process is called ***refinement*** or ***backward proof***.

Viewed in the backward direction, the proof $(*)$ begins with the sequent to be proved, namely $\phi \wedge \psi \vdash \psi \wedge \phi$. This goal is refined by $\wedge$:left to $\phi, \psi \vdash \psi \wedge \phi$; this subgoal is refined by $\wedge$:right to $\phi, \psi \vdash \psi$ and $\phi, \psi \vdash \phi$. These two subgoals are basic sequents, so the proof is finished.

Under the backward reading, each rule attacks a formula in the goal. Applying $\wedge$:left breaks down a conjunction on the left side, while $\wedge$:right breaks down a conjunction on the right. If all the resulting subgoals are basic sequents, then the initial goal has been proved. For propositional logic, this procedure must terminate.

A sequent may have several different proofs, depending on which formulæ are broken down first. The proof $(*)$ first breaks down the conjunction on the left in $\phi \wedge \psi \vdash \psi \wedge \phi$. For a different proof, begin by breaking down the conjunction on the right:

$$\dfrac{\dfrac{\phi, \psi \vdash \psi}{\phi \wedge \psi \vdash \psi}\,\wedge\text{:left} \qquad \dfrac{\phi, \psi \vdash \phi}{\phi \wedge \psi \vdash \phi}\,\wedge\text{:left}}{\phi \wedge \psi \vdash \psi \wedge \phi}\,\wedge\text{:right}$$

This is larger than the proof $(*)$ in that $\wedge$:left is applied twice. Applying $\wedge$:right to the initial goal produced two subgoals, each with a conjunction on the left. Shorter proofs usually result if the rule that produces the fewest subgoals is chosen at each step.

To summarize, we have the following proof procedure:

- Take the sequent to be proved as the initial goal. The root of the proof tree, and its only leaf, is this goal.
- Select some subgoal that is a leaf of the proof tree and apply a rule to it, turning the leaf into a branch node with one or more leaves.
- Stop whenever all the leaves are basic sequents (success), or when no rules can be applied to a leaf (failure).

This procedure is surprisingly effective, though its search is undirected. Both $\vee$:left and $\wedge$:right may be applied to the subgoal $p \vee q, r \vdash r \wedge r$. The former rule performs case analysis on the irrelevant formula $p \vee q$; the latter rule yields two basic subgoals, succeeding immediately.

Exercise 10.4 Construct proofs of the sequents $\phi \vee \psi \vdash \psi \vee \phi$ and $\phi_1 \wedge (\phi_2 \wedge \phi_3) \vdash (\phi_1 \wedge \phi_2) \wedge \phi_3$.

Exercise 10.5 Construct a proof of the sequent

$$\vdash (\phi_1 \wedge \phi_2) \vee \psi \leftrightarrow (\phi_1 \vee \psi) \wedge (\phi_2 \vee \psi).$$

Exercise 10.6 Show that any sequent containing both ϕ and $\neg\phi$ to the left of the $\vdash$ symbol is provable.

10.3 *Sequent rules for the quantifiers*

Propositional logic is decidable; our proof procedure can determine, in finite time, whether any formula is a theorem. With quantifiers, no such decision procedure exists. Quantifiers, moreover, introduce many syntactic complications.

Each quantifier binds a variable; thus x and y occur bound and z occurs free in $\forall x \,.\, \exists y \,.\, R(x, y, z)$. Renaming the bound variables does not affect the meaning of a formula; the previous example is equivalent to $\forall y \,.\, \exists w \,.\, R(y, w, z)$. Some of the inference rules involve substitution, and $\phi[t/x]$ will stand for the result of substituting t for every free occurrence of x in ϕ. Less formally, $\phi(x)$ stands for a formula involving x and $\phi(t)$ stands for the result of substituting t for free occurrences of x. The name-free representation of bound variables (Section 9.6) works as well for quantifier syntax as it does for the λ-calculus.

The universal quantifier has these two sequent rules:

$$\frac{\phi[t/x],\ \forall x \,.\, \phi,\ \Gamma \vdash \Delta}{\forall x \,.\, \phi,\ \Gamma \vdash \Delta}\ \forall\text{:left} \qquad \frac{\Gamma \vdash \Delta,\ \phi}{\Gamma \vdash \Delta,\ \forall x \,.\, \phi}\ \forall\text{:right}$$

proviso: x must not occur free in the conclusion

The rule $\forall$:left is easy to justify; if $\forall x \,.\, \phi$ is true then so is $\phi[t/x]$, where t is any term.

To justify $\forall$:right, which is the more complicated rule, let us assume that its premise is valid and demonstrate that its conclusion is valid. Given some structure and assignment, suppose that every formula in Γ is true and that no formula in Δ is true; then we must show that $\forall x \,.\, \phi$ is true. It suffices to show that ϕ is true for every possible assignment to x that leaves the other variables unchanged. By the proviso of $\forall$:right, changing the value of x does not affect the truth of any formula of Γ or Δ; since the premise is valid, ϕ must be true.

Ignoring the proviso can yield unsound inferences:

$$\frac{P(x) \vdash P(x)}{P(x) \vdash \forall x \,.\, P(x)}\ \forall\text{:right}\ \ ???$$

The conclusion is false if $P(x)$ stands for the predicate $x = 0$ over the integers and x is assigned the value 0.

The existential quantifier has these two sequent rules:

$$\frac{\phi,\ \Gamma \vdash \Delta}{\exists x \,.\, \phi,\ \Gamma \vdash \Delta}\ \exists\text{:left} \qquad \frac{\Gamma \vdash \Delta,\ \exists x \,.\, \phi,\ \phi[t/x]}{\Gamma \vdash \Delta,\ \exists x \,.\, \phi}\ \exists\text{:right}$$

proviso: x must not occur free in the conclusion

They are dual to the rules for the universal quantifier and can be justified similarly. Note that $\exists x \,.\, \phi$ is equivalent to $\neg\forall x \,.\, \neg\phi$.

The rules $\forall$:left and $\exists$:right have one feature that is not present in any of the other rules: in backward proof, they do not remove any formulæ from the goal. They expand a quantified formula, substituting a term into its body; and they retain the formula to allow repeated expansion. It is impossible to determine in advance how many expansions of a quantified formula are required for a proof. Because of this, our proof procedure can fail to terminate; first-order logic is undecidable.

Exercise 10.7 If the premise of $\forall$:right is ignored, can a proof involving this rule reach an inconsistent conclusion? (This means a sequent $\vdash \phi$ such that $\neg\phi$ is a valid formula.)

10.4 *Theorem proving with quantifiers*

Our backward proof procedure is reasonably effective with quantifiers, at least for tackling simple problems that do not require a more discriminating search. Let us begin with an easy proof involving universal quantification:

$$\dfrac{\dfrac{\dfrac{\phi(x),\ \forall x \,.\, \phi(x) \vdash \phi(x),\ \psi(x)}{\forall x \,.\, \phi(x) \vdash \phi(x),\ \psi(x)}\ \forall\text{:left}}{\forall x \,.\, \phi(x) \vdash \phi(x) \vee \psi(x)}\ \vee\text{:right}}{\forall x \,.\, \phi(x) \vdash \forall x \,.\, \phi(x) \vee \psi(x)}\ \forall\text{:right}$$

The proviso of $\forall$:right holds; x is not free in the conclusion. In a backward proof, this conclusion is the initial goal.

If we first applied $\forall$:left, inserting the formula $\phi(x)$, then x would be free in the resulting subgoal. Then $\forall$:right could not be applied without renaming the quantified variable:

$$\dfrac{\dfrac{\dfrac{\phi(x),\ \forall x \,.\, \phi(x) \vdash \phi(y),\ \psi(y)}{\phi(x),\ \forall x \,.\, \phi(x) \vdash \phi(y) \vee \psi(y)}\ \vee\text{:right}}{\phi(x),\ \forall x \,.\, \phi(x) \vdash \forall x \,.\, \phi(x) \vee \psi(x)}\ \forall\text{:right}}{\forall x \,.\, \phi(x) \vdash \forall x \,.\, \phi(x) \vee \psi(x)}\ \forall\text{:left}$$

The topmost sequent is not basic; to finish the proof we must again apply $\forall$:left. The first application of this rule has accomplished nothing. We have a general heuristic: never apply $\forall$:left or $\exists$:right to a goal if a different rule can usefully be applied.

The following proof illustrates some of the difficulties of using quantifiers.[2]

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\phi(x),\ \phi(z) \vdash \exists z\,.\,\phi(z) \to \forall x\,.\,\phi(x),\ \phi(x),\ \forall x\,.\,\phi(x)}{\phi(z) \vdash \exists z\,.\,\phi(z) \to \forall x\,.\,\phi(x),\ \phi(x),\ \phi(x) \to \forall x\,.\,\phi(x)}\to\text{:right}}{\phi(z) \vdash \exists z\,.\,\phi(z) \to \forall x\,.\,\phi(x),\ \phi(x)}\exists\text{:right}}{\phi(z) \vdash \exists z\,.\,\phi(z) \to \forall x\,.\,\phi(x),\ \forall x\,.\,\phi(x)}\forall\text{:right}}{\vdash \exists z\,.\,\phi(z) \to \forall x\,.\,\phi(x),\ \phi(z) \to \forall x\,.\,\phi(x)}\to\text{:right}}{\vdash \exists z\,.\,\phi(z) \to \forall x\,.\,\phi(x)}\exists\text{:right}$$

Working upwards from the goal, $\exists$:right is applied, introducing z as a free variable. Although the existential formula remains in the subgoal, it remains dormant until we again reach a goal where no other rule is applicable. The next inference, $\to$:right, moves $\phi(z)$ to the left. Since x is not free in the subgoal, $\forall$:right can be applied, replacing $\forall x\,.\,\phi(x)$ by $\phi(x)$. In the resulting subgoal, $\exists$:right is again applied (there is no alternative), substituting x for z. The final subgoal after $\to$:right is a basic sequent containing $\phi(x)$ on both sides.

Observe that $\exists z\,.\,\phi(z) \to \forall x\,.\,\phi(x)$ is expanded twice by $\exists$:right. The sequent cannot be proved otherwise. Sequents requiring n expansions of a quantifier, for any given n, are not hard to devise.

Unification. When reasoning about quantifiers, we have a difficulty: how do we choose the term t in the rules $\exists$:right and $\forall$:left? This amounts to predicting which term will ultimately generate basic subgoals and a successful proof. In the proof above, choosing z in the first $\exists$:right was arbitrary; any term would have worked. Choosing x in the second $\exists$:right was crucial — but perhaps not obvious.

We can postpone choosing the term in such rules. Introduce ***meta-variables*** $?a, ?b, \ldots$ as placeholders for terms. When a goal can be solved by substituting appropriate terms for its meta-variables, perform this substitution — throughout the proof. For instance, the subgoal $P(?a), \Gamma \vdash \Delta, P(f(?b))$ becomes basic if we replace $?a$ by $f(?b)$; observe that $?a$ has still not been fully determined, only its outer form $f(\cdots)$. We solve for unknowns incrementally. ***Unification***, the process of determining the appropriate substitutions, is the key to automated reasoning about quantifiers.

The rule $\forall$:left now takes the following form, where $?a$ stands for any meta-

[2] To see that $\exists z\,.\,\phi(z) \to \forall x\,.\,\phi(x)$ is a theorem, first note that in fully parenthesized form it is $\exists z\,.\,[\phi(z) \to (\forall x\,.\,\phi(x))]$. Pushing the existential quantifier inside the implication changes it to a universal quantifier. The formula is thus equivalent to $(\forall z\,.\,\phi(z)) \to (\forall x\,.\,\phi(x))$, which is trivially true.

variable:

$$\frac{\phi[?a/x],\ \forall x\,.\,\phi,\ \Gamma \vdash \Delta}{\forall x\,.\,\phi,\ \Gamma \vdash \Delta}\ \forall\text{:left}$$

Enforcing provisos. Meta-variables cause difficulties of their own. Recall that ∀:right and ∃:left have the proviso 'x not free in conclusion.' What shall we do when the conclusion contains meta-variables, which could be replaced by any terms? Our approach is to label each free variable with a list of forbidden meta-variables. The free variable $b_{?a_1,\ldots,?a_k}$ must never be contained in a term substituted for the meta-variables $?a_1, \ldots, ?a_k$. The unification algorithm can enforce this.

Let us simplify the terminology. Labelled free variables will be called ***parameters***. Meta-variables will be called ***variables***.

Using parameters, the rule ∀:right becomes

$$\frac{\Gamma \vdash \Delta,\ \phi[b_{?a_1,\ldots,?a_k}/x]}{\Gamma \vdash \Delta,\ \forall x\,.\,\phi}\ \forall\text{:right}$$

proviso: b must not occur in the conclusion and $?a_1, \ldots, ?a_k$ must be all the variables in the conclusion.

The first part of the proviso ensures that the parameter b is not already in use, while the second part ensures that b is not slipped in later by a substitution. The treatment of ∃:left is the same.

Parameters ensure correct quantifier reasoning. For example, $\forall x\,.\,\phi(x, x)$ does not, in general, imply $\exists y\,.\,\forall x\,.\,\phi(x, y)$. Consider an attempted proof of the corresponding sequent:

$$\dfrac{\dfrac{\dfrac{\phi(?c, ?c),\ \forall x.\phi(x, x) \vdash \exists y.\forall x.\phi(x, y),\ \phi(b_{?a}, ?a)}{\forall x.\phi(x, x) \vdash \exists y.\forall x.\phi(x, y),\ \phi(b_{?a}, ?a)}\ \forall\text{:left}}{\forall x.\phi(x, x) \vdash \exists y.\forall x.\phi(x, y),\ \forall x.\phi(x, ?a)}\ \forall\text{:right}}{\forall x.\phi(x, x) \vdash \exists y.\forall x.\phi(x, y)}\ \exists\text{:right}$$

The topmost sequent cannot be made basic. To make $\phi(?c, ?c)$ and $\phi(b_{?a}, ?a)$ identical, a substitution would have to replace both $?c$ and $?a$ by $b_{?a}$. However, the parameter $b_{?a}$ is forbidden from occurring in a term substituted for $?a$. The attempted proof may continue to grow upwards through applications of ∀:right and ∃:left, but no basic sequent will ever be generated.

For a contrasting example, let us prove that $\forall x.\phi(x, x)$ implies $\forall x.\exists y.\phi(x, y)$:

$$\dfrac{\dfrac{\dfrac{\phi(?c, ?c),\ \forall x.\phi(x, x) \vdash \exists y.\phi(a, y),\ \phi(a, ?b)}{\forall x.\phi(x, x) \vdash \exists y.\phi(a, y),\ \phi(a, ?b)}\ \forall\text{:left}}{\forall x.\phi(x, x) \vdash \exists y.\phi(a, y)}\ \exists\text{:right}}{\forall x.\phi(x, x) \vdash \forall x.\exists y.\phi(x, y)}\ \forall\text{:right}$$

Replacing $?b$ and $?c$ by a transforms both $\phi(?c, ?c)$ and $\phi(a, ?b)$ into $\phi(a, a)$, completing the proof. The parameter a is not labelled with any variables because there are none in the goal supplied to ∀:right.

Further reading. A number of textbooks present logic from the viewpoint of computer science. They emphasize proof procedures and unification, avoiding the more traditional concerns of mathematical logic, such as model theory. See Galton (1990) or Reeves and Clarke (1990) for a gentle introduction to logic. Gallier (1986) gives a more technical treatment centred around the sequent calculus.

Exercise 10.8 Reconstruct the first three quantifier proofs above, this time using (meta) variables and parameters.

Exercise 10.9 Falsify the sequent $\forall x\,.\,P(x, x) \vdash \exists y\,.\,\forall x\,.\,P(x, y)$ by letting P denote a suitable relation in a structure.

Exercise 10.10 If the attempted proof of $\forall x\,.\,\phi(x, x) \vdash \exists y\,.\,\forall x\,.\,\phi(x, y)$ is continued, will parameters allow it to succeed?

Exercise 10.11 Demonstrate that $\vdash \exists z\,.\,\phi(z) \to \forall x\,.\,\phi(x)$ has no proof that applies ∃:right only once.

Exercise 10.12 For each of the following, construct a proof or demonstrate that no proof exists (a and b are constants):

$$\vdash \exists z\,.\,\phi(z) \to \phi(a) \wedge \phi(b)$$
$$\forall x\,.\,\exists y\,.\,\phi(x, y) \vdash \exists y\,.\,\forall x\,.\,\phi(x, y)$$
$$\exists y\,.\,\forall x\,.\,\phi(x, y) \vdash \forall x\,.\,\exists y\,.\,\phi(x, y)$$

Processing terms and formulæ in ML

Let us code an infrastructure for theorem proving. Terms and formulæ must be represented; abstraction, substitution, parsing and pretty printing must be implemented. Thanks to the methods we have accumulated in recent chapters, none of this programming is especially difficult.

10.5 *Representing terms and formulæ*

The techniques we have developed for the λ-calculus (in Section 9.7) work for first-order logic. In some respects, first-order logic is simpler. An inference can affect only the outermost variable binding; there is nothing corresponding to a reduction within a λ-term.

The signature. Signature *FOL* defines the representation of first-order terms and formulæ:

```
signature FOL =
  sig
  datatype term  = Var     of string
                 | Param   of string * string list
                 | Bound   of int
                 | Fun     of string * term list
  datatype form  = Pred    of string * term list
                 | Conn    of string * form list
                 | Quant   of string * string * form
  type goal      = form list * form list
  val precOf     : string -> int
  val abstract   : int -> term -> form -> form
  val subst      : int -> term -> form -> form
  val termVars   : term * string list -> string list
  val goalVars   : goal *string list -> string list
  val termParams : term * (string * string list) list
                   -> (string * string list) list
  val goalParams : goal *(string * string list) list
                   -> (string * string list) list
  end;
```

Type $term$ realizes the methods described in the previous section. A variable (constructor Var) has a name. A $Bound$ variable has an index. A Fun application has a function's name and argument list; a function taking no arguments is simply a constant. A parameter ($Param$) has a name and a list of forbidden variables.

Type $form$ is elementary. An atomic formula ($Pred$) has a predicate's name and argument list. A connective application ($Conn$) has a connective and a list of formulæ, typically `"~"`, `"&"`, `"|"`, `"-->"`, or `"<->"` paired with one or two formulæ. A $Quant$ formula has a quantifier (either `"ALL"` or `"EX"`), a bound variable name and a formula for the body.

Type $goal$ abbreviates the type of pairs of formula lists. Some older ML compilers do not allow type abbreviations in signatures. We could specify $goal$ simply as a type: its declaration inside the structure would be visible outside.

The function $precOf$ defines the precedences of the connectives, as required for parsing and printing.

Functions $abstract$ and $subst$ resemble their namesakes of the previous chapter, but operate on formulæ. Calling $abstract\ i\ t\ p$ replaces each occurrence of t in p by the index i (which is increased within quantifications); typically $i = 0$ and t is an atomic term. Calling $subst\ i\ t\ p$ replaces the index i (increased within quantifications) by t in the formula p.

The function $termVars$ collects the list of variables in a term (without repetitions); $termVars(t, bs)$ inserts all the variables of t into the list bs. The argument bs may appear to be a needless complication, but it eliminates costly list appends while allowing $termVars$ to be extended to formulæ and goals. This will become clear when we examine the function definitions.

The function $goalVars$, which also takes two arguments, collects the list of variables in a goal. A goal in Hal is a sequent. Although sequents are represented in ML using formula lists, not multisets, we shall be able to implement the style of proof discussed above.

The functions $termParams$ and $goalParams$ collect the list of parameters in a term or goal, respectively. Each parameter consists of its name paired with a list of variable names.

The structure. Structure *Fol* (Figure 10.2) implements signature *FOL*. The `datatype` declarations of $term$ and $form$ are omitted to save space; they are identical to those in the signature. The structure declares several functions not specified in the signature.

Calling $replace\ (u1, u2)\ t$ replaces the term $u1$ by $u2$ throughout the term t. This function is called by $abstract$ and $subst$.

Functionals $accumForm$ and $accumGoal$ demonstrate higher-order programming. Suppose that f has type $term \times \tau \to \tau$, for some type τ, where $f(t, x)$ accumulates some information about t in x. (For instance, f could be $termVars$, which accumulates the list of free variables in a term.) Then $foldr\ f$ extends f to lists of terms. The function $accumForm\ f$ has type $form \times \tau \to \tau$, extending f to operate on formulæ. It lets $foldr\ f$ handle the arguments of a predicate $P(t_1, \ldots, t_n)$; it recursively lets $foldr\ (accumForm\ f)$ handle the formula lists of connectives. The functional $accumGoal$ calls $foldr$ twice, extending a function of type $form \times \tau \to \tau$ to one of type $(form\ list \times form\ list) \times \tau \to \tau$. It extends a function involving formulæ to one involving goals.

Functionals $accumForm$ and $accumGoal$ provide a uniform means of traversing formulæ and goals. They define the functions $goalVars$ and $goalParams$ and could have many similar applications. Moreover, they are efficient: they create no lists or other data structures.

The functions $termVars$ and $termParams$ are defined by recursion, scanning a term to accumulate its variables or parameters. They use $foldr$ to traverse argument lists. The function $insert$ (omitted to save space) builds an ordered list of strings without repetitions. Note that $termVars$ does not regard the parameter $b_{?a_1,\ldots,?a_k}$ as containing $?a_1, \ldots, ?a_k$; these forbidden variables are not logically part of the term and perhaps ought to be stored in a separate table.

Figure 10.2 *First-order logic: representing terms and formulæ*

```
structure Fol : FOL =
  struct
  datatype term = ...;  datatype form = ...
  type goal = form list * form list;

  fun replace (u1,u2) t =
      if t=u1 then u2 else
          case t of Fun(a,ts) => Fun(a, map (replace(u1,u2)) ts)
                  | _         => t;

  fun abstract i t (Pred(a,ts))       = Pred(a, map (replace(t, Bound i)) ts)
    | abstract i t (Conn(b,ps))       = Conn(b, map (abstract i t) ps)
    | abstract i t (Quant(qnt,b,p)) = Quant(qnt, b, abstract (i+1) t p);

  fun subst i t (Pred(a,ts))          = Pred(a, map (replace(Bound i, t)) ts)
    | subst i t (Conn(b,ps))          = Conn(b, map (subst i t) ps)
    | subst i t (Quant(qnt,b,p))    = Quant(qnt, b, subst (i+1) t p);

  fun precOf "~"    = 4
    | precOf "&"    = 3
    | precOf "|"    = 2
    | precOf "<->" = 1
    | precOf "-->" = 1
    | precOf _      = ~1     (*means not an infix*);

  fun accumForm f (Pred(_,ts), z)     = foldr f z ts
    | accumForm f (Conn(_,ps), z)     = foldr (accumForm f) z ps
    | accumForm f (Quant(_,_,p), z) = accumForm f (p,z);

  fun accumGoal f ((ps,qs), z) = foldr f (foldr f z qs) ps;

  fun insert ...

  fun termVars (Var a, bs)      = insert(a,bs)
    | termVars (Fun(_,ts), bs) = foldr termVars bs ts
    | termVars (_, bs)          = bs;

  val goalVars = accumGoal (accumForm termVars);

  fun termParams (Param(a,bs), pairs) = (a,bs) :: pairs
    | termParams (Fun(_,ts), pairs)     = foldr termParams pairs ts
    | termParams (_, pairs)             = pairs;

  val goalParams = accumGoal (accumForm termParams);
  end;
```

Exercise 10.13 Sketch how *FOL* and *Fol* can be modified to adopt a new representation of terms. Bound variables are identified by name, but are syntactically distinct from parameters and meta-variables. Would this representation work for the λ-calculus?

Exercise 10.14 Change the declaration of type *form*, replacing *Conn* by separate constructors for each connective, say *Neg*, *Conj*, *Disj*, *Imp*, *Iff*. Modify *FOL* and *Fol* appropriately.

Exercise 10.15 The function *accumGoal* is actually more polymorphic than was suggested above. What is its most general type?

10.6 *Parsing and displaying formulæ*

Our parser and pretty printer (from Chapters 9 and 8, respectively) can implement the syntax of first-order logic. We employ the following grammar for terms (*Term*), optional argument lists (*TermPack*), and non-empty term lists (*TermList*):

$$\begin{aligned} \mathit{TermList} &= \mathit{Term}\,\{\texttt{,}\ \ \mathit{Term}\}* \\ \mathit{TermPack} &= \texttt{(}\ \ \mathit{TermList}\ \ \texttt{)} \\ &\mid \mathit{Empty} \\ \mathit{Term} &= \mathit{Id}\ \ \mathit{TermPack} \\ &\mid \texttt{?}\ \ \mathit{Id} \end{aligned}$$

Formulæ (*Form*) are defined in mutual recursion with primaries, which consist of atomic formulæ and their negations:

$$\begin{aligned} \mathit{Form} &= \texttt{ALL}\ \ \mathit{Id}\ \ \texttt{.}\ \ \mathit{Form} \\ &\mid \texttt{EX}\ \ \ \mathit{Id}\ \ \texttt{.}\ \ \mathit{Form} \\ &\mid \mathit{Form}\ \ \mathit{Conn}\ \ \mathit{Form} \\ &\mid \mathit{Primary} \\ \mathit{Primary} &= \texttt{\~{}}\ \ \mathit{Primary} \\ &\mid \texttt{(}\ \ \mathit{Form}\ \ \texttt{)} \\ &\mid \mathit{Id}\ \ \mathit{TermPack} \end{aligned}$$

The quantifiers are rendered into ASCII characters as `ALL` and `EX`; the following table gives the treatment of the connectives:

Usual symbol:	$\neg$	$\wedge$	$\vee$	$\rightarrow$	$\leftrightarrow$
ASCII *version*:	`~`	`&`	`|`	`-->`	`<->`

The formula $\exists z\,.\,\phi(z) \rightarrow \forall x\,.\,\phi(x)$ might be rendered as

```
EX z. P(z) --> (ALL x. P(x))
```

since ASCII lacks Greek letters. Hal requires a quantified formula to be enclosed in parentheses if it is the operand of a connective.

Parsing. The signature for parsing is minimal. It simply specifies the function *read*, for converting strings to formulæ:

```
signature PARSE_FOL =
  sig
  val read: string -> Fol.form
  end;
```

Before we can implement this signature, we must build structures for the lexical analysis and parsing of first-order logic. Structure *FolKey* defines the lexical syntax. Let us apply the functors described in Chapter 9:

```
structure FolKey     =
    struct val alphas  = ["ALL","EX"]
           and symbols = ["(", ")", ".", ",", "?", "~",
                          "&", "|", "<->", "-->", "|-"]
    end;
structure FolLex     = Lexical (FolKey);
structure FolParsing = Parsing (FolLex);
```

Figure 10.3 presents the corresponding structure. It is fairly simple, but a few points are worth noting.

Functions *list* and *pack* express the grammar phrases $TermList$ and $TermPack$. They are general enough to define 'lists' and 'packs' of arbitrary phrases.

The parser cannot distinguish constants from parameters or check that functions have the right number of arguments: it keeps no information about the functions and predicates of the first-order language. It regards any identifier as a constant, representing x by Fun("x", []). When parsing the quantification $\forall x\,.\,\phi(x)$, it abstracts the body $\phi(x)$ over its occurrences of the 'constant' x.

As discussed in the previous chapter, our parser cannot accept left-recursive grammar rules such as

$$Form = Form\ Conn\ Form.$$

Figure 10.3 *Parsing for first-order logic*

```
structure ParseFol : PARSE_FOL =
  struct
  local

    open FolParsing
    fun list ph  =     ph -- repeat ("," $-- ph)    >> (op::);

    fun pack ph =     "(" $-- list ph -- $")"      >> #1
                   || empty;

    fun makeQuant ((qnt,b),p) =
        Fol.Quant(qnt, b, Fol.abstract 0 (Fol.Fun(b,[])) p);

    fun makeConn a p q = Fol.Conn(a, [p,q]);
    fun makeNeg p      = Fol.Conn("~", [p]);

    fun term toks =
      (   id    -- pack term                    >> Fol.Fun
       || "?" $-- id                            >> Fol.Var    ) toks;

    fun form toks =
      (   $"ALL" -- id -- "." $-- form      >> makeQuant
       || $"EX"  -- id -- "." $-- form      >> makeQuant
       || infixes (primary, Fol.precOf, makeConn)  ) toks
    and primary toks =
      (   "~" $-- primary                   >> makeNeg
       || "(" $-- form -- $")"              >> #1
       || id -- pack term                   >> Fol.Pred    )   toks;
    in
        val read = reader form
    end
  end;
```

Instead, it relies on the precedences of the connectives. It invokes the parsing function *infixes* with three arguments:

- *primary* parses the operands of connectives.
- *precOf* defines the precedences of the connectives.
- *makeConn* applies a connective to two formulæ.

Most of the structure body is made private by a `local` declaration. At the bottom it defines the only visible identifiers, *form* and *read*. We could easily declare a reading function for terms if necessary.

Displaying. Signature *DISPLAY_FOL* specifies the pretty printing operators for formulæ and goals (which are sequents):

```
signature DISPLAY_FOL =
  sig
  val form: Fol.form -> unit
  val goal: int -> Fol.goal -> unit
  end;
```

The integer argument of function *goal* is displayed before the goal itself. It represents the subgoal number; a proof state typically has several subgoals. The sessions in Section 10.14 illustrate the output.

Structure *DisplayFol* implements this signature; see Figure 10.4. Our pretty printer must be supplied with symbolic expressions that describe the formatting. Function *enclose* wraps an expression in parentheses, while *list* inserts commas between the elements of a list of expressions. Together, they format argument lists as $(t_1, \ldots, t_n)$.

A parameter's name is printed, but not its list of forbidden variables. Another part of the program will display that information as a table.

The precedences of the connectives govern the inclusion of parentheses. Calling *formp k q* formats the formula q — enclosing it in parentheses, if necessary, to protect it from an adjacent connective of precedence k. In producing the string `q & (p | r)`, it encloses `p | r` in parentheses because the adjacent connective (`&`) has precedence 3 while `|` has precedence 2.

Exercise 10.16 Explain the workings of each of the functions supplied to `>>` in *ParseFol*.

Exercise 10.17 Alter the parser to admit `q --> ALL x. p` as correct syntax for $q \to (\forall x\,.\,p)$, for example. It should no longer demand parentheses around quantified formulæ.

Figure 10.4 *Pretty printing for first-order logic*

```
structure DisplayFol : DISPLAY_FOL =
  struct
  fun enclose sexp = Pretty.blo(1, [Pretty.str"(", sexp, Pretty.str")"]);

  fun commas []           = []
    | commas (sexp::sexps) = Pretty.str"," :: Pretty.brk 1 ::
                             sexp :: commas sexps;

  fun list (sexp::sexps)      = Pretty.blo(0, sexp :: commas sexps);

  fun term (Fol.Param(a,_)) = Pretty.str a
    | term (Fol.Var a)       = Pretty.str ("?"^a)
    | term (Fol.Bound i)     = Pretty.str "??UNMATCHED INDEX??"
    | term (Fol.Fun (a,ts))  = Pretty.blo(0, [Pretty.str a, args ts])
  and args []                = Pretty.str""
    | args ts                = enclose (list (map term ts));

  fun formp k (Fol.Pred (a,ts))       = Pretty.blo(0, [Pretty.str a, args ts])
    | formp k (Fol.Conn("~", [p])) =
          Pretty.blo(0, [Pretty.str "~", formp (Fol.precOf "~") p])
    | formp k (Fol.Conn(C, [p,q])) =
          let val pf   = formp (Int.max(Fol.precOf C, k))
              val sexp = Pretty.blo(0, [pf p, Pretty.str(" "^C),
                                        Pretty.brk 1, pf q])
          in  if (Fol.precOf C <= k) then (enclose sexp) else sexp
          end
    | formp k (Fol.Quant(qnt,b,p))  =
          let val q    = Fol.subst 0 (Fol.Fun(b,[])) p
              val sexp = Pretty.blo(2, [Pretty.str(qnt ^ " " ^ b ^ "."),
                                        Pretty.brk 1, formp 0 q])
          in  if  k>0  then  (enclose sexp)  else sexp
          end
    | formp k _                     = Pretty.str"??UNKNOWN FORMULA??";

  fun formList [] = Pretty.str"empty"
    | formList ps = list (map (formp 0) ps);

  fun form p = Pretty.pr (TextIO.stdOut, formp 0 p, 50);

  fun goal (n:int) (ps,qs) =
     Pretty.pr (TextIO.stdOut,
                Pretty.blo (4, [Pretty.str(" " ^ Int.toString n ^ ". "),
                                formList ps, Pretty.brk 2, Pretty.str"|- ",
                                formList qs]),
                50);
  end;
```

Exercise 10.18 The inner parenthesis pair in `q & (p1 --> (p2 | r))` is redundant because `|` has greater precedence than `-->`; our pretty printing often includes such needless parentheses. Suggest modifications to the function *form* that would prevent this.

Exercise 10.19 Explain how quantified formulæ are displayed.

10.7 *Unification*

Hal attempts to unify atomic formulæ in goals. Its basic unification algorithm takes terms containing no bound variables. Given a pair of terms, it computes a set of (variable, term) replacements to make them identical, or reports that the terms cannot be unified. Performing the replacements is called ***instantiation***. Unification involves three cases:

Function applications. Two function applications can be unified only if they apply the same function; clearly no instantiation can transform $f(?a)$ and $g(b, ?c)$ into identical terms. To unify $g(t_1, t_2)$ with $g(u_1, u_2)$ involves unifying t_1 with u_1 and t_2 with u_2 consistently — thus $g(?a, ?a)$ cannot be unified with $g(b, c)$ because a variable ($?a$) cannot be replaced by two different constants (b and c).

The unification of $f(t_1, \ldots, t_n)$ with $f(u_1, \ldots, u_n)$ begins by unifying t_1 with u_1, then applies the resulting replacements to the remaining terms. The next step is unifying t_2 with u_2 and applying the new replacements to the remaining terms, and so forth. If any unifications fail then the function applications are not unifiable. The corresponding arguments can be chosen for unification in any order without significantly affecting the outcome.

Parameters. Two parameters can be unified only if they have the same name. A parameter cannot be unified with a function application.

Variables. The remaining and most interesting case is unifying a variable $?a$ with a term t (distinct from $?a$). If $?a$ does not occur in t then unification succeeds, yielding the replacement $(?a, t)$. If $?a$ does occur in t then unification fails — for possibly two different reasons:

- If $?a$ occurs in a parameter of t, then $?a$ is a 'forbidden variable' for that parameter and for the term. Replacing $?a$ by t would violate the proviso of some quantifier rule.
- If t properly contains $?a$ then the terms cannot be unified: no term can contain itself. For example, no replacement can transform $f(?a)$ and $?a$ into identical terms.

This is the notorious ***occurs check***, which most Prolog interpreters omit because of its cost. For theorem proving, soundness must have priority over efficiency; the occurs check must be performed.

Examples. To unify $g(?a, f(?c))$ with $g(f(?b), ?a)$, first unify $?a$ with $f(?b)$, a trivial step. After replacing $?a$ by $f(?b)$ in the remaining arguments, unify $f(?c)$ with $f(?b)$. This replaces $?c$ by $?b$. The outcome can be given as the set $\{?a \mapsto f(?b),\ ?c \mapsto ?b\}$. The unified formula is $g(f(?b), f(?b))$.

Here is another example. To unify $g(?a, f(?a))$ with $g(f(?b), ?b)$, the first step again replaces $?a$ by $f(?b)$. The next task is unifying $f(f(?b))$ with $?b$ — which is impossible because $f(f(?b))$ contains $?b$. Unification fails.

Instantiation of parameters. Recall that each parameter carries a list of forbidden variables; $b_{?a}$ must never be part of a term t substituted for $?a$. When a legal replacement is performed, the occurrence of $?a$ in $b_{?a}$ is replaced by the variables contained in t, not by t itself. For instance, replacing $?a$ by $g(?c, f(?d))$ transforms $b_{?a}$ into $b_{?c,?d}$. Any substitution for $?c$ or $?d$ is effectively a substitution for $?a$, and therefore $?c$ and $?d$ are forbidden to the parameter.

For example, to unify $g(?a, f(b_{?a}))$ with $g(h(?c, ?d), ?c)$, the first step is to replace $?a$ by $h(?c, ?d)$. The second arguments of g become $f(b_{?c,?d})$ and $?c$; these terms are not unifiable because $?c$ is forbidden to the parameter $b_{?c,?d}$.

Skolem functions. Parameters are not widely used in theorem proving; more traditional are ***Skolem functions***. The rules ∀:left and ∃:right, instead of creating the parameter $b_{?a_1,\ldots,?a_k}$, could introduce the term $b(?a_1, \ldots, ?a_k)$. Here, b is a function symbol not appearing elsewhere in the proof. The term behaves like a parameter; the occurs check prevents unification from violating the rules' provisos. Skolem functions have advantages in automatic proof procedures, but they destroy the formula's readability; in higher-order logic they can even cause faulty reasoning.

The ML code. The signature specifies unification and instantiation functions, as well as an exception *Failed* for reporting non-unifiable terms:

```
signature UNIFY =
  sig
  exception Failed
  val atoms    : Fol.form * Fol.form -> Fol.term StringDict.t
  val instTerm : Fol.term StringDict.t -> Fol.term -> Fol.term
  val instForm : Fol.term StringDict.t -> Fol.form -> Fol.form
  val instGoal : Fol.term StringDict.t -> Fol.goal -> Fol.goal
  end;
```

The function *atoms* attempts to unify two atomic formulæ, while *instTerm*, *instForm* and *instGoal* apply replacements to terms, formulæ and goals, respectively.

We represent a set of replacements by a dictionary, using structure *StringDict* (Section 7.10); variable names are strings.

An atomic formula consists of a predicate applied to an argument list, such as $P(t_1, \ldots, t_n)$. Unifying two atomic formulæ is essentially the same as unifying two function applications; the predicates must be the same and the corresponding argument pairs must be simultaneously unifiable.

Structure *Unify* (Figure 10.5) implements unification. The key functions are declared within *unifyLists* in order to have access to *env*, the environment of replacements. Collecting the replacements in *env* is more efficient than applying each replacement as it is generated. Replacements are regarded as cumulative rather than simultaneous, just as in the λ-calculus interpreter's treatment of definitions (Section 9.7). Simultaneous substitution by

$$\{?b \mapsto g(z),\ ?a \mapsto f(?b)\}$$

would replace $?a$ by $f(?b)$, but our functions replace $?a$ by $f(g(z))$. This is the correct treatment for our unification algorithm.

Here are some remarks about the functions declared in *unifyLists*:

- *chase t* replaces the term t, if it is a variable, by its assignment in *env*. Nonvariable terms are returned without change; at each stage, unification is concerned only with a term's outer form.
- *occurs a t* tests whether the variable $?a$ occurs within term t; like *chase*, it looks up variables in the environment.
- *occsl a ts* tests whether the variable $?a$ occurs within the list of terms *ts*.
- *unify*(t, u) creates a new environment from *env* by unifying t with u, if possible, otherwise raising exception *Failed*. If t and u are variables then they must have no assignment in *env*; violating this condition could result in a variable having two assignments!
- *unifyl*(ts, us) simultaneously unifies the corresponding members of the lists *ts* and *us*, raising *Failed* if their lengths differ. (If two terms are not unifiable, the exception will arise in *unify*, not *unifyl*.)

The implementation is purely functional. Representing variables by references might be more efficient — updating a variable would perform a replacement, with no need for environments — but is incompatible with tactical theorem proving. Applying a tactic to a proof state should create a new state, leaving the

Figure 10.5 *Unification*

```
structure Unify : UNIFY =
  struct
  exception Failed;
  fun unifyLists env =
    let fun chase (Fol.Var a) = (chase(StringDict.lookup(env,a))
                                 handle StringDict.E _ => Fol.Var a)
          | chase t           = t
        fun occurs a (Fol.Fun(_,ts))     = occsl a ts
          | occurs a (Fol.Param(_,bs))   = occsl a (map Fol.Var bs)
          | occurs a (Fol.Var b)         =
                (a=b) orelse (occurs a (StringDict.lookup(env,b))
                              handle StringDict.E _ => false)
          | occurs a _                   = false
        and occsl a = List.exists (occurs a)
        and unify (Fol.Var a, t)         =
              if t = Fol.Var a then env
              else if occurs a t then raise Failed
                                 else StringDict.update(env,a,t)
          | unify (t, Fol.Var a) = unify (Fol.Var a, t)
          | unify (Fol.Param(a,_), Fol.Param(b,_)) = if a=b then env
                                                    else raise Failed
          | unify (Fol.Fun(a,ts), Fol.Fun(b,us))   = if a=b then unifyl(ts,us)
                                                    else raise Failed
          | unify _                                = raise Failed
       and unifyl ([],[])        = env
         | unifyl (t::ts, u::us) = unifyLists (unify (chase t, chase u)) (ts,us)
         | unifyl _              = raise Failed
     in  unifyl  end

  fun atoms (Fol.Pred(a,ts), Fol.Pred(b,us)) =
        if a=b then unifyLists StringDict.empty (ts,us) else raise Failed
    | atoms _                                 = raise Failed;

  fun instTerm env (Fol.Fun(a,ts))     = Fol.Fun(a, map (instTerm env) ts)
    | instTerm env (Fol.Param(a,bs)) =
      Fol.Param(a, foldr Fol.termVars [] (map (instTerm env o Fol.Var) bs))
    | instTerm env (Fol.Var a)         = (instTerm env (StringDict.lookup(env,a))
                                          handle StringDict.E _ => Fol.Var a)
    | instTerm env t                   = t;

  fun instForm env (Fol.Pred(a,ts))       = Fol.Pred(a, map (instTerm env) ts)
    | instForm env (Fol.Conn(b,ps))       = Fol.Conn(b, map (instForm env) ps)
    | instForm env (Fol.Quant(qnt,b,p)) = Fol.Quant(qnt, b, instForm env p);

  fun instGoal env (ps,qs) = (map (instForm env) ps, map (instForm env) qs);
  end;
```

original state unchanged so that other tactics can be tried. A unification algorithm could employ imperative techniques provided they were invisible outside.

The unification function raises exception $Failed$ when two terms cannot be unified. As in parsing, the failure may be detected in deeply nested recursive calls; the exception propagates upwards. This is a typical case where exceptions work well.

Function $instTerm$ substitutes in parameters as described above. Each forbidden variable is replaced by the list of variables in the term resulting from the substitution. This could be done using $List.concat$, but the combination of $foldr$ and $termVars$ performs less copying.

Efficient unification algorithms. The algorithm presented here can take exponential time, in highly exceptional cases. In practice, it is quite usable. More efficient algorithms exist. The linear time algorithm of Paterson and Wegman (1978) is usually regarded as too complicated for practical use. The algorithm of Martelli and Montanari (1982) is almost linear and is intended to be usable. However, Corbin and Bidoit (1983) propose an algorithm based upon the naïve one, but representing terms by graphs (really, pointers) instead of trees. They claim it to be superior to the almost linear algorithms because of its simplicity, despite needing quadratic time. Ružička and Prívara (1988) have refined this approach to be almost linear too.

Exercise 10.20 What could happen if this line were omitted from $unify$?

```
if t = Fol.Var a then env else
```

Tactics and the proof state

Our proof procedure for the sequent calculus operates by successive refinements, working backwards from a goal. The proof tree grows up from the root. Coding the procedure in ML requires a data structure for ***proof states***, which are partially constructed proofs. Inference rules will be implemented as functions, called ***tactics***, on proof states.

10.8 *The proof state*

A formal proof is a tree whose every node carries a sequent and the name of a rule. Each node's branches lead to the premises of its rule. But the ML datatype corresponding to such trees is unsuitable for our purposes. Backward proof requires access to the leaves, not to the root. Extending the proof turns a leaf into a branch node, and would require copying part of the tree. The intermediate nodes would play no useful rôle in the search for a proof.

Hal omits the intermediate nodes altogether. A partial proof tree contains just two parts of the proof. The root, or ***main goal***, is the formula we first set out

to prove. The leaves, or ***current subgoals***, are the sequents that remain to be proved.

A goal ϕ paired with the singleton subgoal list $[\vdash \phi]$ represents the initial state of a proof of ϕ; no rules have yet been applied. A goal ϕ paired with the empty subgoal list is a final state, and represents a finished proof.

If the full proof tree is not stored, how can we be certain that a Hal proof is correct? The answer is to hide the representation of proof states using an abstract type $state$, providing a limited set of operations — to create an initial state, to examine the contents of a state, to test for a final state, and to transform a state into a new state by some rule of inference.

If greater security is required, the proof could be saved and checked by a separate program. Bear in mind that proofs of real theorems can be extremely large, and that no amount of machine checking can provide absolute security. Our programs and proof systems are fallible — as are the theories we use to reduce 'real world' tasks to logic.

Approaches to formalizing an inference system. While developing Edinburgh LCF, Robin Milner conceived the idea of defining an inference system as an abstract type. He designed ML's type system to support this application. LCF's type thm denotes the set of theorems of the logic. Functions with result type thm implement the axioms and inference rules.

Implementing the inference rules as functions from theorems to theorems supports forward proof, LCF's primitive style of reasoning. To support backward proof, LCF provides tactics. LCF tactics represent a partial proof by a function of type $thm\ list \to thm$. This function proves the main goal, using inference rules, when supplied with theorems for each of the subgoals. A finished proof can be supplied with the empty list to prove the main goal. The classic description (Gordon *et al.*, 1979) is out of print, but my book on LCF also describes this work (Paulson, 1987).

Hal differs from LCF in implementing the inference rules as functions on proof states, not on theorems. These functions are themselves tactics and support backward proof as the primitive style. They do not support forward proof. The approach supports unification; tactics may update meta-variables in the proof state.

Isabelle (Paulson, 1994) uses yet another approach. Rules and proof states have a common representation in the typed λ-calculus. Combining these objects yields both forward and backward proof. This requires some form of higher-order unification (Huet, 1975).

10.9 *The ML signature*

Signature *RULE* specifies the abstract type of proof states, with its operations (Figure 10.6). Each value of type $state$ contains a formula (the main goal) and a list of sequents (the subgoals). Although we cannot tell from the signature, each $state$ contains additional information for internal use.

Figure 10.6 *The signature* RULE

```
signature RULE =
  sig
  type state
  type tactic  = state -> state ImpSeq.t
  val main    : state -> Fol.form
  val subgoals : state -> Fol.goal list
  val initial  : Fol.form -> state
  val final    : state -> bool
  val basic    : int -> tactic
  val unify    : int -> tactic
  val conjL    : int -> tactic
  val conjR    : int -> tactic
  val disjL    : int -> tactic
  val disjR    : int -> tactic
  val impL     : int -> tactic
  val impR     : int -> tactic
  val negL     : int -> tactic
  val negR     : int -> tactic
  val iffL     : int -> tactic
  val iffR     : int -> tactic
  val allL     : int -> tactic
  val allR     : int -> tactic
  val exL      : int -> tactic
  val exR      : int -> tactic
  end;
```

Type $tactic$ abbreviates the function type

$$state \to state\ ImpSeq.t,$$

where $ImpSeq$ is the structure for lazy lists presented in Section 8.4. A tactic maps a state to a sequence of possible next states. The primitive tactics generate finite sequences, typically of length zero or one. A complex tactic, say for depth-first search, could generate an infinite sequence of states.

The function $initial$ creates initial states containing a given formula as the main goal and the only subgoal. The predicate $final$ tests whether a proof state is final, containing no subgoals.

The other functions in the signature are the primitive tactics, which define the inference rules of the sequent calculus. Later, we shall introduce ***tacticals*** for combining tactics.

The subgoals of a proof state are numbered starting from 1. Each primitive tactic, given an integer argument i and a state, applies some rule of the sequent calculus to subgoal i, creating a new state. For instance, calling

$$conjL\ 3\ st$$

applies $\wedge$:left to subgoal 3 of state st. If this subgoal has the form $\phi \wedge \psi$, $\Gamma \vdash \Delta$ then subgoal 3 of the next state will be ϕ, ψ, $\Gamma \vdash \Delta$. Otherwise, $\wedge$:left is not applicable to the subgoal and there can be no next state; $conjL$ will return the empty sequence.

If subgoal 5 of st is $\Gamma \vdash \Delta, \phi \wedge \psi$, then

$$conjR\ 5\ st$$

will make a new state whose subgoal 5 is $\Gamma \vdash \Delta, \phi$ and whose subgoal 6 is $\Gamma \vdash \Delta, \psi$. Subgoals numbered greater than 5 in st are shifted up.

Calling $basic\ i\ st$ checks whether subgoal i of state st is a basic sequent. If subgoal i has a common formula on both sides then it is deleted in the next state. Otherwise, the tactic signals failure by returning the empty sequence. A more elaborate treatment of basic sequents is tactic $unify$.

Calling $unify\ i\ st$ attempts to solve subgoal i of state st by converting it into a basic sequent. If it can unify a formula on the left with a formula on the right then it deletes subgoal i and applies the unifying substitution to the rest of the proof state. There may be several different pairs of unifiable formulæ; applying $unify$ to the subgoal

$$P(?a), P(?b) \vdash P(f(c)), P(c)$$

generates a sequence of four next states. Only the first of these is computed, with the others available upon demand, since sequences are lazy.

10.10 *Tactics for basic sequents*

Structure *Rule* is presented in parts. The first part (Figure 10.7) defines the representation of type *state* and its primitive operations, and declares tactics for basic sequents.

Declaring type state. The `datatype` declaration introduces type *state* with its constructor *State*. The constructor is not exported, allowing access to the representation only inside the structure body. Type *tactic* is declared to abbreviate the type of functions from state to state sequences.

Functions *main* and *subgoals* return the corresponding parts of a proof state. The third component of a proof state is an integer, for generating unique names in quantifier rules. Its value is initially 0 and is increased as necessary when the next state is created. If this name counter were kept in a reference cell and updated by assignment, much of the code would be simpler — especially where the counter plays no rôle. However, applying a quantifier rule to a state would affect all states sharing that reference. Resetting the counter to 0, while producing shorter names, could also lead to re-use of names and faulty reasoning. It is safest to ensure that all tactics are purely functional.

Calling *initial* p creates a state containing the sequent $\vdash p$ as its only subgoal, with p as its main goal and 0 for its variable counter. Predicate *final* tests for an empty subgoal list.

The definitions of basic and unify. All tactics are expressed using *spliceGoals*, a function to replace subgoal i by a new list of subgoals in a state. The *List* functions *take* and *drop* extract the subgoals before and after i, so that the new subgoals can be spliced into the correct place.

The declaration of *propRule* illustrates how proof states are processed. This function makes a tactic from a function *goalF* of type $goal \to goal\ list$. Applied to an integer i and a state, it supplies subgoal i to *goalF* and splices in the resulting subgoals; it returns the new state as a one-element sequence. It returns the empty sequence if any exception is raised. Exception *Subscript* results from the call *List*`.`*nth*`(`*gs*`,` *i*`-1)` if there is no ith subgoal; recall that *nth* numbers a list's elements starting from zero. Other exceptions, such as *Match*, can result from *goalF*.

The tactic *basic* is a simple application of *propRule*. It supplies as *goalF* a function that checks whether the goal (ps, qs) is a basic sequent. If so then

Figure 10.7 *First part of* Rule — *tactics for basic sequents*

```
structure Rule :> RULE =
  struct
  datatype state = State of Fol.goal list * Fol.form * int

  type tactic = state -> state ImpSeq.t;

  fun main     (State(gs,p,_)) = p
  and subgoals (State(gs,p,_)) = gs;

  fun initial p = State([ ([],[p]) ], p, 0);

  fun final (State(gs,_,_)) = null gs;

  fun spliceGoals gs newgs i = List.take(gs,i-1) @ newgs @ List.drop(gs,i);

  fun propRule goalF i (State(gs,p,n)) =
      let val gs2 = spliceGoals gs (goalF (List.nth(gs,i-1))) i
      in  ImpSeq.fromList [State(gs2, p, n)]  end
      handle _ => ImpSeq.empty;

  val basic = propRule
                (fn (ps,qs) =>
                 if List.exists (fn p => List.exists (fn q => p=q) qs) ps
                 then [] else raise Match);

  fun unifiable ([], _)     = ImpSeq.empty
    | unifiable (p::ps, qs) =
        let fun find []      = unifiable (ps,qs)
              | find (q::qs) = ImpSeq.cons(Unify.atoms(p,q), fn()=> find qs)
                               handle Unify.Failed => find qs
        in  find qs  end;

  fun inst env (gs,p,n) =
        State (map (Unify.instGoal env) gs,  Unify.instForm env p, n);

  fun unify i (State(gs,p,n)) =
    let val (ps,qs)   = List.nth(gs,i-1)
        fun next env  = inst env (spliceGoals gs [] i, p, n)
    in  ImpSeq.map next (unifiable(ps,qs))  end
    handle Subscript => ImpSeq.empty;
```

it returns the empty list of subgoals; the effect is to delete that subgoal from the next state. But if (ps, qs) is not a basic sequent then the function raises an exception.

The tactic *unify* is more complicated: it can return multiple next states. It calls *unifiable* to generate a sequence of unifying environments, and *inst* to apply them to the other subgoals. Function *next*, which performs the final processing, is applied via the functional $ImpSeq\,.\,map$.

The function *unifiable* takes lists ps and qs of formulæ. It returns the sequence of all environments obtained by unifying some p of ps with some q of qs. The function *find* handles the 'inner loop,' searching in qs for something to unify with p. It generates a sequence whose head is an environment and whose tail is generated by the recursive call $find\ qs$, but if $Unify\,.\,atoms$ raises an exception then the result is simply $find\ qs$.

Look out for other goals. When *unify* solves a subgoal, it may update the state so that some other subgoal becomes unprovable. Success of this tactic does not guarantee that it is the right way to find a proof; in some cases, a different tactic should be used instead. Any search procedure involving *unify* should use backtracking. On the other hand, solving a goal by *basic* is always safe.

Exercise 10.21 Give an example to justify the warning above.

10.11 *The propositional tactics*

The next part of *Rule* implements the rules for $\wedge$, $\vee$, $\neg$, $\to$ and $\leftrightarrow$. Since each connective has a 'left' rule and a 'right' rule, there are ten tactics altogether. See Figure 10.8.

The tactics employ the same basic mechanism. Search for a suitable formula on the given side, left or right; detach the connective; generate new subgoals from its operands. Each tactic returns a single next state if it succeeds. A tactic fails, returning an empty state sequence, if it cannot find a suitable formula. We can express them succinctly with the help of *propRule* and a new function, *splitConn*.

An example demonstrates the workings of *splitConn*. Given the string `"&"` and a formula list qs, it finds the first element that matches $Conn\,(\texttt{"\&"}, ps)$, raising exception *Match* if none exists. It also copies qs, omitting the matching element. It returns ps paired with the shortened qs. Note that ps contains the operands of the selected formula.

The functional *propL* helps express sequent rules. Given a sequent, it searches for a connective on the left side. It supplies the result of the *splitConn* call to

Figure 10.8 *Part of* Rule — *the propositional tactics*

```
fun splitConn a qs =
  let fun get []                                = raise Match
        | get (Fol.Conn(b,ps) :: qs)            = if a=b then ps else get qs
        | get (q::qs)                           = get qs;
      fun del []                                = []
        | del ((q as Fol.Conn(b,_)) :: qs)      = if a=b then qs
                                                  else q :: del qs
        | del (q::qs)                           = q :: del qs
  in (get qs, del qs)  end;

fun propL a leftF = propRule (fn (ps,qs) => leftF (splitConn a ps, qs));

fun propR a rightF = propRule (fn (ps,qs) => rightF (ps, splitConn a qs));

val conjL = propL "&" (fn (([p1,p2], ps), qs) => [(p1::p2::ps, qs)]);

val conjR = propR "&"
    (fn (ps, ([q1,q2], qs)) => [(ps, q1::qs),  (ps, q2::qs)]);

val disjL = propL "|"
    (fn (([p1,p2], ps), qs) => [(p1::ps, qs),  (p2::ps, qs)]);

val disjR = propR "|" (fn (ps, ([q1,q2], qs)) => [(ps, q1::q2::qs)]);

val impL = propL "-->"
    (fn (([p1,p2], ps), qs) => [(p2::ps, qs),  (ps, p1::qs)]);

val impR = propR "-->" (fn (ps, ([q1,q2], qs)) => [(q1::ps, q2::qs)]);

val negL = propL "~" (fn (([p], ps), qs) => [(ps, p::qs)]);

val negR = propR "~" (fn (ps, ([q], qs)) => [(q::ps, qs)]);

val iffL = propL "<->"
    (fn (([p1,p2], ps), qs) => [(p1::p2::ps, qs),  (ps, p1::p2::qs)]);

val iffR = propR "<->"
    (fn (ps, ([q1,q2], qs)) => [(q1::ps, q2::qs),  (q2::ps, q1::qs)]);
```

another function *leftF*, which creates new subgoals. The functional *propR* is similar, but searches on the right side.

The tactics are given by `val` declarations, since they have no explicit arguments. Each tactic consists of a call to *propL* or *propR*. Each passes in `fn` notation the argument *leftF* or *rightF*. Each function takes an analysed subgoal and returns one or two subgoals. Thus *conjL* searches for a conjunction in the left part and inserts the two conjuncts into the new subgoal, while *conjR* searches for a conjunction in the right part and makes two subgoals.

10.12 *The quantifier tactics*

The mechanism presented above is easily modified to express the quantifier tactics. There are a few differences from the propositional case. The code appears in Figure 10.9, which completes the presentation of structure *Rule*.

The function *splitQuant* closely resembles *splitConn*. It finds the first formula having a particular quantifier, `"ALL"` or `"EX"`. It returns the entire formula (rather than its operands) because certain quantifier tactics retain it in the subgoal.

Although our sequent calculus is defined using multisets, it is implemented using lists. The formulæ in a sequent are ordered; if the list contains two suitable formulæ, the leftmost one will be found. To respect the concept of multisets, Hal provides no way of reordering the formulæ. The quantifier tactics ensure that no formula is permanently excluded from consideration.

The tactics need a source of fresh names for variables and parameters. Calling *letter* n, for $0 \leq n \leq 25$, returns a one-character string from `"a"` to `"z"`. The function *gensym* — whose name dates from Lisp antiquity — generates a string from a natural number. Its result contains a base 26 numeral whose 'digits' are lower-case letters; the prefix `"_"` prevents clashes with names supplied from outside.

The functional *quantRule* creates a tactic from a function *goalF*. It supplies both a subgoal and a fresh name to *goalF*, which accordingly has type $goal \times string \to goal\ list$. When constructing the next state, it increments the variable counter. Otherwise, *quantRule* is identical to *propRule*.

Each tactic is expressed by applying *quantRule* to a function in `fn` notation. The function takes the subgoal (ps, qs) and the fresh name b; it returns one subgoal.

Tactics *allL* and *exR* expand a quantified formula. They substitute a variable with the name b into its body. They include the quantified formula (bound to *qntForm* using `as`) in the subgoal. The formula is placed last in the list; thus, other quantified formulæ can be selected when the tactic is next applied.

Figure 10.9 *Final part of* Rule — *the quantifier tactics*

```
fun splitQuant qnt qs =
  let fun get [] = raise Match
        | get ((q as Fol.Quant(qnt2,_,p)) :: qs) = if qnt=qnt2 then q
                                                   else get qs
        | get (q::qs) = get qs;
      fun del [] = []
        | del ((q as Fol.Quant(qnt2,_,p)) :: qs) = if qnt=qnt2 then qs
                                                   else q :: del qs
        | del (q::qs) = q :: del qs
  in (get qs, del qs) end;

fun letter n = String.substring("abcdefghijklmnopqrstuvwxyz", n, 1)

fun gensym n =
   if n<26 then "_" ^ letter n
   else gensym(n div 26) ^ letter(n mod 26);

fun quantRule goalF i (State(gs,p,n)) =
    let val gs2 = spliceGoals gs (goalF (List.nth(gs,i-1), gensym n)) i
    in ImpSeq.fromList [State(gs2, p, n+1)] end
    handle _ => ImpSeq.empty;

val allL = quantRule (fn ((ps,qs), b) =>
    let val (qntForm as Fol.Quant(_,_,p), ps') = splitQuant "ALL" ps
        val px = Fol.subst 0 (Fol.Var b) p
    in [(px :: ps' @ [qntForm], qs)] end);

val allR = quantRule (fn ((ps,qs), b) =>
    let val (Fol.Quant(_,_,q), qs') = splitQuant "ALL" qs
        val vars = Fol.goalVars ((ps,qs), [])
        val qx   = Fol.subst 0 (Fol.Param(b, vars)) q
    in [(ps, qx::qs')] end);

val exL = quantRule (fn ((ps,qs), b) =>
    let val (Fol.Quant(_,_,p), ps') = splitQuant "EX" ps
        val vars = Fol.goalVars ((ps,qs), [])
        val px   = Fol.subst 0 (Fol.Param(b, vars)) p
    in [(px::ps', qs)] end);

val exR = quantRule (fn ((ps,qs), b) =>
    let val (qntForm as Fol.Quant(_,_,q), qs') = splitQuant "EX" qs
        val qx = Fol.subst 0 (Fol.Var b) q
    in [(ps, qx :: qs' @ [qntForm])] end);
end;
```

Tactics $allR$ and exL select a quantified formula and substitute a parameter into its body. The parameter has the name b and carries, as forbidden variables, all the variables in the subgoal.

As we reach the end of $Rule$, we should remember that the tactics declared in it are the only means of creating values of type $state$. All proof procedures — even if they demonstrate validity using sophisticated data structures — must ultimately apply these tactics, constructing a formal proof. If the code given above is correct, and the ML system is correct, then Hal proofs are guaranteed to be sound. No coding errors after this point can yield faulty proofs. This security comes from defining $state$ as an abstract type.

Exercise 10.22 Suggest a representation of type $state$ that would store the entire proof tree. Best would be an encoding that uses little space while allowing the proof tree to be reconstructed. Sketch the modifications to *RULE* and $Rule$.

Exercise 10.23 Our set of tactics provides no way of using a previously proved theorem in a proof. A tactic based on the rule

$$\frac{\vdash \phi \quad \phi, \Gamma \vdash \Delta}{\Gamma \vdash \Delta}$$

could insert the theorem $\vdash \phi$ as a lemma into a goal.[3] Describe how such a tactic could be implemented.

Exercise 10.24 'Structure $Rule$ does not involve $ParseFol$ or $DisplayFol$, so faults in parsing and pretty printing cannot result in the construction of faulty proofs.' Comment on this statement.

Searching for proofs

Most of the programming is now behind us. We are nearly ready to attempt proofs on the machine. We shall implement a package of commands for applying tactics to a goal. This will demonstrate the treatment of proof states, but will also reveal the tedium of rule-by-rule proof checking. Tacticals, by providing control structures for tactics, will allow us to express automatic theorem provers in a few lines of code.

[3] This rule is a special case of 'cut'; its first premise could be $\Gamma \vdash \Delta, \phi$.

10.13 *Commands for transforming proof states*

The user interface does not read from the terminal, but consists of a set of commands to be invoked from the ML top level. This is typical of tactical theorem provers. The most important command is 'apply a tactic,' and the tactic could be given by an arbitrary ML expression; therefore, the command language is ML itself. Remember that ML stands for Meta Language.

Hal's interface is crude. It merely provides commands for setting, updating and inspecting a stored proof state. Practical theorem proving requires additional facilities, such as an *undo* command for reverting to a previous state. Because a tactic can return several next states, applying tactics defines a search tree rooted in the initial state. A graphical user interface would provide means for exploring this tree. To keep the code simple, such facilities are left as exercises. Some ML systems can communicate with scripting languages such as Tcl/Tk, making it easy to put windows and menus on the screen. Good interface design requires, in addition, careful study of users' work habits.

Signature *COMMAND* specifies the user interface:

```
signature COMMAND =
  sig
  val goal     : string -> unit
  val by       : Rule.tactic -> unit
  val pr       : Rule.state -> unit
  val getState : unit -> Rule.state
  end;
```

The interface consists of the following items, which (except pr) act upon a stored proof state:

- The $goal$ command accepts a formula ϕ, given as a string; it sets the stored proof state to the initial state for ϕ.
- The by command applies a tactic to the current state. If the resulting sequence of next states is non-empty, its head is taken to update the stored proof state. Otherwise, the tactic has failed; an error message is displayed.
- The pr command prints its argument, a proof state, on the terminal.
- The function $getState$ returns the stored proof state.

Structure $Command$ implements these items (Figure 10.10). The current state is stored in a reference cell, initialised with the fictitious goal `"No goal yet!"`.

Recall that a parameter, such as $b_{?c,?d}$, is displayed simply as b. The interface displays a table of each parameter, with its forbidden variables. Function $printpar$ prints the line

Figure 10.10 *User interface commands*

```
structure Command : COMMAND =
  struct

  val currState = ref (Rule.initial (Fol.Pred("No goal yet!",[])));

  fun question (s,z) = " ?" :: s :: z;

  fun printParam (a,[]) = ()      (*print a line of parameter table*)
    | printParam (a,ts) =
       print (String.concat (a :: " not in " ::
                             foldr question ["\n"] ts));

  fun printGoals (_, [])    = ()
    | printGoals (n, g::gs) = (DisplayFol.goal n g;  printGoals (n+1,gs));

  fun pr st =  (*print a proof state*)
      let val p  = Rule.main st
          and gs = Rule.subgoals st
      in  DisplayFol.form p;
          if Rule.final st then print "No subgoals left!\n"
          else (printGoals (1,gs);
                app printParam (foldr Fol.goalParams [] gs))
      end;

  (*print new state, then set it*)
  fun setState state = (pr state;  currState := state);

  val goal = setState o Rule.initial o ParseFol.read;

  fun by tac = setState (ImpSeq.hd (tac (!currState)))
               handle ImpSeq.Empty => print "** Tactic FAILED! **\n"

  fun getState() = !currState;
  end;
```

```
b not in ?c ?d
```

for $b_{?c,?d}$; it prints nothing at all for a parameter that has no forbidden variables. Function *printgoals* prints a list of numbered subgoals. With the help of these functions, *pr* prints a state: its main goal, its subgoal list, and its table of parameters.

Exercise 10.25 Design and implement an *undo* command that cancels the effect of the most recent *by* command. Repeated *undo* commands should revert to earlier and earlier states.

Exercise 10.26 There are many ways of managing the search tree of states. The interface could explore a single path through the tree. Each node would store a sequence of possible next states, marking one as the active branch. Changing the active branch at any node would select a different path. Develop this idea.

10.14 *Two sample proofs using tactics*

To demonstrate the tactics and the user interface, let us do some proofs on the machine. For convenience in referring to commands, we open the corresponding module:

```
open Command;
```

Now we can perform proofs. The first example is brief, a proof of $\phi \wedge \psi \to \psi \wedge \phi$. The *goal* command gives this formula to Hal.

```
goal "P & Q  -->  Q & P";
> P & Q --> Q & P
>  1. empty  |-  P & Q --> Q & P
```

Now $\phi \wedge \psi \to \psi \wedge \phi$ is the main goal and the only subgoal. We must apply $\to$:right to subgoal 1; no other step is possible:

```
by (Rule.impR 1);
> P & Q --> Q & P
>  1.  P & Q  |-  Q & P
```

Subgoal 1 becomes $\phi \wedge \psi \vdash \psi \wedge \phi$, which we have proved on paper. Although $\wedge$:right could be applied to this goal, $\wedge$:left leads to a shorter proof because it makes only one subgoal.

```
by (Rule.conjL 1);
> P & Q --> Q & P
```

```
> 1.  P, Q  |-  Q & P
```

Again we have no choice. We must apply $\wedge$:right to subgoal 1. Here is what happens if we try a different tactic:

```
by (Rule.disjR 1);
> ** Tactic FAILED! **
```

This time, apply $\wedge$:right. It makes two subgoals.

```
by (Rule.conjR 1);
> P & Q --> Q & P
>  1.  P, Q  |-  Q
>  2.  P, Q  |-  P
```

Tactics are usually applied to subgoal 1; let us tackle subgoal 2 for variety. It is a basic sequent, so it falls to $Rule\,.\,basic$.

```
by (Rule.basic 2);
> P & Q --> Q & P
>  1.  P, Q  |-  Q
```

Subgoal 1 is also a basic sequent. Solving it terminates the proof.

```
by (Rule.basic 1);
> P & Q --> Q & P
> No subgoals left!
```

Most theorem provers provide some means of storing theorems once proved, but this is not possible in Hal. We go on to the next example, $\exists z\,.\,\phi(z) \rightarrow \forall x\,.\,\phi(x)$, which was discussed earlier.

```
goal "EX z.  P(z) --> (ALL x.  P(x))";
> EX z.  P(z) --> (ALL x.  P(x))
>  1. empty  |-  EX z.  P(z) --> (ALL x.  P(x))
```

The only possible step is to apply $\exists$:right to subgoal 1. The tactic generates a variable called `?_a`.

```
by (Rule.exR 1);
> EX z.  P(z) --> (ALL x.  P(x))
>  1. empty
>      |-  P(?_a) --> (ALL x.  P(x)),
>          EX z.  P(z) --> (ALL x.  P(x))
```

We could apply $\exists$:right again, but it seems sensible to analyse the other formula in subgoal 1. So we apply $\rightarrow$:right.

```
by (Rule.impR 1);
> EX z.  P(z) --> (ALL x.  P(x))
```

```
>  1.  P(?_a)
>      |-  ALL x.  P(x),
>          EX z.  P(z) --> (ALL x.  P(x))
```

Continuing to work on the first formula, we apply ∀:right. The tactic generates a parameter called _b, with ?_a as its forbidden variable. A table of parameters is now displayed.

```
by (Rule.allR 1);
> EX z.  P(z) --> (ALL x.  P(x))
>  1.  P(?_a) |-  P(_b),
>                 EX z.  P(z) --> (ALL x.  P(x))
> _b not in  ?_a
```

Since the subgoal contains P(?_a) on the left and P(_b) on the right, we could try unifying these formulæ by calling $Rule.unify$. However, the forbidden variable of _b prevents this unification. Replacing ?_a by _b would violate the proviso of ∀:right.

```
by (Rule.unify 1);
> ** Tactic FAILED! **
```

The situation is like it was at the start of the proof, except that the subgoal contains two new atomic formulæ. Since they are not unifiable, we have no choice but to expand the quantifier again, using ∃:right. The variable ?_c is created.

```
by (Rule.exR 1);
> EX z.  P(z) --> (ALL x.  P(x))
>  1.  P(?_a)
>      |-  P(?_c) --> (ALL x.  P(x)), P(_b),
>          EX z.  P(z) --> (ALL x.  P(x))
> _b not in  ?_a
```

The proof continues as it did before, with the two atomic formulæ carried along. We avoid applying ∃:right a third time and instead apply →:right.

```
by (Rule.impR 1);
> EX z.  P(z) --> (ALL x.  P(x))
>  1.  P(?_c), P(?_a)
>      |-  ALL x.  P(x), P(_b),
>          EX z.  P(z) --> (ALL x.  P(x))
> _b not in  ?_a
```

The subgoal has a new formula on the left, namely P(?_c), and ?_c is not a forbidden variable of _b. Therefore P(?_c) and P(_b) are unifiable.

```
by (Rule.unify 1);
```

```
> EX z.  P(z) --> (ALL x.  P(x))
> No subgoals left!
```

Although the first attempt with $Rule$ **.** $unify$ failed, a successful proof was finally found. This demonstrates how parameters and variables behave in practice.

10.15 *Tacticals*

The sample proofs of the previous section are unusually short. The proof of even a simple formula can require many steps. To convince yourself of this, try proving

$$((\phi \leftrightarrow \psi) \leftrightarrow \chi) \leftrightarrow (\phi \leftrightarrow (\psi \leftrightarrow \chi)).$$

Although proofs are long, each step is usually obvious. Often, only one or two rules can be applied to a subgoal. Moreover, the subgoals can be tackled in any order because a successful proof must prove them all. We can always work on subgoal 1. A respectable proof procedure can be expressed using tactics, with the help of a few control structures.

The basic tacticals. Operations on tactics are called ***tacticals*** by analogy with functions and functionals. The simplest tacticals implement the control structures of sequencing, choice and repetition. They are analogous to the parsing operators $--$, $||$ and $repeat$ (see Section 9.2). So they share the same names, with the additional infix operator $|@|$.

Tacticals in Hal involve operations on sequences. Type $multifun$ abbreviates types in the signature (Figure 10.11). The tacticals are not restricted to tactics. They are all polymorphic; type $state$ appears nowhere. Let us describe these tacticals by their effect on arbitrary functions of suitable type, not just tactics.

The tactical $--$ composes two functions sequentially. When the function $f--g$ is applied to x, it computes the sequence $f(x) = [y_1, y_2, \ldots]$ and returns the concatenation of the sequences $g(y_1)$, $g(y_2)$, ... With tactics, $--$ applies one tactic and then another to a proof state, returning all 'next next' states that result.

The tactical $||$ chooses between two functions. When the function $f||g$ is applied to x, it returns $f(x)$ if this sequence is non-empty, and otherwise returns $g(y)$. With tactics, $||$ applies one tactic to a proof state, and if it fails, tries another. The tactical $|@|$ provides a less committal form of choice; when $f|@|g$ is applied to x, it concatenates the sequences $f(x)$ and $g(x)$.

The tactics all and no can be used with tacticals to obtain effects such as repetition. For all x, $all(x)$ returns the singleton sequence $[x]$ while $no(x)$

Figure 10.11 *The signature* TACTICAL

```
infix 6 $--;
infix 5 --;
infix 0 || |@|;
signature TACTICAL =
  sig
  type ('a,'b) multifun = 'a -> 'b ImpSeq.t
  val --          : ('a,'b) multifun * ('b,'c) multifun -> ('a,'c) multifun
  val ||          : ('a,'b) multifun * ('a,'b) multifun -> ('a,'b) multifun
  val |@|         : ('a,'b) multifun * ('a,'b) multifun -> ('a,'b) multifun
  val all         : ('a,'a) multifun
  val no          : ('a,'b) multifun
  val try         : ('a,'a) multifun -> ('a,'a) multifun
  val repeat      : ('a,'a) multifun -> ('a,'a) multifun
  val repeatDeterm : ('a,'a) multifun -> ('a,'a) multifun
  val depthFirst  : ('a->bool) -> ('a,'a) multifun -> ('a,'a) multifun
  val depthIter   : ('a->bool) * int -> ('a,'a) multifun -> ('a,'a) multifun
  val firstF      : ('a -> ('b,'c) multifun) list -> 'a -> ('b,'c) multifun
  end;
```

returns the empty sequence. Thus, all succeeds with all arguments while no succeeds with none. Note that all is the identity element for $--$:

$$all -- f = f -- all = f$$

Similarly, no is the identity for `||` and `|@|`.

Implementing the tacticals. Let us turn to the structure $Tactical$ (Figure 10.12). The rôle of sequence concatenation in $--$ is clear, but its rôle in `|@|` may be obscure. What is wrong with this obvious definition?

```
fun (tac1 |@| tac2) x = ImpSeq.append(tac1 x, tac2 x);
```

This version of `|@|` may prematurely (or needlessly) call $tac2$. Defining `|@|` using $ImpSeq.concat$ ensures that $tac2$ is not called until the elements produced by $tac1$ have been exhausted. In a lazy language, the obvious definition of `|@|` would behave properly.

The tactical try attempts to apply its argument.

The tactical $repeat$ applies a function repeatedly. The result of $repeat\ f\ x$ is a sequence of values obtained from x by repeatedly applying f, such that a

Figure 10.12 *Tacticals*

```
structure Tactical : TACTICAL =
  struct
  type ('a,'b) multifun = 'a -> 'b ImpSeq.t

  fun (tac1 -- tac2) x = ImpSeq.concat (ImpSeq.map tac2 (tac1 x));

  fun (tac1 || tac2) x =
      let val y = tac1 x
      in  if ImpSeq.null y  then  tac2 x  else y  end;

  fun (tac1 |@| tac2) x =
      ImpSeq.concat(ImpSeq.cons(tac1 x,   (*delay application of tac2!*)
                                fn()=> ImpSeq.cons(tac2 x,
                                                   fn()=> ImpSeq.empty)));

  fun all x = ImpSeq.fromList [x];

  fun no x = ImpSeq.empty;

  fun try tac = tac || all;

  fun repeat tac x = (tac -- repeat tac || all) x;

  fun repeatDeterm tac x =
      let fun drep x = drep (ImpSeq.hd (tac x))
                       handle ImpSeq.Empty => x
      in  ImpSeq.fromList [drep x]  end;

  fun depthFirst pred tac x =
     (if pred x then all else tac -- depthFirst pred tac) x;

  fun depthIter (pred,d) tac x =
   let val next = ImpSeq.toList o tac
       fun dfs i (y, sf) () =
            if i<0 then sf()
            else if i<d andalso pred y
                 then ImpSeq.cons(y, foldr (dfs (i-1)) sf (next y))
                 else foldr (dfs (i-1)) sf (next y) ()
       fun deepen k = dfs k (x, fn()=> deepen (k+d)) ()
   in  deepen 0  end;

  fun orelseF (tac1, tac2) u = tac1 u || tac2 u;

  fun firstF ts = foldr orelseF (fn _ => no) ts;
  end;
```

further application of f would fail. The tactical is defined recursively, like the analogous parsing operator.

The tactical *repeatDeterm* also provides repetition. It is deterministic: it considers only the first outcome returned at each step. When the other outcomes are not needed, *repeatDeterm* is much more efficient than *repeat*.

The tactical *depthFirst* explores the search tree generated by a function. Calling *depthFirst pred f x* returns a sequence of values, all satisfying the predicate *pred*, that were obtained from x by repeatedly applying f.

The tactical *depthIter* explores the search tree using depth-first iterative deepening. It searches first to depth d, then depth $2d$, then $3d$ and so forth; this ensures that no solutions are missed. Its other arguments are as in *depthFirst*. Its rather messy implementation is based upon the code discussed in Section 5.20.

Finally, *firstF* is a convenient means of combining primitive inference rules; see Figure 10.13 below.

Some examples. In order to demonstrate the tacticals, we first open their structure, making available the infixes.

```
open Tactical;
```

Now let us prove the following formula, which concerns the associative law for conjunction:

```
goal "(P & Q) & R  -->  P & (Q & R)";
> (P & Q) & R --> P & (Q & R)
>  1. empty  |-  (P & Q) & R --> P & (Q & R)
```

The only rule that can be applied is $\to$:right. Looking ahead a bit, we can foresee two applications of $\wedge$:left. With *repeat* we can apply both rules as often as necessary:

```
by (repeat (Rule.impR 1 || Rule.conjL 1));
> (P & Q) & R --> P & (Q & R)
>  1.  P, Q, R  |-  P & (Q & R)
```

Now $\wedge$:right must be applied twice. We repeatedly apply the corresponding tactic along with *Rule.basic*, which detects basic sequents:

```
by (repeat (Rule.basic 1 || Rule.conjR 1));
> (P & Q) & R --> P & (Q & R)
> No subgoals left!
```

We have proved the theorem using only two *by* commands; a rule-by-rule proof would have needed eight commands. For another demonstration, let us prove a theorem using one fancy tactic. Take our old quantifier example:

```
goal "EX z.  P(z) --> (ALL x.  P(x))";
> EX z.  P(z) --> (ALL x.  P(x))
>  1. empty  |-  EX z.  P(z) --> (ALL x.  P(x))
```

Let us *repeat* the tactics used in Section 10.14, choosing their order carefully. Certainly *Rule*.*unify* should be tried first, since it might solve the goal altogether. And *Rule*.*exR* must be last; otherwise it will apply every time and cause an infinite loop.

```
by (repeat (Rule.unify 1 || Rule.impR 1 ||
              Rule.allR 1 || Rule.exR 1));
> EX z.  P(z) --> (ALL x.  P(x))
> No subgoals left!
```

A brief history of tacticals. Tacticals originated with Edinburgh LCF (Gordon *et al.*, 1979). Similar control structures crop up in rewriting (Paulson, 1983), for expressing rewriting methods called *conversions*. The HOL system relies upon this approach to rewriting (Gordon and Melham, 1993, Chapter 23).

Tacticals in LCF and HOL resemble our parsing operators: they use exceptions instead of returning a sequence of outcomes. Isabelle tacticals return sequences in order to allow backtracking and other search strategies (Paulson, 1994). Hal's tacticals are closely based on Isabelle's.

Tacticals traditionally have names such as *THEN*, *ORELSE*, *REPEAT*, etc., but this violates the convention that only constructor names should start with a capital letter.

Exercise 10.27 What does the tactic $repeat(f{-}{-}f)(x)$ do?

Exercise 10.28 Does $depthFirst$ really perform depth-first search? Explain in detail how it works.

Exercise 10.29 Describe situations where the sequence returned by $-{-}$ or `|@|` omits some elements that intuitively should be present. Implement new tacticals that do not have this fault. Do $-{-}$ and `|@|` have any compensating virtues?

Exercise 10.30 Tacticals $repeat$ and $depthFirst$ appear in their traditional form. Their efficiency is adequate for interactive proof but not for use in proof procedures. Code more efficient versions, not using $-{-}$.

10.16 *Automatic tactics for first-order logic*

Using tacticals, we shall code two simple tactics for automatic proof. Given a subgoal, $depth$ attempts to solve it by unification, or by breaking down some formula, or by expanding quantifiers. Quantifiers can be expanded repeatedly without limit; the tactic may run forever.

The components of *depth* are themselves useful for interactive proof, especially when *depth* fails. They are specified in signature *TAC*:

```
signature TAC =
  sig
  val safeSteps: int -> Rule.tactic
  val quant    : int -> Rule.tactic
  val step     : int -> Rule.tactic
  val depth    : Rule.tactic
  val depthIt  : int -> Rule.tactic
  end;
```

The signature specifies five tactics:

- *safeSteps* i applies a nonempty series of 'safe' rules to subgoal i. These are any rules except ∃:right and ∀:left. Tactic *unify* is also excluded, because it can affect other goals.
- *quant* i expands quantifiers in subgoal i. It applies both ∃:right and ∀:left, if possible.
- *depth* solves all subgoals by depth-first search. It uses *safeSteps*, *unify* and *quant*.
- *step* i refines subgoal i by safe steps if possible, otherwise trying unification and quantifier expansion.
- *depthIt* d solves all subgoals by depth-first iterative deepening with increment d. It uses *step* 1, and is exhaustive but slow.

Structure *Tac* (Figure 10.13) shows how succinctly tactics can express proof procedures. The declaration of *safe* simply lists the necessary tactics, combined by *firstF*. Tactics that create one subgoal precede tactics that create two; apart from this, their order is arbitrary. Repeating *safe* via the tacticals −− and *repeatDeterm* yields *safeSteps*. We can see that *quant* expands at least one quantifier, perhaps two: if *allL* succeeds then it attempts *exR* too.

Of the two search tactics, *depth* is the faster, but is incomplete. It employs depth-first search, which can go down blind alleys. Also it applies *Rule*.*unify* whenever possible, regardless of its effect on other goals. Tactic *depthIt* remedies these points. Note that *step* uses not `||` but `|@|` to combine *Rule*.*unify* with the quantifier tactics; even if unification is successful, the search may investigate quantifier expansions too. Both search tactics test for final proof states using *Rule*.*final* (declared on page 431).

Let us try *Tac*.*depth* on some of the problems of Pelletier (1986). This is problem 39:

```
goal "~ (EX x.  ALL y.  J(x,y) <-> ~J(y,y))";
```

Figure 10.13 *The structure* Tac

```
structure Tac : TAC =
  struct
  local open Tactical Rule
    in
    val safe =
          firstF [basic,
                  conjL, disjR, impR, negL, negR, exL, allR, (*1 subgoal*)
                  conjR, disjL, impL, iffL, iffR             (*2 subgoals*)];
    fun safeSteps i = safe i -- repeatDeterm (safe i);
    fun quant i     = (allL i -- try (exR i)) || exR i;
    val depth       = depthFirst final (safeSteps 1 || unify 1 || quant 1);
    fun step i      = safeSteps i || (unify i |@| allL i |@| exR i);
    fun depthIt d   = depthIter (final, d) (step 1);
    end
  end;
```

```
> ~(EX x.  ALL y.  J(x, y) <-> ~J(y, y))
>  1. empty  |-  ~(EX x.  ALL y.  J(x, y) <-> ~J(y, y))
```

Applying *Tac*.*depth* proves it:

```
by Tac.depth;
> ~(EX x.  ALL y.  J(x, y) <-> ~J(y, y))
> No subgoals left!
```

Problem 40 is more complicated.[4]

```
goal "(EX y.  ALL x.  J(y,x) <-> ~J(x,x))  -->         \
\         ~ (ALL x.  EX y.  ALL z.  J(z,y) <-> ~ J(z,x))";
> (EX y.  ALL x.  J(y, x) <-> ~J(x, x)) -->
> ~(ALL x.  EX y.  ALL z.  J(z, y) <-> ~J(z, x))
>  1. empty
>     |-  (EX y.  ALL x.  J(y, x) <-> ~J(x, x)) -->
>          ~(ALL x.  EX y.  ALL z.  J(z, y) <-> ~J(z, x))
```

This problem too is easily proved.

```
by Tac.depth;
```

[4] Since the goal formula does not fit on one line, the \...\ escape sequence divides the string over two lines.

```
> (EX y.  ALL x.  J(y, x) <-> ~J(x, x)) -->
> ~(ALL x.  EX y.  ALL z.  J(z, y) <-> ~J(z, x))
> No subgoals left!
```

Problem 42 is harder still: *Tac*.*depth* never returns.

```
goal "~(EX y. ALL x. p(x,y) <-> ~(EX z. p(x,z) & p(z,x)))";
> ~(EX y.  ALL x. p(x, y) <-> ~(EX z. p(x, z) & p(z, x)))
>  1. empty
>      |-  ~(EX y.
>              ALL x.
>                p(x, y) <-> ~(EX z. p(x, z) & p(z, x)))
```

But our other search tactic succeeds:

```
by (Tac.depthIt 1);
> ~(EX y.  ALL x. p(x, y) <-> ~(EX z. p(x, z) & p(z, x)))
> No subgoals left!
```

It is worth reiterating that our tactics cannot compete with automatic theorem provers. They work by applying primitive inference rules, whose implementation was designed for interactive use. Their 'inner loop' (the tactic *safe*) searches for connectives in a profligate manner. No heuristics govern the expansion of quantifiers. This simple-looking example (problem 43) is not solved in a reasonable time:

```
goal "(ALL x. ALL y. q(x,y) <-> (ALL z.p(z,x)<->p(z,y))) \
\      --> (ALL x. (ALL y. q(x,y) <-> q(y,x)))";
```

Tactics work best when the logic has no known automatic proof procedure. Tacticals allow experimentation with different search procedures, while the abstract type *state* guards against faulty reasoning.

Other theorem provers. Most automatic theorem provers are based on the resolution principle (Chang and Lee, 1973). They prove a formula A by converting $\neg A$ to clause form (based upon conjunctive normal form) and deriving a contradiction. A popular resolution prover is W. McCune's Otter. For example, Quaife (1992) has used Otter for proofs in Peano arithmetic, geometry and set theory. Another impressive system is SETHEO (Letz *et al.*, 1992).

Tableau provers are less powerful, but more natural than resolution provers, since they do not require conversion to clause form. Examples include HARP (Oppacher and Suen, 1988) and the amazingly simple leanTAP (Beckert and Posegga, 1995), which consists of a few lines of Prolog. Tactic *depthIt* is loosely based upon leanTAP but is much slower.

The tactical approach combines modest automation with great flexibility. Systems apply it not for classical first-order logic, but for other logics of computational importance. LCF supports a logic of domain theory (Gordon *et al.*, 1979; Paulson, 1987). The HOL system supports Church's higher-order logic (Gordon and Melham, 1993). Nuprl

supports a form of constructive type theory (Constable *et al.*, 1986). Isabelle is a generic theorem prover, supporting several different logics (Paulson, 1994).

Exercise 10.31 Draw a diagram showing the structures, signatures and functors of Hal and their relationships.

Exercise 10.32 Implement a tactic for the rule of mathematical induction, involving the constant 0 and the successor function suc:

$$\frac{\Gamma \vdash \Delta, \phi[0/x] \quad \phi, \Gamma \vdash \Delta, \phi[suc(x)/x]}{\Gamma \vdash \Delta, \forall x \,.\, \phi}$$

proviso: x must not occur free in the conclusion

Can you foresee any difficulties in adding the tactic to an automatic proof procedure?

Exercise 10.33 Declare a tactical $someGoal$ such that, when applied to a state with n subgoals, $someGoal\ f$ is equivalent to

$$f(n) \mid\mid f(n-1) \mid\mid \ldots \mid\mid f(1).$$

What does $repeat\ \ (someGoal\ \ Rule\,.\,conjR)$ do to a proof state?

Exercise 10.34 Our proof procedure always works on subgoal 1. When might it be better to choose other subgoals?

Summary of main points

- The sequent calculus is a convenient proof system for first-order logic.
- Unification assists reasoning about quantifiers.
- The occurs check in unification is essential for soundness.
- Quantified variables can be treated like the bound variables of the λ-calculus.
- Inference rules can be provided as operations on an abstract type of theorems or proofs.
- The operations $--$, $||$ and $repeat$ have analogues throughout functional programming.
- The tactical approach allows a mixture of automatic and interactive theorem proving.