Tool Support for Logics of Programs

Lawrence C. Paulson

Computer Laboratory, University of Cambridge

November 1996

Key words: proof tools / generic reasoning / logic programming / logical variables / unification / Isabelle / higher-order logic / set theory / inductive definitions / cryptographic protocols

Summary. Proof tools must be well designed if they are to be more effective than pen and paper. Isabelle supports a range of formalisms, two of which are described (higher-order logic and set theory). Isabelle's representation of logic is influenced by logic programming: its "logical variables" can be used to implement step-wise refinement. Its automatic proof procedures are based on search primitives that are directly available to users. While emphasizing basic concepts, the article also discusses applications such as an approach to the analysis of security protocols.

To appear in the proceedings *Mathematical Methods in Program Development* (Summer School Marktoberdorf 1996), Springer-Verlag.

Table of Contents

1. What Should Proof Tools Do For Us?

Computer scientists often use pen and paper for proofs, as mathematicians have always done. Informal proofs leave big gaps, gaps that minds can bridge but machines often cannot. Proof tools require formal calculi, comprising a rigid syntax of formulas and rules for transforming them. Tools and formal calculi can be hard to use. They must give us something in return.

- *Soundness*: we can trust the result
- *Transparency*: we can follow the reasoning
- *Power*: the tool is mightier than the pen

There are trade-offs among these benefits. A tool that puts all the emphasis on soundness may sacrifice power, and vice versa. Transparency involves a combination of soundness (the reasoning is correct) and power (the reasoning is expressed at a high level). Even unsound tools can be valuable: consider floating-point arithmetic. If soundness is not guaranteed then we need transparency, in order to check for ourselves that a derivation is sound.

Soundness can be obtained by recording proofs down to the level of inference rules, and checking them with a separate tool. But this requires considerable storage, and does not aid transparency: detailed proofs are too big to understand. If the proof tool is allowed to invoke external reasoners, such as model checkers or computer algebra systems, then it could record all dependencies on such reasoners.

The tool must let us prove things that we cannot using pen and paper. This is the most important requirement of all, and is perhaps the hardest to attain. Tools are mainly valuable for proofs about objects, such as hardware or formal specifications, that are too big to manage on paper.

Marktoberdorf is traditionally devoted to pen-and-paper calculations. This year is an exception, with two lecture courses on automatic proof tools. Shankar presents some of the impressive applications of PVS. The present article complements his by discussing the principles underlying Isabelle [36].

Isabelle uses a unique representation of logic. The benefits include support for many formalisms and powerful tools for logical reasoning. "Logical variables" in goals serve as placeholders for as-yet-unknown subterms, to support refinement of specifications into implementations. Soundness comes from Milner's abstract type approach,[1] transparency from the use of high-level rules, and power through unification and search.

The article continues by covering generic reasoning (§2), extending it to quantifiers (§3). It then presents specialized tools: automatic proof (§4), mechanized set theory (§5), and induction (§6). We see how Isabelle has been applied to proofs about cryptographic protocols (§7). The conclusions (§8) briefly survey other applications, such as refinement of functional programs.

2. Generic Reasoning: The Propositional Case

A generic proof tool supports reasoning in a variety of logics. It provides independence from formalization details, a flexible treatment of notation, and ease of extension.

There are many formal systems that differ only slightly. Changing just a few rules can make a logic classical instead of constructive, higher-order instead of first-order. A proof tool should let us share the implementation effort for the common part of such logics.

[1] I have described his approach elsewhere [32, Chapter 7].

Users should not have to know which of the facts they use are axioms as opposed to theorems. They should not have to know which operators are primitive as opposed to derived. Proof tools should not make it harder to use non-primitive concepts. Minor changes to a formalization should not force us to redo existing proofs.

Good notation matters. The pen can draw any symbols and figures. Our tools cannot match that, but they should be built to be as flexible as possible. We must not dismiss this question as mere syntax.

Most of us do not switch between formal systems, but any proof development requires extending the formal system. Each definition may involve new notation, new laws to be proved and new reasoning methods for those laws. Over time, the effect is to create a new formal system.

Isabelle's approach to generic reasoning is based on a unique interpretation of inference rules. Combining rules in various ways yields different forms of reasoning on the level of theorems.

2.1 Two Readings of Proof Rules

Most approaches to formal logic take rules of the form

$$\frac{X_1 \quad \dots \quad X_m}{X}$$

as primitive. We call $X_1, \dots, X_m$ the *premises* and X the *conclusion*. There are two ways of reading such rules:

- *forward*: if $X_1, \dots, X_m$ then X
- *backward*: to show X it suffices to show $X_1, \dots, X_m$

The forward reading is sometimes used in logic texts, especially in proofs organized as a numbered list of formulas. Forward proof is useful when applying a general law to a specific case, and when simplifying the instance so obtained.

The backward reading is better for proof discovery. It concentrates on the given problem, analysing it to simpler subproblems.

Hand proofs consist of a mixture of backward and forward proof. Backward proof forms the main structure of the argument (such as induction followed by case analysis), while forward proof may be used at lower levels. A proof tool should support both styles.

2.1.1 Proof Trees. Here is a typical proof tree, for a simple theorem about sets: if $C \subseteq A$ and $z \in C \cap B$ then $z \in A$.

$$\dfrac{C \subseteq A \qquad \dfrac{z \in C \cap B}{z \in C}\,\cap\text{-elim1}}{z \in A}\,\subseteq\text{-elim}$$

Proof trees are constructed by composing rules. The tree above consists of two rules joined together. It simply assumes the formulas $C \subseteq A$ and $z \in C \cap B$, which perhaps we could prove given more information about A, C and z.

This example presupposes a proof system for set theory with rules such as these:

$$\frac{x \in A \cap B}{x \in A}\,\cap\text{-elim1} \qquad \frac{x \in A \cap B}{x \in B}\,\cap\text{-elim2} \qquad \frac{A \subseteq B \quad x \in A}{x \in B}\,\subseteq\text{-elim}$$

A common misconception is that proof trees can only use the rules for the standard connectives $\wedge$, $\vee$, etc. It is better to reason using rules that directly express the subject matter. The definition of $\cap$ in set theory is surprisingly complicated, and best forgotten as soon as possible.

In Isabelle, we can derive new rules and use them as if they were primitive rules. Proving X from the assumptions $X_1, \ldots, X_m$ derives the rule

$$\frac{X_1 \quad \ldots \quad X_m}{X.}$$

If $m = 0$ we might call this a theorem.

2.1.2 Forward versus Backward Proof. Proof trees can be built from the root upwards (backward reasoning) or from the leaves downwards (forward reasoning). Refer again to the proof tree shown above. In the forward style, we begin with the assumption $z \in C \cap B$. We apply rule $\cap$-elim1 to obtain $z \in C$, etc., finally concluding $z \in A$.

In the backward style, we begin with the desired conclusion, $z \in A$. Call this the *main goal*. We observe that rule $\subseteq$-elim can reduce it to $z \in C$, where we must prove $C \subseteq A$. We then apply rule $\cap$-elim1 to obtain the subgoal $z \in C \cap B$. At this point, we have reduced the main goal to the two subgoals $C \subseteq A$ and $z \in C \cap B$. This is the derived rule

$$\frac{C \subseteq A \quad z \in C \cap B}{z \in A.}$$

It is of no permanent interest. But it perfectly captures the state of the backward proof. If we know enough about A, C and z to prove the two subgoals then we shall eventually be left with a proof state, $z \in A$, with no subgoals. This proof state is the theorem we intended to prove.

Proof tools usually derive theorems. Perhaps they should instead derive rules. The operation of joining two rules would then implement both forward and backward proof. Isabelle is designed to operate on rules.

2.2 A Simple Rule Calculus

Let us replace the traditional two-dimensional notation for inference rules by $[\![X_1; \ldots ; X_m]\!] \Longrightarrow X$. The brackets $[\![\ ]\!]$ are optional if there is only one premise, and $[\![X_1; \ldots ; X_m]\!] \Longrightarrow X$ abbreviates

$$X_1 \Longrightarrow (\cdots X_m \Longrightarrow X \cdots).$$

Here are the rules $\cap$-elim1 and $\subseteq$-elim in the notation:

$$x \in A \cap B \Longrightarrow x \in A \qquad [\![A \subseteq B; x \in A]\!] \Longrightarrow x \in B$$

Isabelle implements a calculus of inference rules. Its most basic primitives are the *trivial rule* and *resolution*.

The trivial rule has the form $X \Longrightarrow X$. It supports our use of rules as representing proof states, serving the same role as zero does in arithmetic. At the very start of the backward proof, before we have applied any rules, there is one subgoal to be proved, namely the main goal itself. If the main goal is X then the initial proof state is $X \Longrightarrow X$.

Rules of the form $[\![X_1; \ldots ; X_m]\!] \Longrightarrow X$ are called Horn clauses in Prolog terminology. Resolution on Horn clauses involves matching the conclusion of one rule with a premise of another rule. In this example, the conclusion of $[\![X_1; X_2]\!] \Longrightarrow X$ matches Y_2.

$$\begin{aligned} &[\![X_1; X_2]\!] \Longrightarrow X \qquad + \qquad [\![Y_1; \underline{Y_2}; Y_3]\!] \Longrightarrow Y \\ &\qquad\qquad\qquad\qquad = \text{instance of } [\![Y_1; \underline{X_1; X_2}; Y_3]\!] \Longrightarrow Y \end{aligned}$$

Underlining indicates the affected parts. In general we *unify* the formulas X and Y_2, applying the unifying substitution to the result — hence the words "instance of" above.

In general, resolution unifies the conclusion of one rule $[\![X_1; \ldots ; X_m]\!] \Longrightarrow X$ with the ith premise of another, $[\![Y_1; \ldots ; Y_n]\!] \Longrightarrow Y$. In the resulting rule, Y_i is replaced by $X_1; \ldots ; X_m$.

Prolog resolution, extended slightly, is all we need to build proof trees. It automatically matches rules and axioms to the goal being proved. We can even allow some variables in the goal to be updated. Such variables stand for currently unknown parts of the goal. They let us extract information from proofs, say for interactive program derivation. They also make it easier to implement proof procedures for quantifiers.

2.3 Assumptions in Natural Deduction

Natural deduction, due to Gentzen, is based upon three principles.

1. Proof takes place within a varying context of assumptions.
2. Each connective is defined independently of the others.
3. Each connective is defined by *introduction* and *elimination* rules.

In the case of $\wedge$, the *introduction* rule describes how to infer $P \wedge Q$ while the *elimination* rules for $\wedge$ describe what to infer from $P \wedge Q$:

$$\frac{P \quad Q}{P \wedge Q}\ {\scriptstyle \wedge\text{-intr}} \qquad \frac{P \wedge Q}{P}\ {\scriptstyle \wedge\text{-elim1}} \quad \frac{P \wedge Q}{Q}\ {\scriptstyle \wedge\text{-elim2}}$$

The elimination rule for $\rightarrow$ says what to deduce from $P \rightarrow Q$. It is sometimes called Modus Ponens.

$$\frac{P \rightarrow Q \quad P}{Q}\ {\scriptstyle \rightarrow\text{-elim}}$$

The introduction rule for $\rightarrow$ is characteristic of natural deduction. It says that $P \rightarrow Q$ is proved by assuming P and deriving Q:

$$\frac{\begin{matrix}[P]\\ Q\end{matrix}}{P \rightarrow Q} \qquad (\rightarrow\text{-intr})$$

The key point is that rule $\rightarrow$-intr *discharges* its assumption: even if the proof of Q requires the assumption P, the conclusion $P \rightarrow Q$ does not depend upon that assumption. The notation $[P]$ indicates that any uses of the assumption P are discharged.

Here is a little example. Assuming P and Q, we may prove $P \wedge Q$ using rule $\wedge$-intr. By applying rule $\rightarrow$-intr, we discharge the assumption Q to obtain $Q \rightarrow (P \wedge Q)$, still assuming P. By applying rule $\rightarrow$-intr again, we discharge the assumption P and get $P \rightarrow (Q \rightarrow (P \wedge Q))$. The conclusion no longer depends upon any assumption, and is clearly a tautology. The proof is shown as follows:

$$\cfrac{\cfrac{\cfrac{[P] \quad [Q]}{P \wedge Q}\ {\scriptstyle \wedge\text{-intr}}}{Q \rightarrow (P \wedge Q)}\ {\scriptstyle \rightarrow\text{-intr}}}{P \rightarrow (Q \rightarrow (P \wedge Q))}\ {\scriptstyle \rightarrow\text{-intr}}$$

The introduction rules for $\vee$ are straightforward. The elimination rule says that to show some R from $P \vee Q$ there are two cases to consider, one assuming P and one assuming Q.

$$\frac{P}{P \vee Q}\ \vee\text{-intr1} \qquad \frac{Q}{P \vee Q}\ \vee\text{-intr2} \qquad \frac{P \vee Q \quad \begin{matrix}[P]\\ R\end{matrix} \quad \begin{matrix}[Q]\\ R\end{matrix}}{R}\ \vee\text{-elim}$$

Horn clauses can accommodate natural deduction and assumptions. We allow them to be nested, extend resolution to nested clauses, and introduce a notion of proof by assumption.

2.4 Generalized Horn Clauses

Natural deduction requires a simple generalization of our rule calculus. We can formalize the rule $\rightarrow$-intr as $(P \Longrightarrow Q) \Longrightarrow P \rightarrow Q$. Thus, we regard assumption discharge as the same sort of entailment as that from premises to conclusion. It is implication: not the implication of first-order logic, but implication at the meta-level. We can regard $\rightarrow$-intr as a rule whose premise is itself a rule, namely $\frac{P}{Q}$ [48].

We must augment resolution to allow for nesting of $\Longrightarrow$. Let us consider why. To prove $P \rightarrow (Q \rightarrow (P \wedge Q))$, resolution with $\rightarrow$-intr yields the subgoal $P \Longrightarrow Q \rightarrow (P \wedge Q)$; as expected, the step adds P to the assumptions. Now we need to apply $\rightarrow$-intr again, to add Q to the assumptions. But the subgoal has the form $\cdots \Longrightarrow \cdots \rightarrow \cdots$ instead of just $\cdots \rightarrow \cdots$. Lifting in resolution allows a rule to be applied in any context. Lifting the rule $[\![X_1; \ldots; X_m]\!] \Longrightarrow X$ over the assumption P transforms it to

$$[\![P \Longrightarrow X_1; \ldots; P \Longrightarrow X_m]\!] \Longrightarrow (P \Longrightarrow X).$$

Lifting and matching to our subgoal transforms $\rightarrow$-intr into the Horn clause

$$(P \Longrightarrow (Q \Longrightarrow P \wedge Q)) \Longrightarrow (P \Longrightarrow Q \rightarrow (P \wedge Q)).$$

Resolving this with the proof state replaces our subgoal by $P \Longrightarrow (Q \Longrightarrow P \wedge Q)$, which may be written more concisely as $[\![P; Q]\!] \Longrightarrow P \wedge Q$. Finally, we must apply the rule $\wedge$-intr. Lifting it over the assumptions P and Q, and matching it to the subgoal, transforms $\wedge$-intr into the Horn clause

$$[\![\,[\![P; Q]\!] \Longrightarrow P;\ [\![P; Q]\!] \Longrightarrow Q\,]\!] \Longrightarrow ([\![P; Q]\!] \Longrightarrow P \wedge Q).$$

Resolution using this yields a new proof state having two subgoals,

$$[\![P; Q]\!] \Longrightarrow P \quad \text{and} \quad [\![P; Q]\!] \Longrightarrow Q.$$

We may generalize our notion of trivial rule from $X \Longrightarrow X$ to include subgoals of the form above. Proof by assumption involves deleting a subgoal of the form $[\![X_1; \ldots; X_m]\!] \Longrightarrow X$ where X matches X_i for some i between 1 and m; the matching substitution must be applied to all other subgoals.

Resolution is normally combined with lifting. For example, the rule

$$[\![Y_1;\ H \Longrightarrow Y_2]\!] \Longrightarrow Y$$

may be seen as a proof state with two subgoals Y_1 and Y_2. The second subgoal has an assumption, H. Resolving the rule $[\![X_1;\ X_2]\!] \Longrightarrow X$ against this proof state replaces the second subgoal by two new ones, each with H as an assumption.

$$\begin{aligned}[\![X_1;\ X_2]\!] \Longrightarrow X \quad &+ \quad [\![Y_1;\ H \Longrightarrow Y_2]\!] \Longrightarrow Y \\ &= \text{instance of } [\![Y_1;\ H \Longrightarrow X_1;\ H \Longrightarrow X_2]\!] \Longrightarrow Y\end{aligned}$$

2.5 The Role of Resolution

Isabelle uses resolution to provide proof checking in the forward and backward styles. Unlike classical resolution theorem provers such as Otter [22], Isabelle does not seek contradictions, but derives rules in positive form. Horn clause resolution is a special case of the sophisticated forms of resolution used in Otter. On the other hand, Isabelle generalizes Horn clause resolution in unusual ways. It allows clauses having nested implication, and resolves them using lifting.

Isabelle matches rules and assumptions automatically. Asked to prove a subgoal by assumption, Isabelle searches for suitable assumptions: we do not have to specify one by number. Isabelle considers all matching assumptions, not just the first. Similarly, if we supply a list of rules to match against a subgoal, Isabelle considers all possible matches. We can apply known facts with minimal effort.

Exercise 2.1. Express the rules $\vee$-intr1, $\vee$-intr2 and $\vee$-elim as (possibly generalized) Horn clauses.

Exercise 2.2. Comment on these alternative introduction rules for $\wedge$ and $\vee$. Why are they correct? What are they good for?

$$\frac{P \quad \begin{matrix}[P] \\ Q\end{matrix}}{P \wedge Q} \qquad\qquad \frac{\begin{matrix}[\neg Q] \\ P\end{matrix}}{P \vee Q}$$

Exercise 2.3. Give a natural deduction proof of $P \to (Q \to P)$.

3. A General Approach to Quantifiers

Quantifiers require a radical generalization of resolution. Isabelle uses the typed λ-calculus to represent the syntax of terms, formulas and rules. Ordinary unification, such as Prolog uses, is unsuitable. Isabelle bases resolution on higher-order unification. Our approach handles not only the quantifiers $\forall$ and $\exists$, but other binding operators such as indexed summations, unions or Cartesian products. It supports sound inference, not just a means of expressing the syntax.

3.1 The Typed λ-Calculus

Here is a capsule review of the typed λ-calculus.[2] The notation $M[L/y]$ stands for the result of substituting the term L for all free occurrences of the variable y in a term M. We regard $((\lambda x.M)N)$ as equal to $M[N/x]$: a β-conversion substitutes the argument, N, into the abstraction's body, M. Bound variables in M are renamed to prevent clashes with variables in N.

The application MN has type τ if M has the function type $\sigma \to \tau$ and N has type σ. The abstraction $\lambda x.M$ has type $\sigma \to \tau$ if M has type τ given that x has type σ. The function type $[\sigma_1, \ldots, \sigma_k] \to \tau$ abbreviates $\sigma_1 \to \cdots (\sigma_k \to \tau) \cdots$. By the standard treatment of curried functions, if M has this type and $N_1, \ldots, N_k$ have types $\sigma_1, \ldots, \sigma_k$, respectively, then $MN_1 \ldots N_K$ has type τ.

Isabelle uses a polymorphic type system. For example, the identity function $\lambda x.x$ gets type $\alpha \to \alpha$, where α is a *type variable*. Type variables can be replaced by any types. The identity function may be regarded as having any type of the form $\tau \to \tau$. Each occurrence of it in an expression may have a different type.

[2] The easiest way to learn about the λ-calculus and polymorphism is to learn a programming language based on them, like Haskell or ML. My ML book includes material about the λ-calculus itself [38, Chapter 9].

The type prop (short for proposition) is built-in. It is the type of inference rules, which include theorems and axioms as special cases. Rules are sometimes called *meta-level theorems* because Isabelle provides an inference system for them: the *meta-logic*. The meta-logic represents other inference systems, the *object-logics*: HOL, ZF, etc.

3.2 Declaring Types and Connectives

To represent an object-logic in Isabelle we extend the meta-logic with types, constants, and axioms. For example, take the predicate calculus.

To represent predicate calculus syntax, we introduce the type bool for meanings of formulas. It is preferable to avoid identifying bool with prop, the type of rules. So we use the constant isTrue to convert one to the other; if A has type bool then isTrue A has type prop.

The logical constants True, False, And, Or, etc., are declared in the obvious manner. With Isabelle such declarations are made in a *theory file*:

```
types
  bool
consts
  isTrue           :: bool => prop
  True, False      :: bool
  And, Or, Implies :: [bool, bool] => bool
```

The theory file may also specify the constants as having special syntax (such as infix) and describe pretty printing. Let us ignore such matters for now; assume that And, Or, Implies represent the infix operators $\wedge$, $\vee$, $\rightarrow$ with the usual precedences, and associating to the right. Thus the formula $P \wedge Q \wedge R \rightarrow Q$ can be represented by the λ-term

$$\mathsf{Implies}(\mathsf{And}\, P\, (\mathsf{And}\, Q R))Q.$$

We may continue to use the conventional syntax, keeping this representation hidden underneath.

The theory file declares the conjunction rules as follows:

```
rules
  conjI        "[| P;  Q |] ==> P&Q"
  conjunct1    "P&Q ==> P"
  conjunct2    "P&Q ==> Q"
```

Strictly speaking, a rule such as $\wedge$-elim1 should be written

$$\mathsf{isTrue}(P \wedge Q) \Longrightarrow \mathsf{isTrue}(P).$$

Usually isTrue is left implicit, to avoid clutter:

$$P \wedge Q \Longrightarrow P.$$

The need for isTrue can be inferred from the context. Above, both $P \wedge Q$ and P must have type bool since conjunction has type $[\mathsf{bool}, \mathsf{bool}] \rightarrow \mathsf{bool}$. Both operands of $\Longrightarrow$ must have type prop.

3.3 Declaring Quantifiers

Variable-binding notations exist throughout mathematics. Consider the integral $\int_a^b f(x)dx$, where x is bound, and the product $\Pi_{k=0}^{n} p(k)$, where k is bound. The quantifiers $\forall$ and $\exists$ also bind variables. Isabelle uses λ-notation to represent all such operators uniformly.

To handle the standard quantifiers we could use a theory file containing the following declarations:

```
types
  ind
consts
  All :: (ind => bool) => bool              (binder "ALL " 10)
  Ex  :: (ind => bool) => bool              (binder "EX " 10)
```

Here ind is some arbitrary type of individuals. For now it does not matter what these individuals are — numbers, sets, etc.[3] The *binder* specification causes Isabelle to interpret `ALL` $x.P$ as $\mathsf{All}(\lambda x.P)$, and similarly for `EX`; the 10 concerns operator precedence. Let us use the usual $\forall$ and $\exists$ symbols whenever possible.

Isabelle hides the representation from the user. Let us examine the internal form of an example:

$$\forall x.\, Px \wedge Qx \quad \mapsto \quad \mathsf{All}(\lambda x.\, Px \wedge Qx) \quad \mapsto \quad \mathsf{All}(\lambda x.\, \mathsf{And}(Px)(Qx))$$

How do we define a new quantifier, say $\forall_{x \in A} P$ to mean $\forall x.\, x \in A \to P$? For the semantics, we simply define a new constant abbreviating the appropriate expression. For the variable-binding notation, we must add four more lines to the theory file; no programming is required.

3.4 Quantifier Rules Involving Substitution

Of the four natural deduction rules for the quantifiers, two of them involve substituting a term for a bound variable:

$$\frac{\forall x.P}{P[t/x]}\ \forall\text{-elim} \qquad \frac{P[t/x]}{\exists x.P}\ \exists\text{-intr}$$

Isabelle's typed λ-calculus handles the substitution in these rules:

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

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

In both of these, P has type $\mathsf{ind} \to \mathsf{bool}$ and stands for a formula with a hole. We may replace it by the λ-abstraction of any formula over x, say $\lambda x.\, Qx \wedge Rxy$. The corresponding instance of $\forall$-elim is then

$$(\forall x.(\lambda x.\, Qx \wedge Rxy)x) \Longrightarrow (\lambda x.\, Qx \wedge Rxy)t$$

or equivalently, by β-reduction,

$$(\forall x.\, Qx \wedge Rxy) \Longrightarrow Qt \wedge Rty.$$

We thus obtain all instances of the traditional rule.

Isabelle hides this machinery: we do not see the β-reductions . In backward proof, $\forall$-elim generalizes a formula of the form $P[t]$, yielding the subgoal $\forall x.\, P[x]$. There are usually countless ways of doing so; to constrain the choices, Isabelle lets you specify P or t.

A rule like $\forall$-elim is normally applied in the forward direction, mapping a theorem such as $\forall x\,.\,0 + x = x$ to the new theorem $0 + ?t = ?t$. Here $?t$ is an *unknown*: a variable that can be replaced automatically. In Prolog terminology, it is called a "logical variable".

Isabelle provides both free variables x, y, ..., and unknowns $?x$, $?y$, ... From a logical point of view they are all free variables. The difference between the two

[3] Isabelle actually uses polymorphic declarations. Using type classes, we can specify whether to allow quantifications over booleans and functions. If we allow them, we get higher-order logic; otherwise we get many-sorted first-order logic [29].

kinds of variables is pragmatic. Unknowns may be replaced during unification; free variables remain fixed. This article often omits the question marks to avoid clutter; they should be present in all the Isabelle rules shown.

The rule $\exists$-intr is represented in precisely the same way. In backward proof it replaces the goal $\exists x.\, Qx \wedge Rxy$ by the subgoal $Q\,?t \wedge R\,?t\,y$. We need not specify $?t$, but may leave it as an unknown. Then we can split the subgoal in two (by applying $\wedge$-intr). Proving $Q\,?t$ will probably replace $?t$ by something, say 3. The other subgoal will become $R\,3\,y$.

Thus we can strip quantifiers without specifying terms for the bound variables. During the proof, the variables may be filled in automatically, which is valuable in both interactive and automatic proofs.

3.5 Quantifier Rules Involving Parameters

The other two quantifier rules involve provisos expressed in English:

$$\frac{P}{\forall x.P}\ \forall\text{-intr} \qquad \frac{\exists x.P \quad \begin{matrix}[P]\\ Q\end{matrix}}{Q}\ \exists\text{-elim}$$

Rule $\forall$-intr holds provided x is not free in the assumptions, while $\exists$-intr holds provided x is not free in Q or the assumptions. Most quantifier provisos are typical of these. They ensure that the premise makes no assumptions about the value of x: it holds for all x. Isabelle expresses them using !!, its inbuilt notion of 'for all'. We formalize $\forall$-intr by

$$(!!x.\, \mathsf{isTrue}(P\,x)) \Longrightarrow \mathsf{isTrue}(\forall x.P\,x).$$

This means, 'if Px is true for all x, then $\forall x.Px$ is true'. Hiding isTrue simplifies it to

$$(!!x.\, P\,x) \Longrightarrow (\forall x.\, P\,x). \qquad (\forall\text{-intr})$$

Applying this rule in backwards proof creates a subgoal prefixed by !!; the bound variable is called a *parameter* or *eigenvariable*. We have discussed (§2.4) how rules are lifted over assumptions; they are analogously lifted over parameters.[4] A subgoal's parameters and assumptions form a context; all subgoals resulting from it will have the same context, or one derived from it.

We have defined the object-level universal quantifier ($\forall$) using !!. But we do not require meta-level counterparts of all the connectives of the object-logic. The existential quantifier rule can also be formalized using !!:

$$[\![\exists x.P\,x;\ !!x.P\,x \Longrightarrow Q]\!] \Longrightarrow Q \qquad (\exists\text{-elim})$$

For another example of !!, consider the rule of mathematical induction:

$$\frac{P(0) \quad \begin{matrix}[\,P(x)\,]\\ P(x+1)\end{matrix}}{P(n)}$$

There is the usual x not free … proviso. In Isabelle, the rule becomes

$$[\![P(0);\ !!x.\, P(x) \Longrightarrow P(x+1)]\!] \Longrightarrow P(n)$$

[4] The rule's premises and conclusion receive the additional quantification $!!x$. All variables in the rule are given x as an additional argument. Their types are changed accordingly [24, 33].

In higher-order logic (HOL), we can express induction using predicate variables:

$$P(0) \wedge (\forall x. P(x) \rightarrow P(x+1)) \rightarrow P(n).$$

Isabelle provides the meta-level connectives !! and $\Longrightarrow$ so that users are not forced to work in HOL. Isabelle's treatment of rules recognizes !! and $\Longrightarrow$ but not $\forall$ and $\rightarrow$.

3.6 Higher-Order Unification

Isabelle resolves rules by unifying typed λ-terms. This process is called *higher-order unification.* To handle β-conversion, it reduces $(\lambda x.t)u$ to $t[u/x]$: this is easy. But sometimes Isabelle must solve equations like

$$?f(t) \equiv g\, u_1 \ldots u_k.$$

This task involves making guesses for the unknown function $?f$. Isabelle uses a refinement of Huet's [14] search procedure. It solves equations by guessing the leading symbol of $?f$, simplifying, then recursively unifying the result.

In the general case, higher-order unification is undecidable. Fortunately, we can usually recognize the problematical cases beforehand: they involve *function unknowns.* Terms such as $?f\,?x\,?y$ and $?f(?g\,x)$ match anything in countless ways. There may be infinitely many unifiers, and the search need not terminate. Isabelle lets you specify some unknowns before attempting unification.

Some uses of function unknowns are harmless. The term $?f\,a$ matches $a + a$ in four ways. Isabelle generates them lazily. Solutions that use the function's argument appear first, as they are usually preferable:

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

Terms like $?f\,x\,y\,z$, where the arguments are distinct bound variables, cause no difficulties. They can match another term in at most one way. If the other term is $x + y \times z$ then the only unifier is

$$?f \equiv \lambda xyz.\, x + y \times z$$

The approach can be implemented without full higher-order unification. Pattern unification [28] is much easier to implement because it does not attempt to invent functions. The cost is just a little more user intervention in the problematic cases.

3.7 A Close Look at Quantifier Reasoning

To see how quantifier reasoning works, let us examine two tiny proofs in detail. Both involve stripping quantifiers from the initial goal. Because of the order of the quantifiers, one goal is provable and the other is not. (I have modified Isabelle's output by using special symbols, etc., to improve readability.)

3.7.1 The Good Proof. We start with the goal $\forall x.\, \exists y.\, x = y$.

```
goal thy "∀x. ∃y. x = y";
 Level 0
 ∀x. ∃y. x = y
  1. ∀x. ∃y. x = y
```

The initial proof state is an instance of the trivial rule: the main goal and the only subgoal are identical. For our first inference, we apply the rule $\forall$-intr.

```
by (resolve_tac [allI] 1);
 Level 1
 ∀x. ∃y. x = y
  1. !!x. ∃y. x = y
```

This yields the subgoal $\exists y.\, x = y$, where x is bound (by !!) in that subgoal. Remember, x is called a parameter. Next, we remove the existential quantifier by applying ∃-intr. We get a subgoal containing an unknown.

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

Lifting makes the unknowns depend on the subgoal's parameters. Our subgoal has the form $x = ?y1\, x$, where $?y1$ is a *function* unknown. The unknown $?y1\, x$ may be replaced by any term containing x. The term we require is x itself, with $?y1$ updated to $\lambda x.x$. The last step proves the goal by reflexivity.

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

3.7.2 The Bad Proof. We start with the invalid goal $\exists y.\, \forall x.\, x = y$.

```
goal thy "∃y. ∀x. x = y";
 Level 0
 ∃y. ∀x. x = y
  1. ∃y. ∀x. x = y
```

The first inference applies ∃-intr, yielding the subgoal $\forall x.\, x = ?y$. The unknown is just $?y$; it may not be updated to terms containing x, since x is a bound variable.

```
by (resolve_tac [exI] 1);
 Level 1
 ∃y. ∀x. x = y
  1. ∀x. x = ?y
```

Removing the universal quantifier, by ∀-intr, yields the subgoal $x = ?y$. Still x is bound, though by !! instead of ∀.

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

When we attempt to use reflexivity, unification fails.

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

Quantifier proofs frequently produce unknowns of function type, applied to bound variables. This assures correct treatment of alternating quantifiers. Informally, it is best to regard a term like $?y_1(x)$ as standing for any term, including terms containing occurrences of x. Isabelle takes care of the function unknowns automatically.

The workings of lifting and higher-order unification are complicated, but take place out of sight. The method can be proved correct [33]. It is reasonably fast: Isabelle can do hundreds of such steps per second. It works just as well for user-defined quantifiers like $\forall_{x\in A} Px$ or $\bigcup_{x\in A} Bx$.

Isabelle's original representation of rules was inspired by Schroeder-Heister's [48] 'rules of higher level' in natural deduction. The current approach is essentially identical to that of Felty and Miller [9, 10]. Pfenning [43] surveys other work on logical frameworks.

Exercise 3.1. Express this substitution rule, where P serves as a template for substitution, in Isabelle form:

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

Exercise 3.2. Suggest a representation of $\forall_{x \in A} P$ in the λ-calculus, where occurrences of x in P are bound.

Exercise 3.3. Give the Isabelle form of this bounded quantifier rule:

$$\frac{\begin{matrix}[x \in A] \\ P\end{matrix}}{\forall_{x \in A} P} \quad (x \text{ not free in assumptions other than } x \in A)$$

4. Generic, Automatic Proof

Tactics and tacticals form the basis for semi-automatic theorem proving in any Isabelle object-logic. The built-in tactics support proof checking using the rules you have declared. Complex tactics can be built using tacticals, which implement several search strategies.

Isabelle provides two tools that work at a higher level. The *classical reasoner* works with analytic rules, such as are found in natural deduction formalisms. The *simplifier* performs rewriting and can be extended to exploit the properties of new connectives.

4.1 Tactics, for Backward Proof

Tactics are the basic unit of backward proof. They can be expressed by combining primitive tactics, using operators called *tacticals*. Milner introduced these concepts in Edinburgh LCF. Regarding automatic proof as impossible, he sought to provide a language in which users could express proof methods, which might be variations on the built-in tactics, or complicated strategies. Although Milner had developed an elegant high-level language for this purpose (ML), he sought an even higher level language just for proof.

Isabelle tactics differ substantially from those of LCF, HOL, etc. A tactic looks at the entire proof state: at all outstanding subgoals instead of just one. While this sacrifices the proof's tree structure, it is essential in order to support unification, where proving one goal may affect others. An Isabelle tactic can update unknowns in other subgoals or the main goal. It can search for proofs of all remaining subgoals.

An Isabelle tactic is a function from a state to a sequence (lazy list) of next states. The sequence enumerates alternatives for backtracking and other kinds of searches [38, Chapter 5]. Because of all this, Isabelle's tactics and tacticals are considerably more powerful than HOL's. We shall discuss applications below — especially the classical reasoner, which is a general tool for quantifier reasoning.

4.2 Proof Checking Tactics

The primitive tactics provide proof checking: one rule at a time. Section 2. describes the concepts behind `assume_tac` and `resolve_tac`; other tactics are variations on those themes.

Tactic `assume_tac` proves a subgoal by assumption. Shouldn't Isabelle do such trivial steps without being told to? But proof by assumption is less trivial than it looks. Consider this proof state:

```
(P(a) & P(b) → P(?x)) & (P(b) & P(c) → P(?x))
 1. [| P(a); P(b) |] ⟹ P(?x)
 2. [| P(b); P(c) |] ⟹ P(?x)
```

Both subgoals have two proofs, but only one of the four combinations proves both goals simultaneously. If backtracking occurs, `assume_tac` searches for another matching assumption. Assumptions are not referred to by number.

Tactic `resolve_tac` applies rules, which may be primitive or user-derived, searching for those that match the subgoal. Backtracking searches for other matches and other rules.

Tactic `eresolve_tac` uses and then deletes an assumption. It is suitable when a rule might make an assumption obsolete. Its effect with the rule $\vee$-elim (given in §2.3) is to replace a subgoal of the form

$$\cdots P \vee Q \cdots \Longrightarrow X$$

by two new subgoals of the form $\cdots P \cdots \Longrightarrow X$ and $\cdots Q \cdots \Longrightarrow X$. Backtracking makes `eresolve_tac` search for another matching assumption.

Tactic `res_inst_tac` lets us partly instantiate a rule explicitly during refinement, specifying some unknowns. It is needed when applying rules like $\forall$-elim, whose conclusion is too general for automatic matching.

Resolution performs the most basic step in mathematics: appeal to a previous result. Many proof checkers make this difficult.

4.3 Tacticals: Control for Tactics

Proofs expressed entirely in terms of primitive tactics would be too long. Isabelle's *tacticals* `THEN`, `ORELSE`, etc., combine existing tactics to yield new ones, providing a rich control language for tactics. They achieve the desired behaviour by operating on lazy lists. Recall that a tactic maps a proof state to a lazy list of possible next states. If *tac*, *tac1*, etc., are tactics then so are the following:

tac1 `THEN` *tac2* returns all states reachable by applying *tac1* then *tac2*
tac1 `ORELSE` *tac2* tries *tac1*; if this fails, it uses *tac2*
tac1 `APPEND` *tac2* calls both *tac1* and *tac2*, appending their results
`DETERM` *tac* returns the first state reachable by applying *tac*
`REPEAT` *tac* returns all states reachable by applying *tac* as long as possible
`DEPTH_FIRST` *satp tac* returns all states satisfying *satp* reachable by applying *tac* in depth-first search

Explicit control of backtracking can help keep the search space small. Using `DETERM` prevents backtracking inside its argument. The difference between `ORELSE` and `APPEND` is that `ORELSE` forbids backtracking from its first argument to its second. It is the user's responsibility to ensure that the eliminated alternatives are not needed.

There are tacticals for several other search strategies: iterative deepening [16], best-first, etc. The argument *satp* is a boolean-valued function specifying what kind of state to search for, typically in terms of how many subgoals are left unsolved. Artificial Intelligence textbooks [47] discuss these strategies. Depth-first search is fastest but often gets stuck down an infinite path; iterative deepening is safe but much slower; best-first search can be fast, but must be guided by an accurate heuristic function.

Figure 4.1 illustrates some of the possibilities. Consider *tac*1 `THEN` *tac*2. Here *tac*1 returns a sequence of three possible next states. Given the first of these, *tac*2 returns two next states. Given the second of these, *tac*2 returns no next states; thus this possibility contributes nothing to the final sequence of outcomes. Given the

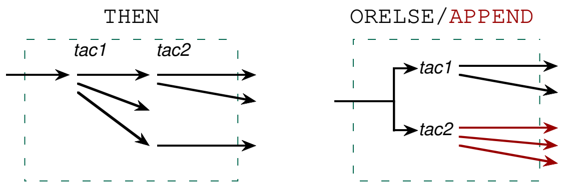

Fig. 4.1. The tacticals `THEN`, `ORELSE` and `APPEND`

third of these, *tac*2 returns one next state. A total of three states can arise from this call to *tac*1 `THEN` *tac*2.

Consider *tac*1 `ORELSE` *tac*2. Here *tac*1 returns a sequence of two possible next states. As this sequence is nonempty, it becomes the full output of *tac*1 `ORELSE` *tac*2; had it been empty, then the output would have been that of *tac*2. Similar is *tac*1 `APPEND` *tac*2, but its output comprises those of both *tac*1 and *tac*2.

Alternative versions of `THEN` and `APPEND` could interleave sequence elements instead of putting all possibilities from one sequence first. That would give more of a depth-first flavour.

Now let us consider a simple example of tacticals. Trying to prove that disjunction is associative has yielded three subgoals, each to prove $P \vee (Q \vee R)$ from the assumption P, Q or R:

```
(P ∨ Q) ∨ R → P ∨ Q ∨ R
 1. P ⟹ P ∨ Q ∨ R
 2. Q ⟹ P ∨ Q ∨ R
 3. R ⟹ P ∨ Q ∨ R
```

At our disposal are the disjunction rules $P \implies P \vee Q$ and $Q \implies P \vee Q$. We command Isabelle to perform a brute-force search:

```
by (DEPTH_SOLVE
    (assume_tac 1 ORELSE resolve_tac [disjI1, disjI2] 1));
 (P ∨ Q) ∨ R → P ∨ Q ∨ R
 No subgoals!
```

The command attempts proof by assumption and the disjunction rules, with backtracking. Tactical `DEPTH_SOLVE` uses `DEPTH_FIRST` to search for a fully solved proof state: no subgoals.

We often reach a point where the result clearly follows by repeated application of certain rules. We can then compose a command like the one above. We could have used `APPEND` instead of `ORELSE`, or used other search tacticals.

4.4 The Classical Reasoner

The classical reasoner is a package of tactics, such as `fast_tac` and `best_tac`, that prove goals automatically. They work by breaking up the formulas in the goal's conclusion and assumptions, and proving resulting trivial subgoals by assumption.

Here are some examples of what the classical reasoner can prove. We begin with #40 from Pelletier's problem set [42]. It is rather easy; its proof requires only 0.5 seconds on a fast SPARCstation.

$$(\exists y \forall x.\, Pxy \leftrightarrow Pxx) \rightarrow \neg\forall x \exists y \forall z.\, Pzy \leftrightarrow \neg Pzx$$

The classical reasoner can prove many set-theoretic identities. For this task, `fast_tac` uses rules proved specifically about the primitives of set theory, rather than expanding the definitions to primitive logical concepts. High-level rules promote transparency (short, clear proofs) as well as efficiency. This distributive law is proved in 0.3 seconds.

$$\Big(\bigcup_{i\in I} A_i \cup B_i\Big) = \Big(\bigcup_{i\in I} A_i\Big) \cup \Big(\bigcup_{i\in I} B_i\Big)$$

The third example comes from a proof of the soundness and completeness of propositional logic, by Tobias Nipkow and myself [37]. The relation $H \vdash p$ is defined inductively to express that proposition p is deducible from H. The deduction theorem is expressed as follows:

$$\{p\} \cup H \vdash q \Longrightarrow H \vdash \text{implies}\, p\, q$$

To prove it, the first step is induction on $\{p\} \cup H \vdash q$; this yields five subgoals. The second step is application of the classical reasoner, equipped with basic rules of the embedded logic. All five subgoals are proved in under 0.2 seconds.

Analogous but much harder is a proof of the Church-Rosser theorem for combinators. A key lemma is the diamond property for parallel reduction [39]. Again it consists of induction followed by classical reasoning. But this time `fast_tac` (equipped with rules about the behaviour of reductions) needs nearly 50 seconds to prove the four subgoals.

Now let us consider how the classical reasoner works. *Analytic* rules are those that break up a formula. We must distinguish between *safe* and *unsafe* rules. Safe rules do not require backtracking; they represent logical equivalences, and are analogous to rewrite rules such as $x \in A \cup B \leftrightarrow x \in A \vee x \in B$. Unsafe rules may require backtracking.

Safe rules lose no information; they may be attempted on any subgoal. For predicate calculus they include the following:

$$\frac{P \quad Q}{P \wedge Q} \qquad \frac{\begin{matrix}[P]\\ Q\end{matrix}}{P \to Q} \qquad \frac{P}{\forall x.P}$$

The following rules are unsafe because their premises are stronger than their conclusion. (They are sound, but in backward proof they discard information.) The latter rule is also unsafe in the operational sense that repeated application of it could run forever.

$$\frac{m < n}{m < n+1} \qquad \frac{x < y \quad y < z}{x < z}$$

The sequent calculus is better than natural deduction for reasoning about $\vee$ and $\exists$. The classical reasoner simulates the sequent calculus, representing the sequent $P_1, \ldots, P_m \vdash Q_1, \ldots, Q_n$ by

$$[\![P_1; \ldots; P_m; \neg Q_2; \ldots; \neg Q_n]\!] \Longrightarrow Q_1$$

The classical reasoner is coded in terms of tactics and tacticals. It is less efficient than hard-wired tautology checkers, but is more flexible. We may apply it in the predicate calculus, set theory, and in specialized theories. It is especially good at reasoning about inductively defined relations. When it fails, it often fails quickly, and can be single-stepped to locate the trouble spot.

4.5 A Generic Simplifier

Virtually all proof tools provide a simplifier that performs rewriting. Most of them handle conditional write rules such as $x \neq 0 \Longrightarrow x/x = 1$. Some of them even do permutative rewriting, to handle rules like $x + y = y + x$ without looping; this can be used to sort the terms in expressions of the form $e_1 + \cdots + e_n$. But few of them provide generic support to adapt rewriting to new connectives.

Consider the question of using context. For the occurrence of Q in $P \wedge Q$, we may assume that P is true. Simplification can take advantage of this. To rewrite $P \wedge Q$, first rewrite P, yielding say P', and then assume P' while rewriting Q. Here is a case where contextual information assists the simplifier:

$$f(1) = 2 \wedge x \neq 0 \wedge f(x/x) = 2.$$

The first conjunct yields a rewrite rule for f; the second allows conditional rewriting about division, so that the third conjunct can be simplified to true.

If information about context is built in, then users cannot extend it. Isabelle's simplifier is supplied such information as inference rules. To make it treat conjunction as described above, we give it the following rule:

$$\frac{P \leftrightarrow P' \quad \begin{matrix}[P'] \\ Q \leftrightarrow Q'\end{matrix}}{(P \wedge Q) \leftrightarrow (P' \wedge Q')}$$

We can prove an alternative rule that works from right to left, and dual rules for disjunction. If we define bounded quantification over sets, we can prove and install the analogous rule for it:

$$\frac{A = A' \quad \begin{matrix}[x \in A'] \\ P(x) \leftrightarrow P'(x)\end{matrix}}{(\forall_{x \in A} P(x)) \leftrightarrow (\forall_{x \in A'} P'(x))}$$

This says, rewrite $\forall_{x \in A} P(x)$ by first rewriting A to A' and then assuming $x \in A'$ when rewriting $P(x)$. Rules of this form are known as *congruence rules*.

The simplifier can perform automatic case splits and similar operations, also described by inference rules. The simplifier also lets users describe how to extract rewrite rules from general formulas. For example, a formula of the form $\forall_{x \in A} P(x)$ might yield rewrite rules conditional upon $x \in A$.

These extensions are sound because they work by building proofs. A user error cannot make the simplifer replace everything by true, or make the classical reasoner prove assertions that do not follow from the axioms.

5. Mechanized Set Theories

We need set theory whether or not we have types. Simple types (as found in the HOL system [12]) are too rigid: there is no type of n-element lists. Predicate subtyping (as in PVS) can help. But even in highly expressive type systems, one usually defines some sort of set theory. Set-theoretic primitives can express complex assertions, which is why they are found in specification languages such as Z and B.

Isabelle implements two set theories. Isabelle/ZF [34] is built upon first-order logic using the standard Zermelo-Fraenkel axioms [50]. Isabelle's higher-order logic (Isabelle/HOL) includes a polymorphically typed set theory, with sets represented by predicates. The two theories are similar but not identical; below we shall consider only ZF set theory.

Isabelle/ZF includes the standard operators such as $a \in A$, $A \subseteq B$ and $A \cup B$. Comprehensions are written $\{x \in A.\, P(x)\}$ or $\{b(x).\, x \in A\}$. Finite sets are written $\{a_1, \ldots, a_n\}$, tuples $\langle a_1, \ldots, a_n \rangle$ and functions $\lambda x \in A.\, b(x)$. There are disjoint sums $A + B$, Cartesian products $A \times B$ and function spaces $A \to B$. There are bounded quantifiers $\forall_{x \in A} P(x)$ and $\exists_{x \in A} P(x)$, indexed intersections $\bigcap_{x \in A} B(x)$ and unions $\bigcup_{x \in A} B(x)$, etc.

The definite description $\iota x. P(x)$ provides a means of naming an object that is characterized uniquely by P. It is like Hilbert's ϵ-operator, found in the HOL system, but does not assume the axiom of choice.

Note: Isabelle provides a choice of syntaxes for its underlying λ-calculus. In Isabelle/HOL, the application of term M to arguments $N_1, \ldots, N_k$ is written $M\, N_1\, N_k$; in Isabelle/ZF, it is written $M(N_1, \ldots, N_k)$. The former syntax has the advantage of (λ-calculus) tradition; it also facilitates partial application of curried functions, omitting arguments from the right. The latter syntax has the advantage of (set theory) tradition; it also forbids partial application, which does not make sense in first-order logic. The only ambiguity arises in terms of the form $M(a, b)$: in HOL there is one argument, an ordered pair; in ZF there are two arguments.

5.1 High-Level Rules

Proofs are conducted in terms of high-level rules, such as these for the subset relation:[5]

$$\frac{\begin{array}{c}[x \in A] \\ \vdots \\ x \in B\end{array}}{A \subseteq B}\ \subseteq\text{-intr} \qquad \frac{A \subseteq B \quad c \in A}{c \in B}\ \subseteq\text{-elim}$$

This is how the rules are expressed Isabelle:

$$(!!x.\, x \in A \Longrightarrow x \in B) \Longrightarrow A \subseteq B \qquad (\subseteq\text{-intr})$$

$$[\![A \subseteq B;\ a \in A]\!] \Longrightarrow a \in B \qquad (\subseteq\text{-elim})$$

They are proved from the Zermelo-Fraenkel axioms when Isabelle/ZF is built. In use, there is no perceptible difference between primitive and derived rules.

The $\{x \in A.\, P(x)\}$ notation deserves special mention. The axiom of separation is traditionally written $a \in \{x \in A.\, P(x)\} \leftrightarrow a \in A \wedge P(a)$. Strictly speaking this is an axiom scheme, with a distinct instance for each first-order predicate P. Here is the Isabelle version; for once, let us stop hiding the question marks:

$$?a \in \{x \in ?A\ .\ ?P(x)\} \leftrightarrow ?a \in ?A \wedge ?P(?a)$$

This formula behaves as a scheme, with $?P$ automatically replaced by the matching λ-expression and the result β-reduced. Derived rules, incidentally, are just a form of theorem scheme. We can derive convenient rules such as the following:

$$\frac{a \in A \quad P(a)}{a \in \{x \in A.P(x)\}} \qquad \frac{a \in \{x \in A.P(x)\}}{a \in A} \qquad \frac{a \in \{x \in A.P(x)\}}{P(a)}$$

5.2 A Proof about Unions

For a demonstration, let us prove a simple theorem in small steps using a tactic called `step_tac`. This tactic is normally called via `fast_tac`. Calling it directly allows for single-step reasoning and is especially useful when the proof fails.

Calling `step_tac` either applies safe inferences to the entire proof state, or applies one unsafe inference to the specified subgoal. Safe inferences are like rewriting by logical equivalences such as

[5] The first rule has a standard proviso on x: it must not occur free elsewhere in the inference.

$$
\begin{aligned}
x \in A \cap B &\iff x \in A \wedge x \in B \\
\langle x, y\rangle \in A \times B &\iff x \in A \wedge y \in B \\
\{x \in A.\, P(x)\} &\iff x \in A \wedge P(x) \\
x \in \bigcup(A) &\iff \exists y.\, x \in y \wedge y \in A
\end{aligned}
$$

Natural deduction rules express each direction of an equivalence separately.

The last one above concerns $\bigcup(A)$, the union of a set of sets, which might also be written $\bigcup_{X \in A} X$. Our sample theorem is $\bigcup(A \cap B) \subseteq \bigcup(A)$.

Isabelle's response to the initial `goal` command is to print an initial proof state, represented internally by an instance of the trivial clause $X \Longrightarrow X$.

```
goal ZF.thy "Union(A Int B) ⊆ Union(A)";
 Union(A Int B) ⊆ Union(A)
  1. Union(A Int B) ⊆ Union(A)
```

Calling `step_tac` performs the safe steps of assuming $x \in \bigcup(A \cap B)$, then $x \in y$ and $y \in A \cap B$, replacing the latter by $y \in A$ and $y \in B$, where x and y are arbitrary. In this context we must show $x \in \bigcup(A)$.

```
by (step_tac ZF_cs 1);
 Union(A Int B) ⊆ Union(A)
  1. !!x y. [| x∈y; y∈A; y∈B |] ⟹ x∈Union(A)
```

The next two `step_tac` calls apply unsafe inferences to subgoal 1. The first of these is the rule $\bigcup$-intr:

$$
[\![B \in C;\ A \in B]\!] \Longrightarrow A \in \bigcup(C)
$$

To show $x \in \bigcup(A)$ we can exhibit some set U such at $U \in A$ and $x \in U$.

```
by (step_tac ZF_cs 1);
 Union(A Int B) ⊆ Union(A)
  1. !!x y. [| x∈y; y∈A; y∈B |] ⟹ ?B3(x, y)∈A
  2. !!x y. [| x∈y; y∈A; y∈B |] ⟹ x∈?B3(x, y)
```

Isabelle displays the unknown set U as `?B3(x,y)`. It appears in two goals. In general, different proofs of a goal update its unknowns differently, and we may have to search for one that lets the other subgoals be proved.

The second unsafe step proves subgoal 1 by assumption; this amounts to guessing that `?B3(x,y)` is y. The remaining subgoal (now numbered 1) has changed to $x \in y$, and is trivially provable as there is an identical assumption.

```
by (step_tac ZF_cs 1);
 Union(A Int B) ⊆ Union(A)
  1. !!x y. [| x∈y; y∈A; y∈B |] ⟹ x∈y
```

Few examples are simple enough to present and hard enough to need more than one call to `step_tac`. More interesting and slightly harder to prove is the monotonicity rule for $\bigcup$:

$$
A \subseteq B \Longrightarrow \bigcup(A) \subseteq \bigcup(B)
$$

Its `fast_tac` proof takes about 0.2 seconds. Such rules, once proved, can themselves be used in later proofs as atomic inferences.

5.3 A Proof About Functions and Injections

Isabelle/ZF defines functions and the sets of functions and injections, essentially as follows:

$$\begin{aligned}
\mathsf{func}(f) &\equiv \forall x\, y.\ \langle x,y\rangle \in f \longrightarrow (\forall z.\ \langle x,z\rangle \in f \longrightarrow y = z)\\
A \to B &\equiv \{f \subseteq A \times B.\, A \subseteq \mathsf{dom}(f) \land \mathsf{func}(f)\}\\
\mathsf{inj}(A,B) &\equiv \{f \in A \to B.\ \forall w, x \in A.\ f`w = f`x \longrightarrow w = x\}
\end{aligned}$$

Here $f`x$ denotes application of the function f to the argument x. This object-level application differs from meta-level application, which is written $f(x)$. There is a real distinction between the two levels. With object-level application, the function is a set of pairs; with meta-level application, it is an Isabelle term of function type. Many meta-level functions, such as $\bigcup$, inj and range, are defined over the entire universe of sets; they are not sets themselves.

Because treating functions as sets of pairs is too low-level, most reasoning is done in terms of λ-abstraction and function application. Users need only know the definition of $\mathsf{inj}(A,B)$, the set of injections from A to B.

Injections, surjections and bijections are fundamental to discrete mathematics and computer science. Most databases label each record with a unique primary key, giving an injection from records to primary keys. The inverse of that injection is a function whose domain is the set of primary keys actually used. This fact is an application of our next theorem,

$$f \in \mathsf{inj}(A,B) \Longrightarrow f^{-1} \in \mathsf{range}(f) \to A.$$

Here is the initial proof state after expanding the definition[6] of $\mathsf{inj}(A,B)$.

```
goalw Perm.thy [inj_def]
    "!!f. f∈inj(A,B) ⟹ f⁻¹∈range(f)→A";
 !!f. f∈inj(A, B) ⟹ f⁻¹∈range(f) → A
  1. !!f. f ∈
          {f ∈ A → B .
           ∀ w∈A. ∀ x∈A. f ‘ w = f ‘ x ⟶ w = x}
          ⟹ f⁻¹∈range(f) → A
```

Unfolding the definitions of function space and $\mathsf{func}(f)$ yields a new proof state. Here, rewriting affects only the subgoal's conclusion, not its assumptions; an occurrence of the function space operator remains in the assumptions as $A \to B$.

```
by (asm_simp_tac (ZF_ss addsimps [Pi_iff, function_def]) 1  THEN
    eresolve_tac [CollectE] 1);
 !!f. f∈inj(A, B) ⟹ f⁻¹∈range(f) → A
  1. !!f. [| f ∈ A → B;
             ∀ w∈A. ∀ x∈A. f ‘ w = f ‘ x ⟶ w = x |]
          ⟹ (∀ x y. <y,x>∈f ⟶
                     (∀ z. <z,x>∈f ⟶ y = z)) &
              f⁻¹ ⊆ range(f) × A
```

The second part of the command applies the rule CollectE to break down an assumption of the form $\{f \in A \to B.\ \ldots\}$.

Default rules (stored in ZF_ss) perform most of the reasoning. Furthermore we specify Pi_iff and function_def as rewrite rules, to expose the representation of functions. Such low-level rules are not included by default.

Rewriting has proved part of the unfolded subgoal, namely $\mathsf{dom}(f^{-1}) \subseteq \mathsf{range}(f)$. We still have to show that f^{-1} is a function and is included in $\mathsf{range}(f) \times A$. The following command rewrites with apply_iff, which replaces $\langle a,b\rangle \in f$ by $f`a = b$ provided f is a function.

[6] via the goalw command

```
by (asm_simp_tac (ZF_ss addsimps [apply_iff]) 1);
 !!f. f∈inj(A, B) ⟹ f⁻¹ ∈ range(f) → A
  1. !!f. [| f ∈ A → B;
             ∀ w∈A. ∀ x∈A. f ` w = f ` x ⟶ w = x |]
          ⟹ (∀ x y.
                 y∈A & f ` y = x ⟶
                 (∀ z. z∈A & f ` z = x ⟶ y = z)) &
             f⁻¹ ⊆ range(f) × A
```

The equivalence between the quantified assumption and the first conjunct can now be discerned. We still have to prove it, as well as the second conjunct, $f^{-1} \subseteq \mathsf{range}(f) \times A$. Isabelle finds this task to be trivial.

```
by (fast_tac (ZF_cs addDs [fun_is_rel]) 1);
 !!f. f∈inj(A, B) ⟹ f⁻¹ ∈ range(f) → A
 No subgoals!
```

The rule `fun_is_rel` stands for $f \in A \to B \Longrightarrow f \subseteq A \times B$. Again, it concerns how functions are represented as sets of pairs. With the present subgoal, `fast_tac` needs only 1.3 seconds, of which one second is devoted to the first conjunct. This example illustrates how Isabelle's simplifier and classical reasoner can prove nontrivial theorems with a few commands.

Exercise 5.1. Using high-level rules such as those demonstrated above, prove the monotonicity of $\bigcup$, namely $A \subseteq B \Longrightarrow \bigcup(A) \subseteq \bigcup(B)$.

Exercise 5.2. Use the rule $[\![f \in \mathsf{inj}(A,B);\ b \in \mathsf{range}(f)]\!] \Longrightarrow f`(f^{-1}`b) = b$ to strengthen our result to $f \in \mathsf{inj}(A,B) \Longrightarrow f^{-1} \in \mathsf{inj}(\mathsf{range}(f), A)$. The conclusion could even be that f^{-1} is a bijection between $\mathsf{range}(f)$ and A.

6. Induction and Recursion

Most proof tools allow the definition of recursive types and functions. More unusually, Isabelle also supports inductive and co-inductive relations. Such relations are commonly found in theoretical computer science.

6.1 Recursive Types and Functions

A datatype is a disjoint sum and may be recursive. The syntax is based upon Standard ML's [38]. Datatypes can model lists, trees and finite enumerations.[7]

For example, here is a specification of binary trees. A binary tree may be a leaf (Lf) or a branch node (Br) carrying a label and two subtrees. Instead of specifying a fixed type of labels, we use polymorphism and specify the type variable `'a`.

```
datatype 'a bt = Lf  |  Br 'a ('a bt) ('a bt)
```

Datatype declarations may be placed in theory files along with declarations of other types, constants, etc. Isabelle makes appropriate definitions and derives the properties required of the datatype.

Isabelle (with ZF or HOL) supports function definition by well-founded recursion. Any relation that can be proved to be well-founded may be used to show termination of recursive calls. We may even interleave the proofs of termination and those of other correctness properties; this is essential for reasoning about certain nested recursive functions, such as the unification algorithm. Slind has recently

[7] Here we switch back from ZF to HOL. ZF has similar facilities (more general in fact) but HOL's type checking makes the declarations more concise.

written a tool to automate much of this process; its accepts function definitions expressed using pattern-matching [49].

Primitive recursion is a simpler form of function definition. It allows recursive calls only to immediate subparts of the argument. Here are two functions to count the branch nodes and leaves, respectively, of a binary tree.

```
consts nodes, leaves :: 'a bt => nat

primrec nodes bt
  nodes Lf           = 0
  nodes(Br a t1 t2)  = Suc(nodes t1 + nodes t2)

primrec leaves bt
  leaves Lf          = 1
  leaves(Br a t1 t2) = leaves t1 + leaves t2
```

Reasoning about recursive functions is often easy. A classic theorem states that each binary tree has one more leaf node than branch node. Let us set up the induction in Isabelle.

```
goal BT.thy "leaves(t) = Suc(nodes(t))";

by (bt.induct_tac "t" 1);
 leaves t = Suc (nodes t)
  1. leaves Lf = Suc (nodes Lf)
  2. !!a t1 t2.
        [| leaves t1 = Suc (nodes t1);
           leaves t2 = Suc (nodes t2) |]
        ==> leaves (Br a t1 t2) = Suc (nodes (Br a t1 t2))
```

The simplifier trivially proves the base case and inductive step shown above.

6.2 Inductive Definitions

An *inductive definition* specifies the least set closed under a given collection of rules [1]. The set of theorems in a logic is inductively defined. A structural operational semantics [13] inductively defines an evaluation relation on programs. Dually, a *coinductive definition* specifies the greatest set closed under given rules. Equivalence of concurrent processes is often defined coinductively, in terms of bisimulation relations [25].

Figure 6.1 gives an inductive definition of the "permutation of" relation for lists. Here $x\#l$ stands for the list with head x and tail l. The upper two rules say that [] is a permutation of itself and that exchanging the first two elements of a list creates a permutation. The lower left rule says that adding identical elements to both lists preserves the "permutation of" relation. The final rule says that the relation is transitive.

$$[] \leftrightsquigarrow [] \qquad\qquad y\#x\#l \leftrightsquigarrow x\#y\#l$$

$$\frac{xs \leftrightsquigarrow ys}{z\#xs \leftrightsquigarrow z\#ys} \qquad\qquad \frac{xs \leftrightsquigarrow ys \quad ys \leftrightsquigarrow zs}{xs \leftrightsquigarrow zs}$$

Fig. 6.1. Inductive Definition: Permutations of Lists

A desired collection of rules may be given to Isabelle (ZF and HOL) to specify a (co)inductive definition. Isabelle reduces it to a least fixedpoint (greatest fixedpoint for a coinductive definition). A broad class of definitions is acceptable [35].

Rule induction [52] is a powerful inference rule for proving consequences of $xs \leftrightsquigarrow ys$. Recall that $\leftrightsquigarrow$ is the least set closed under the rules given in Figure 6.1. If some predicate P is also closed under those rules then $xs \leftrightsquigarrow ys$ implies $P\,xs\,ys$ for all x and y. The corresponding subgoals are as follows:

$$\begin{aligned} & P\,[]\,[] \\ & P\,(y\#x\#l)\,(x\#y\#l) \\ \text{if } P\,xs\,ys \text{ then } & P\,(z\#xs)\,(z\#ys) \\ \text{if } P\,xs\,ys \text{ and } P\,ys\,zs \text{ then } & P\,xs\,zs \end{aligned}$$

Let us use rule induction to prove that $xs \leftrightsquigarrow ys$ implies $\mathsf{length}\,xs = \mathsf{length}\,ys$. The four subgoals are easily proved:

$\mathsf{length}[] = 0 = \mathsf{length}[]$
$\mathsf{length}\,(y\#x\#l) = 2 + \mathsf{length}\,l = \mathsf{length}\,(x\#y\#l)$
If $\mathsf{length}\,xs = \mathsf{length}\,ys$ then

$$\mathsf{length}(z\#xs) = 1 + \mathsf{length}\,xs = 1 + \mathsf{length}\,ys = \mathsf{length}(z\#ys)$$

If $\mathsf{length}\,xs = \mathsf{length}\,ys$ and $\mathsf{length}\,ys = \mathsf{length}\,zs$ then

$$\mathsf{length}\,xs = \mathsf{length}\,zs \quad \text{(by transitivity of equality)}$$

Symmetry of $\leftrightsquigarrow$ is another example. We can show that $xs \leftrightsquigarrow ys$ implies $ys \leftrightsquigarrow xs$ by simple rule induction: just put $ys \leftrightsquigarrow xs$ for $P\,xs\,ys$. Equivalently, observe that all the rules are preserved if we exchange the operands of $\leftrightsquigarrow$. The Isabelle versions of the declaration of $\leftrightsquigarrow$ and the proof of symmetry appear below.

6.3 Declaring Inductive Sets to Isabelle

This example illustrates the syntax for inductive definitions. Details have been simplified. The theory file declares the constant perm to stand for the relation. Three omitted lines inform Isabelle to use `xs <~~> ys` as notation for `(xs,ys):perm`.

```
consts  perm :: ('a list × 'a list) set

inductive perm
 Nil        []  ⇜⇝ []
 swap   y#x#l  ⇜⇝ x#y#l
 Cons      xs  ⇜⇝ ys ⟹ z#xs ⇜⇝ z#ys
 trans  [| xs  ⇜⇝ ys; ys ⇜⇝ zs |] ⟹ xs ⇜⇝ zs
```

If the recursion involves other operators then they must be monotonic; we must supply the corresponding monotonicity theorems.

(Co)inductive definitions are conservative. Isabelle derives basic properties of the newly defined set. These include the rules specified in the definition: the set's *introduction* rules. They include a rule for case analysis over the definition: the *elimination* rule. Finally, they include an induction or coinduction rule. These rules are proved, not merely asserted; for most definitions, this takes only seconds.

Let us prove that $\leftrightsquigarrow$ is symmetric. After stating the goal we apply rule induction, which is called `perm.induct`.

```
goal Perm.thy "!!xs. xs ⇜⇝ ys ⟹ ys ⇜⇝ xs";
by (etac perm.induct 1);
  1. !!xs. [] ⇜⇝ []
  2. !!xs l x y. x # y # l ⇜⇝ y # x # l
  3. !!xs ws ys z.
        [| ws ⇜⇝ ys; ys ⇜⇝ ws |] ⟹ z # ys ⇜⇝ z # ws
  4. !!xs ws ys zs.
        [| ws ⇜⇝ ys; ys ⇜⇝ zs;
           ys ⇜⇝ ws; zs ⇜⇝ ys |] ⟹ zs ⇜⇝ ws
```

We can prove all four subgoals trivially using a single command:

```
by (REPEAT (assume_tac 1 ORELSE resolve_tac perm.intrs 1));
```

This repeatedly attempts proof by assumption or one of the introduction rules (denoted by `perm.intrs`). We need only mention subgoal 1, as other subgoals are moved up when the first subgoal is proved.

6.4 Applications of (Co)Inductive Definitions

Several large studies use inductive definitions. Lötzbeyer et al. [18, 31] have related the operational and denotational semantics of Winskel's toy programming language IMP [52]. Using different techniques, Nipkow [30] and Rasmussen [45] have both proved the Church-Rosser theorem. A datatype specifies the set of λ-terms, while inductive definitions specify several reduction relations.

To demonstrate coinductive definitions, Frost [11] has proved the consistency of the dynamic and static semantics for a small functional language. The example, by Milner and Tofte [26], concerns a coinductively defined typing relation. Isabelle/ZF supports *codatatypes*, which are like datatypes but admit infinitely deep nesting. (Constructing non-well-founded trees in the presence of the foundation axiom requires variant pairs and function [40].) Frost defines a codatatype of values and value environments in mutual recursion. Non-well-founded values represent recursive functions; value environments are functions from variables into values.

The Ackermann's function proof [35] demonstrates the flexibility of inductive definitions in Isabelle. The set of primitive recursive functions is difficult to define formally — the composition operator combines a function with a list of functions. The "list of" operator is monotonic, however, and Isabelle allows monotonic operators to appear in inductive definitions.

Pusch [44] is proving the correctness of a compiling algorithm from Prolog to the Warren Abstract Machine (WAM). She uses datatypes to formalize Prolog's syntax and data structures involved in the interpretation, and inductive definitions to formalize the semantics of Prolog and the WAM. The proof involves around ten refinement steps from Prolog to the WAM; five of these steps have been verified using Isabelle. Each step introduces some low-level feature, such as pointers or optimizations of backtracking, and proves semantic equivalence.

Exercise 6.1. Using recursion and an if-then-else construct, define a function $\mathsf{count}\, l\, z$ to count how many times z occurs in the list l.

Exercise 6.2. Using rule induction, prove that $xs \leadsto ys$ implies $\mathsf{count}\, xs\, z = \mathsf{count}\, ys\, z$.

7. Reasoning about Cryptographic Protocols

Cryptographic protocols are designed to let agents communicate securely over an insecure network. An obvious security goal is *secrecy*: a spy cannot read the contents of messages intended for others. Also important is *authentication*: agents can identify themselves to others, but nobody can masquerade as somebody else. Specialist applications may require other security goals, such as anonymity.

Cryptographic protocols typically employ a *server*: a trusted agent whose job is to manage everyone's encryption keys, generate fresh keys and distribute them to the appropriate agents for use in new conversations.

Without breaking the encryption method, a spy can exploit flaws in a protocol. A typical attack involves intercepting a message sent by A and replaying it at some

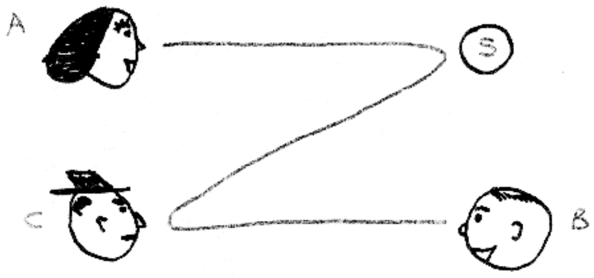

Fig. 7.1. A Private Conversation?

later time, hoping to be accepted as A. Protocols try to prevent replay attacks by using *nonces*, which may be random numbers or counters. By including a fresh nonce in a message, and checking for its presence in the acknowledgement, an agent can reject old messages replayed by the spy.

Many cryptographic protocols have been shown to be vulnerable, often in subtle ways [3]. Rigorous correctness proofs seem essential. One popular proof method is based upon logics of belief [7]. Security protocols can also be analyzed using Isabelle.

In preliminary experiments, I have proved properties of several well-known protocols, and discovered a new attack on one of them. Inductive definitions are used to specify the elements of messages and possible traces of protocol runs. Isabelle's simplifier and classical reasoner are used heavily.

7.1 Agents and Messages

The protocol proofs rest upon a common theory of agents, messages and their analysis. Protocols are treated at a high level, rather than as strings of bits. Encryption is regarded as a primitive; we cannot detect attacks that rely on numerical idiosyncrasies of encryption methods [27]. Such attacks can be prevented by including redundancy in the body of each encrypted message.

Agents include the server, the friendly agents and the spy. We can model attacks where the spy is an insider.

```
datatype agent = Server | Friend nat | Spy
```

Messages are agent names, nonces, keys, pairs and encryptions. Keys and nonces are just natural numbers. Long messages, consisting of pairs nested to the right, have the special syntax $\{\!|X_1, \ldots, X_n|\!\}$.

```
datatype msg = Agent agent
             | Nonce nat
             | Key   key
             | MPair msg msg
             | Crypt msg key
```

We use an operator invKey to model public-key encryption. Each public key K has an inverse, written K^{-1}, which should be kept private. In shared-key cryptography, a key is its own inverse.

```
rules invKey   "invKey (invKey K) = K"
```

Three inductively defined set operators model the processing of messages. If H is a set of messages then

- parts H is the set of all components of H (going down recursively). The key is not regarded as part of an encrypted message.

– $\mathsf{analz}\, H$ is similar, but needs the key K^{-1} to analyze the body of a message encrypted with K.
– $\mathsf{synth}\, H$ is the set of all messages that can be built using elements of H as components.

Here is the definition of $\mathsf{analz}\, H$. The set includes the whole of H as well as elements that can be obtained by taking pairs apart. It also contains the bodies of encrypted messages whose keys are available. The set $\mathsf{parts}\, H$ is defined similarly, except that it includes the bodies of all encrypted messages.

```
consts  analz   :: msg set => msg set
inductive "analz H"
  intrs
    Inj     "X ∈ H ⟹ X ∈ analz H"
    Fst     "{|X,Y|} ∈ analz H ⟹ X  ∈ analz H"
    Snd     "{|X,Y|} ∈ analz H ⟹ Y  ∈ analz H"
    Decrypt "[| Crypt X K  ∈ analz H; Key(invKey K) ∈ analz H |]
            ⟹ X  ∈ analz H"
```

The definition of $\mathsf{synth}\, H$ attempts to model the spy's capabilities. He cannot use nonces or keys except those in H: they are unguessable. But he can mention the name of any agent.

```
consts  synth   :: msg set => msg set
inductive "synth H"
  intrs
    Inj     "X ∈ H ⟹ X ∈ synth H"
    Agent   "Agent agt  ∈ synth H"
    MPair   "[| X ∈ synth H; Y ∈ synth H |] ⟹ {|X,Y|} ∈ synth H"
    Crypt   "[| X ∈ synth H; Key(K) ∈ H |] ⟹ Crypt X K  ∈ synth H"
```

7.2 An Algebraic Theory of Messages

More than eighty theorems are proved about parts, analz, synth and similar operators. They form a strong algebraic theory for reasoning about protocols. Thanks to Isabelle's automatic tools, the proof scripts are short: on average, each law is proved using under three commands. The proofs are mostly by rule induction, classical reasoning and rewriting.

Of the three set operators, parts is the easiest to reason about. It distributes over union, and can be evaluated symbolically. Atomic members of its argument (agents, keys and nonces) are simply extracted, becoming members of the result. Compound members are broken down and the new argument recursively evaluated. (Here $\mathsf{ins}\, X\, H$ denotes the set $\{X\} \cup H$.)

$$\begin{aligned}
\mathsf{parts}(\mathsf{parts}\, H) &= \mathsf{parts}\, H \\
\mathsf{parts}\, G \cup \mathsf{parts}\, H &= \mathsf{parts}(G \cup H) \\
\mathsf{parts}(\mathsf{ins}(\mathsf{Key}\, K)\, H) &= \mathsf{ins}(\mathsf{Key}\, K)(\mathsf{parts}\, H) \\
\mathsf{parts}(\mathsf{ins}\{\!|X, Y|\!\}H) &= \mathsf{ins}\{\!|X, Y|\!\}(\mathsf{parts}(\mathsf{ins}\, X\, (\mathsf{ins}\, Y\, H))) \\
\mathsf{parts}(\mathsf{ins}(\mathsf{Crypt}\, X\, K)H) &= \mathsf{ins}(\mathsf{Crypt}\, X\, K)(\mathsf{parts}(\mathsf{ins}\, X\, H))
\end{aligned}$$

On the other hand, analz is the hardest of the operators to reason about. During evaluation, we cannot extract a key without first showing that there are no encrypted messages that it could decrypt. Similarly, we cannot extract an encrypted message without knowing whether there is a key available to decrypt it. Here are some laws concerning analz, the last of which states that agent names can be extracted during evaluation.

$$\mathsf{analz}(\mathsf{analz}\, H) = \mathsf{analz}\, H$$

$$\mathsf{analz}\, G \cup \mathsf{analz}\, H \subseteq \mathsf{analz}(G \cup H)$$
$$\mathsf{analz}(\mathsf{ins}(\mathsf{Agent}\, a)\, H) = \mathsf{ins}(\mathsf{Agent}\, a)(\mathsf{analz}\, H)$$

Symbolic evaluation of $\mathsf{synth}\, H$ is impossible because the result is infinite. But other forms of reasoning are possible. Here are some properties of synth:

$$\mathsf{synth}(\mathsf{synth}\, H) = \mathsf{synth}\, H$$
$$\mathsf{synth}\, G \cup \mathsf{synth}\, H \subseteq \mathsf{synth}(G \cup H)$$
$$\mathsf{Key}\, K \in \mathsf{synth}\, H \Longrightarrow \mathsf{Key}\, K \in H$$

We have several laws to simplify expressions in which the operators are nested. All the operators are idempotent. Similarly we have

$$\mathsf{parts}(\mathsf{analz}\, H) = \mathsf{parts}\, H$$
$$\mathsf{analz}(\mathsf{parts}\, H) = \mathsf{parts}\, H.$$

The following laws are more interesting. The combination of building up followed by breaking down can be separated:

$$\mathsf{parts}(\mathsf{synth}\, H) = \mathsf{parts}\, H \cup \mathsf{synth}\, H$$
$$\mathsf{analz}(\mathsf{synth}\, H) = \mathsf{analz}\, H \cup \mathsf{synth}\, H$$

The proofs are largely automatic, but take a long time to run.

There is no law to break down the combinations $\mathsf{synth}(\mathsf{parts}\, H)$ and $\mathsf{synth}(\mathsf{analz}\, H)$. The latter is the set of messages that can be built from whatever can be decrypted from H. The spy might send any such message.

7.3 Specifying a Protocol

A protocol inductively specifies a set of possible traces. Each trace is a list of events. An *event* is something of the form $\mathsf{Says}\, A\, B\, X$, namely A says X to B. Other events could be envisaged, corresponding to internal actions of agents. We try to prove that all possible traces are safe. A typical safety guarantee states that receipt of a certain message implies that it was sent by some designated agent.

We assume that the spy already knows the keys of some agents, lost through carelessness. The protocol should still work for other agents: it must not simply collapse.

The function sees describes what an agent sees from a list of events. The spy sees all traffic; other agents see only what is intended for them. From the empty list, each agent sees his initial state, which contains only the key shared with the server. The spy holds all the "lost" keys.

Consider the Otway-Rees protocol, as simplified by Burrows et al. [7, page 247]. A protocol run starts with some agent A sending to B a message, requesting to start a secure conversation. Agent B forwards that message and further information to the server S. He replies by sending B a message containing a new key Kab and further information that B must forward to A.

1. $A \to B : Na, A, B, \{Na, A, B\}_{Ka}$
2. $B \to S : Na, A, B, \{Na, A, B\}_{Ka}, Nb, \{Na, A, B\}_{Kb}$
3. $S \to B : Na, \{Na, Kab\}_{Ka}, \{Nb, Kab\}_{Kb}$
4. $B \to A : Na, \{Na, Kab\}_{Ka}$

Figure 7.2 presents the inductive definition of this protocol. The first rule, `Nil`, allows the empty trace as the starting point. Rule `Fake` describes spy behaviour:

```
Nil  [] ∈ otway

Fake [| evs∈otway;  B≠Spy;  X∈synth (analz (sees lost Spy evs)) |]
     ⟹ Says Spy B X  # evs  ∈ otway

OR1  [| evs∈otway;  A ≠ B;  B ≠ Server |]
     ⟹ Says A B {|Nonce (newN evs), Agent A, Agent B,
                   Crypt {|Nonce (newN evs), Agent A, Agent B|}
                         (shrK A) |}
          # evs  ∈ otway

OR2  [| evs∈otway;  B ≠ Server;
        Says A' B {|Nonce NA, Agent A, Agent B, X|}∈set_of_list evs |]
     ⟹ Says B Server
             {|Nonce NA, Agent A, Agent B, X, Nonce (newN evs),
               Crypt {|Nonce NA, Agent A, Agent B|} (shrK B)|}
          # evs  ∈ otway

OR3  [| evs∈otway;  B ≠ Server;
        Says B' Server
             {|Nonce NA, Agent A, Agent B,
               Crypt {|Nonce NA, Agent A, Agent B|} (shrK A),
               Nonce NB,
               Crypt {|Nonce NA, Agent A, Agent B|} (shrK B)|}
          ∈set_of_list evs |]
     ⟹ Says Server B
             {|Nonce NA,
               Crypt {|Nonce NA, Key (newK evs)|} (shrK A),
               Crypt {|Nonce NB, Key (newK evs)|} (shrK B)|}
          # evs  ∈ otway

OR4  [| evs∈otway;  A ≠ B;
        Says S B {|Nonce NA, X, Crypt {|Nonce NB, Key K|} (shrK B)|}
          ∈set_of_list evs;
        Says B Server {|Nonce NA, Agent A, Agent B, X', Nonce NB, X''|}
          ∈set_of_list evs |]
     ⟹ Says B A {|Nonce NA, X|} # evs  ∈ otway

Oops [| evs∈otway;  B ≠ Spy;
        Says Server B {|Nonce NA, X, Crypt {|Nonce NB, Key K|} (shrK B)|}
          ∈set_of_list evs |]
     ⟹ Says B Spy {|Nonce NA, Nonce NB, Key K|} # evs  ∈ otway
```

Fig. 7.2. Specifying a Protocol

he may say anything he is able to. (He may also use the other rules, as if he were honest.) Rules `OR1` to `OR4` describe the messages of the protocol. Each is couched in terms of what message an agent receives and what message is sent in response. Finally, `Oops` models the accidental loss of a session key to the spy; the nonces identify the protocol run.

A spy might intercept messages. We do not need to model interception because nothing forces agents to respond to messages. There are traces in which some messages are ignored and others trigger several replies. Traces allow protocol runs to take place concurrently: middle-person attacks, where the spy uses messages from one run to break another, can be detected.

The protocol shown above turns out to be faulty. A spy C intercepts message 1 and starts a new protocol run with A. He ends up connected to A, to whom he masquerades as B. Here is the attack in detail:

1. $A \to B : Na, A, B, \{Na, A, B\}_{Ka}$ (intercepted)
1'. $C \to A : Nc, C, A, \{Nc, C, A\}_{Kc}$
2'. $A \to S : Nc, C, A, \{Nc, C, A\}_{Kc}, Na', \{Nc, C, A\}_{Ka}$ (intercepted)
2''. $C \to S : Nc, C, A, \{Nc, C, A\}_{Kc}, \underline{Na}, \{Nc, C, A\}_{Ka}$
3'. $S \to A : Nc, \{Nc, Kca\}_{Kc}, \{Na, Kca\}_{Ka}$ (intercepted)
4'. $C \to A : Na, \{Na, Kca\}_{Ka}$

The attack proceeds by replacing nonce Na' by A's original nonce Na (in message 2"), thereby fooling A into accepting key K_{ca} as a key for talking with B. This attack is more serious than that discovered by Mao and Boyd [20], where the server could detect that nonces are being misused.

Unaware of Mao and Boyd's attack, I attempted to prove the protocol correct. I could not prove a subgoal containing messages 1 and 2". In the original version of Otway-Rees, where nonce Nb is encrypted, one can prove that this situation is impossible. Thus, this method of analysing protocols lets us prove correctness properties and also detect flaws.

8. Other Work and Conclusions

Isabelle has been applied to a variety of other tasks: refinement, specification languages, theoretical studies. Let us consider a few of these. (Applications of inductive definitions were presented in §6.4.)

Isabelle's logical variables support interactive refinement, where a specification is transformed step-by-step into a proven implementation. Investigators at the Max Planck Institute, Saarbrücken have worked on deriving logic programs [2], functional programs [4] and hardware [5].

Coen [8] has implemented a variant of Manna and Waldinger's approach [19] to refinement of functional programs. His Classical Computational Logic (CCL) extends first-order logic with a functional language defined by an operational semantics. He derives programs in this language, not mathematical functions. Termination arguments are expressed using well-founded recursion, not primitive recursion. Later, unpublished work extended the approach to lazy functional programs, replacing termination by reduction to weak head normal form. Coen did some extended examples, deriving functional programs for insertion sort and unification.

Rasmussen has embedded the relational hardware description language Ruby using Isabelle's ZF set theory [46]. Two separate projects aim to support the Z specification language. Kolyang et al. [15] report a promising implementation of Z schemas. The TokiZ project [17] has built a prototype including a deductive system for Z and much of Z's mathematical library.

Isabelle has been applied to studies in logic. Basin et al. [6] are applying Isabelle to study labelled deductive systems. As a first example of modular presentation of logics, they have implemented a wide variety of modal logics. Matthews is using Isabelle to implement Feferman's theory of finitary inductive definitions, FS0 [21]. Grąbczewski has mechanized the first two chapters of Rubin and Rubin's *Equivalents of the Axiom of Choice* [41].

To conclude, let us recall those features of Isabelle that have turned out to be particularly successful. Designers of new tools should bear them in mind.

– A higher-order syntax supports variable binding.
– Unknowns in goals allow refinement and automatic proof search.
– A generic framework supports a wide range of notations and methods.
– The classical reasoner and simplifier let us construct proofs of realistic size.
– Set-theoretic primitives find many uses in specifications.
– Many computational phenomena can be modelled using inductive or coinductive definitions.

Acknowledgement. Giampaolo Bella, Rachel Cardell-Oliver, Michael Jones, Fabio Massacci, Chris Owens, Mark Staples and Myra VanInwegen commented on this material. Isabelle's simplifier is largely the work of Prof. Tobias Nipkow, now at the Technical University of Munich. Isabelle/ZF includes work by Martin Coen, Philippe de Groote and Philippe Noël. The research was funded by numerous grants including EPSRC GR/K57381 "Mechanising Temporal Reasoning" and GR/K77051 "Authentication Logics" and ESPRIT 6453 "Types." Many thanks are also due to the summer school organizers.

References

1. Aczel, P., An introduction to inductive definitions, In *Handbook of Mathematical Logic*, J. Barwise, Ed. North-Holland, 1977, pp. 739–782
2. Anderson, P., Basin, D., Deriving and applying logic program transformers, In *Algorithms, Concurrency and Knowledge (1995 Asian Computing Science Conference)* (Pathumthani, Thailand, December 1995), LNCS 1023, Springer, pp. 301–318
3. Anderson, R., Needham, R., Programming Satan's computer, In *Computer Science Today: Recent Trends and Developments*, J. van Leeuwen, Ed., LNCS 1000. Springer, 1995, pp. 426–440
4. Ayari, A., Basin, D., Generic system support for deductive program development, In *Second International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '96)* (1996), LNCS 1055, Springer, pp. 313–328
5. Basin, D., Friedrich, S., Modeling a hardware synthesis methodology in Isabelle, In von Wright et al. [51], pp. 33–50
6. Basin, D., Matthews, S., Viganò, L., Labelled propositional modal logics: theory and practice, Tech. Rep. MPI-I-96-2-002, Max-Planck-Institut für Informatik, Saarbrücken, 1996
7. Burrows, M., Abadi, M., Needham, R. M., A logic of authentication, *Proceedings of the Royal Society of London* **426** (1989), 233–271
8. Coen, M. D., *Interactive Program Derivation*, PhD thesis, University of Cambridge, Nov. 1992, Computer Laboratory Technical Report 272
9. Felty, A., Implementing tactics and tacticals in a higher-order logic programming language, *Journal of Automated Reasoning* **11**, 1 (1993), 43–82
10. Felty, A., Miller, D., Encoding a dependent-type λ-calculus in a logic programming language, In *10th International Conference on Automated Deduction* (1990), M. E. Stickel, Ed., LNAI 449, Springer, pp. 221–235
11. Frost, J., A case study of co-induction in Isabelle, Tech. Rep. 359, Computer Laboratory, University of Cambridge, Feb. 1995

12. Gordon, M. J. C., Melham, T. F., *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*, Cambridge University Press, 1993
13. Hennessy, M., *The Semantics of Programming Languages: An Elementary Introduction Using Structural Operational Semantics*, Wiley, 1990
14. Huet, G. P., A unification algorithm for typed λ-calculus, *Theoretical Comput. Sci.* **1** (1975), 27–57
15. Kolyang, Santen, T., Wolff, B., A structure preserving encoding of Z in Isabelle/HOL, In von Wright et al. [51], pp. 283–298
16. Korf, R. E., Depth-first iterative-deepening: an optimal admissible tree search, *Artificial Intelligence* **27** (1985), 97–109
17. Kraan, I., Baumann, P., Implementing Z in Isabelle, In *ZUM '95 : The Z Formal Specification Notation* (1995), J. P. Bowen M. G. Hinchey, Eds., LNCS 967, Springer
18. Lötzbeyer, H., Sandner, R., Proof of the equivalence of the operational and denotational semantics of IMP in Isabelle/ZF, Project report, Institut für Informatik, TU München, 1994
19. Manna, Z., Waldinger, R., Fundamantals of deductive program synthesis, *IEEE Trans. Softw. Eng.* **18**, 8 (Aug. 1992), 674–704
20. Mao, W., Boyd, C., Towards formal analysis of security protocols, In *Computer Security Foundations Workshop VI* (1993), IEEE Computer Society Press, pp. 147–158
21. Matthews, S., Implementing FS_0 in Isabelle: Adding structure at the metalevel, In *Design and Implementation of Symbolic Computation Systems. International Symposium, DISCO '96* (1996), J. Calmet C. Limongelli, Eds., LNCS 1128, Springer, pp. 228–239
22. McCune, W., OTTER 3.0 Reference Manual and Guide, Tech. Rep. ANL-94/6, Argonne National Laboratory, Argonne, IL, 1994
23. McRobbie, M., Slaney, J. K., Eds., *Automated Deduction — CADE-13 International Conference* (1996), LNAI 1104, Springer
24. Miller, D., Unification under a mixed prefix, *Journal of Symbolic Computation* **14**, 4 (1992), 321–358
25. Milner, R., *Communication and Concurrency*, Prentice-Hall, 1989
26. Milner, R., Tofte, M., Co-induction in relational semantics, *Theoretical Comput. Sci.* **87** (1991), 209–220
27. Moore, J. H., Protocol failures in cryptosystems, In *Contemporary Cryptology: The Science of Information Integrity*, G. J. Simmons, Ed. IEEE Press, 1992, pp. 541–558
28. Nipkow, T., Functional unification of higher-order patterns, In *Eighth Annual Symposium on Logic in Computer Science* (1993), M. Vardi, Ed., IEEE Computer Society Press, pp. 64–74
29. Nipkow, T., Order-sorted polymorphism in Isabelle, In *Logical Environments* (1993), G. Huet G. Plotkin, Eds., Cambridge University Press, pp. 164–188
30. Nipkow, T., More Church-Rosser proofs (in Isabelle/HOL), In McRobbie, Slaney [23], pp. 733–747
31. Nipkow, T., Winskel is (almost) right: Towards a mechanized semantics textbook, In *Foundations of Software Technology and Theoretical Computer Science* (1996), LNCS, Springer, In press
32. Paulson, L. C., *Logic and Computation: Interactive proof with Cambridge LCF*, Cambridge University Press, 1987
33. Paulson, L. C., The foundation of a generic theorem prover, *Journal of Automated Reasoning* **5**, 3 (1989), 363–397
34. Paulson, L. C., Set theory for verification: I. From foundations to functions, *Journal of Automated Reasoning* **11**, 3 (1993), 353–389
35. Paulson, L. C., A fixedpoint approach to implementing (co)inductive definitions, In *Automated Deduction — CADE-12 International Conference* (1994), A. Bundy, Ed., LNAI 814, Springer, pp. 148–161
36. Paulson, L. C., *Isabelle: A Generic Theorem Prover*, Springer, 1994, LNCS 828
37. Paulson, L. C., Set theory for verification: II. Induction and recursion, *Journal of Automated Reasoning* **15**, 2 (1995), 167–215
38. Paulson, L. C., *ML for the Working Programmer*, 2nd ed., Cambridge University Press, 1996
39. Paulson, L. C., Generic automatic proof tools, In *Automated Reasoning and its Applications*, R. Veroff, Ed. MIT Press, 1997, ch. 3, In press. Available as Report 396, Computer Laboratory, University of Cambridge
40. Paulson, L. C., A concrete final coalgebra theorem for ZF set theory, In *Types for Proofs and Programs: International Workshop TYPES '94* (published 1995), P. Dybjer, B. Nordström, J. Smith, Eds., LNCS 996, Springer, pp. 120–139

41. Paulson, L. C., Grąbczewski, K., Mechanizing set theory: Cardinal arithmetic and the axiom of choice, *Journal of Automated Reasoning* (1996), In press
42. Pelletier, F. J., Seventy-five problems for testing automatic theorem provers, *Journal of Automated Reasoning* **2** (1986), 191–216, Errata, JAR 4 (1988), 235–236
43. Pfenning, F., The practice of logical frameworks, In *Trees in Algebra and Programming - CAAP '96. 21st International Colloquium* (1996), H. Kirchner, Ed., LNCS 1059, Springer, pp. 119–134
44. Pusch, C., Verification of compiler correctness for the WAM, In von Wright et al. [51]
45. Rasmussen, O., The Church-Rosser theorem in Isabelle: A proof porting experiment, Tech. Rep. 364, Computer Laboratory, University of Cambridge, May 1995
46. Rasmussen, O., An embedding of Ruby in Isabelle, In McRobbie, Slaney [23], pp. 186–200
47. Rich, E., Knight, K., *Artificial Intelligence*, 2nd ed., McGraw-Hill, 1991
48. Schroeder-Heister, P., A natural extension of natural deduction, *Journal of Symbolic Logic* **49**, 4 (Dec. 1984), 1284–1300
49. Slind, K., Function definition in higher-order logic, In von Wright et al. [51]
50. Suppes, P., *Axiomatic Set Theory*, Dover, 1972
51. von Wright, J., Grundy, J., Harrison, J., Eds., *Theorem Proving in Higher Order Logics* (1996), LNCS 1125
52. Winskel, G., *The Formal Semantics of Programming Languages*, MIT Press, 1993