Tool Support for Logics of Programs

International Summer School Marktoberdorf
Mathematical Methods in Program Development

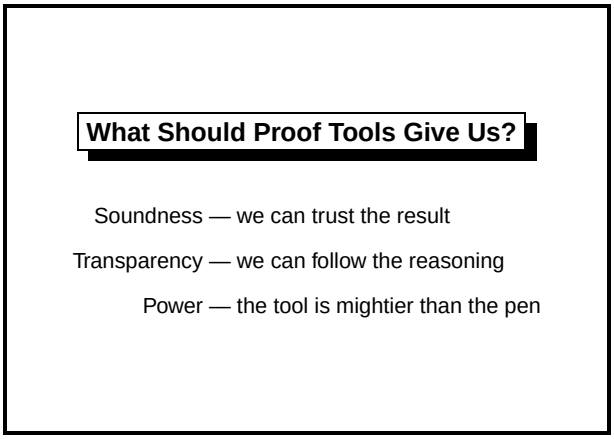
2002

Lawrence C Paulson
Computer Laboratory
University of Cambridge

lcp@cl.cam.ac.uk

Contents

Slide 101



Computer scientists often use pen and paper for proofs, as mathematicians have always done. Informal proofs leave big gaps, gaps that human minds can overleap but computers often cannot. Automatic proof tools require formal proof calculi, comprising a rigid syntax of formulæ and rules for transforming them. Both tools and formal calculi can be hard to use. They need to give us something in return, such as the benefits shown on the slide.

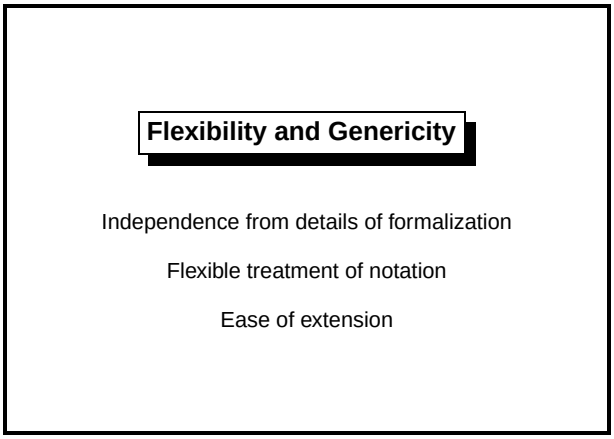
There are trade-offs among these benefits. A tool that puts all the emphasis on soundness may sacrifice power, and vice versa. Transparency involves a combination of soundness (the reasoning is correct) and power (the reasoning is expressed concisely, at a high level). Even unsound tools can be valuable: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

In developing Edinburgh LCF, Robin Milner exploited *abstract types* to achieve soundness without storing proofs. Several modern systems including HOL [11], use his technique. These lectures concern Isabelle [30], which uses related but not identical methods.

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, because detailed proofs are too big for people 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 oracles without storing proofs in full. Several modern systems store proofs in full, thereby suffering in efficiency. Others take a relaxed view of soundness, omitting simple checks for inconsistent definitions.

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 large objects such as hardware, or where we have chosen in advance to use a formal calculus. Tools have been built just to investigate the potential of a new calculus, but most tool users wish to solve problems.

Slide 102



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 well-designed proof tool should let us share the implementation effort for the common part of such logics. If the sharing cannot be realized explicitly, a uniform representation of inference rule allows proof procedures to be written for classes of logics, instead of individual logics.

Users should not have to know which of the properties they use are axioms as opposed to theorems. They should not have to know which operators are primitive as opposed to derived. With most formal systems the choice of primitives is arbitrary. Proof tools should not make it harder (or more expensive) to use non-primitive concepts. Minor changes to a formalization should not force us to discover new proofs for our theorems.

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 concrete 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. Making a host of definitions has the effect of creating a new formal system.

Slide 103

Two Readings of Proof Rules

There are two ways of using inference rules of the form

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

forward: if $X_1, \ldots, X_m$ then X

backward: to show X it suffices to show $X_1, \ldots, X_m$

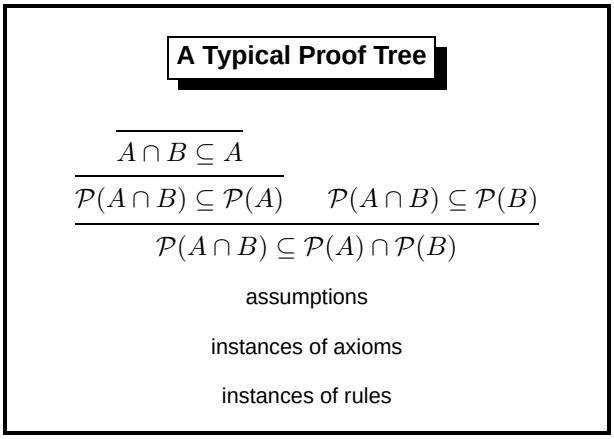
Nearly all approaches to formal logic take rules of the form above as primitive. We call $X_1, \ldots, X_m$ the *premises* and X the *conclusion*. In the simplest logics, the premises and conclusion are formulæ. They can be more complex objects, such as sequents (sequences of formulæ). In the *natural deduction* style, proofs of premises may be allowed to depend upon assumptions.

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

The backward reading is more useful in general; it concentrates on the given problem, analysing it to simpler subproblems. The backward reading can be seen in early work such as Gentzen's cut-elimination theorem of the 1930s.

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 reasoning may be used at lower levels. A proof tool should support both styles.

Slide 104



Proof trees are constructed by composing rules. This example presumes a proof system in which the subset properties of intersection are expressed as rules such as

$$\frac{C \subseteq A \qquad C \subseteq B}{C \subseteq A \cap B} \qquad (\cap\text{-greatest})$$

The root of the proof tree follows from this rule. And $\mathcal{P}(A \cap B) \subseteq \mathcal{P}(A)$ follows by another rule, which says that $\mathcal{P}$ is monotonic:

$$\frac{A \subseteq B}{\mathcal{P}(A) \subseteq \mathcal{P}(B)} \qquad (\mathcal{P}\text{-mono})$$

The occurrence of $\overline{A \cap B \subseteq A}$ means that $A \cap B \subseteq A$ is inferred by an axiom. Here the axiom has precisely this form; in general, we could infer any *instance* of the axiom: any formula $t \cap u \subseteq u$ for terms t and u. Strictly speaking, $A \cap B \subseteq A$ is an *axiom scheme*.

The occurrence of $\mathcal{P}(A \cap B) \subseteq \mathcal{P}(B)$ means this formula has simply been assumed. In this case the proof tree can be extended to prove the formula. In other cases, the derivation might be regarded as depending upon the assumed formula.

Proving X from the assumptions $X_1, \ldots, X_m$ derives the rule

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

If $m = 0$ then we might call this a *theorem* instead of a rule. The axioms and rules used in the proof above can be derived from the primitive axioms and rules of set theory. Derived rules may be used exactly like primitive rules, resulting in shorter, more abstract proofs.

Slide 105

Forward v Backward Proof

Theorem proved by forward proof:

$$\frac{\overline{A \cap B \subseteq A}}{\mathcal{P}(A \cap B) \subseteq \mathcal{P}(A)}$$

Rule derived by backward proof:

$$\frac{A \cap B \subseteq A \qquad \mathcal{P}(A \cap B) \subseteq \mathcal{P}(B)}{\mathcal{P}(A \cap B) \subseteq \mathcal{P}(A) \cap \mathcal{P}(B)}$$

Proof trees can be built from the root upwards (backward reasoning) or from the leaves downwards (forward reasoning). Forward proof typically proves theorems. Applying the rule $\mathcal{P}$-mono to the axiom $A \cap B \subseteq A$ yields the theorem $\mathcal{P}(A \cap B) \subseteq \mathcal{P}(A)$. Forward proof can also derive rules. Applying $\mathcal{P}$-mono to itself derives the rule

$$\frac{A \subseteq B}{\mathcal{P}(\mathcal{P}(A)) \subseteq \mathcal{P}(\mathcal{P}(B))}$$

In the backward style we begin with the desired conclusion, $\mathcal{P}(A \cap B) \subseteq \mathcal{P}(A) \cap \mathcal{P}(B)$. Call this the *main goal*. We observe that $\cap$-greatest can reduce it to $\mathcal{P}(A \cap B) \subseteq \mathcal{P}(A)$ and $\mathcal{P}(A \cap B) \subseteq \mathcal{P}(B)$. Turning to the first subgoal, we reduce it by $\mathcal{P}$-mono to $A \cap B \subseteq A$. At this point, we have reduced the main goal to two subgoals:

$$A \cap B \subseteq A \quad \text{and} \quad \mathcal{P}(A \cap B) \subseteq \mathcal{P}(B)$$

This is the derived rule shown on the slide. It is of no permanent interest. But it perfectly captures the state of the backward proof.

The axiom $A \cap B \subseteq A$ proves the first subgoal, erasing the first premise of the rule above. Applying $\mathcal{P}$-mono to the remaining premise reduces it to $A \cap B \subseteq B$. If we have an axiom of this form, then this premise too can be erased. We are left with a proof state that has no subgoals:

$$\mathcal{P}(A \cap B) \subseteq \mathcal{P}(A) \cap \mathcal{P}(B).$$

This derived rule *is* of permanent interest, as it 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.

Slide 106

A Simple Rule Calculus

Rules $[\![X_1; \ldots ; X_m]\!] \Longrightarrow X$ (Horn clauses)

Trivial rule $X \Longrightarrow X$

Resolution to join rules. Example: (X matches Y_2)

$$[\![X_1;\, X_2]\!] \Longrightarrow X \quad + \quad [\![Y_1;\, Y_2;\, Y_3]\!] \Longrightarrow Y$$
$$= \text{instance of } [\![Y_1;\, X_1;\, X_2;\, Y_3]\!] \Longrightarrow Y$$

We replace the traditional notation for inference rules by $[\![X_1; \ldots ; X_m]\!] \Longrightarrow X$. This suits the computer implementation (where the two-dimensional syntax is inconvenient). The brackets $[\![\ ]\!]$ are optional if there is only one premise, and $[\![X_1; \ldots ; X_m]\!] \Longrightarrow X$ is an abbreviation for $X_1 \Longrightarrow (\cdots (X_m \Longrightarrow X) \cdots)$. Here are two subset rules in the notation:

$$A \subseteq B \Longrightarrow \mathcal{P}(A) \subseteq \mathcal{P}(B) \qquad [\![C \subseteq A; C \subseteq B]\!] \Longrightarrow C \subseteq A \cap B$$

The trivial rule 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 $X \Longrightarrow X$. The backward proof of the previous slide starts with the trivial rule $\mathcal{P}(A \cap B) \subseteq \mathcal{P}(A) \cap \mathcal{P}(B) \Longrightarrow \mathcal{P}(A \cap B) \subseteq \mathcal{P}(A) \cap \mathcal{P}(B)$.

The most basic operation on rules is Horn clause *resolution*, which matches the conclusion of one rule with the ith premise of another rule:

$$[\![X_1; \ldots ; X_m]\!] \Longrightarrow X \quad + \quad [\![Y_1; \ldots ; Y_{i-1}; Y_i; Y_{i+1}; \ldots ; Y_n]\!] \Longrightarrow Y$$
$$= \text{instance of } [\![Y_1; \ldots ; Y_{i-1}; X_1; \ldots ; X_m; Y_{i+1}; \ldots ; Y_n]\!] \Longrightarrow Y$$

In the resulting rule, Y_i is replaced by $X_1; \ldots ; X_m$. In general we *unify* the formulæ X and Y_i, applying the unifying substitution to the result — hence the words "instance of" above. (The version on the slide is a special case with $m = 2$, $i = 2$ and $n = 3$.)

This is Horn clause resolution as found in Prolog. It is all we need to build proof trees. It automatically instantiates rules and axioms so that they match the goal being proved. We can even allow some variables in the goal to be instantiated. Such variables stand for *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.

Slide 107

Natural Deduction; Assumptions

$$\frac{\begin{array}{c}[P]\\ Q\end{array}}{P \to Q}\ (\to\text{-intr})$$

$$\frac{\dfrac{[P]}{Q \to P}}{P \to (Q \to P)}$$

Natural deduction, devised by Gerhard Gentzen, is based upon three principles. (1) Proof takes place within a varying context of assumptions. (2) Each logical connective is defined independently of the others (possible because (1) eliminates the need for tricky uses of implication). (3) Each connective is defined by *introduction* and *elimination* rules.

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

$$\frac{P \qquad Q}{P \wedge Q}\ (\wedge\text{-intr}) \qquad \frac{P \wedge Q}{P}\ (\wedge\text{-elim1}) \qquad \frac{P \wedge Q}{Q}\ (\wedge\text{-elim2})$$

The elimination rule for $\to$ says what to deduce from $P \to Q$. It is just Modus Ponens:

$$\frac{P \to Q \qquad P}{Q}\ (\to\text{-elim})$$

The introduction rule for $\to$ says that $P \to Q$ is proved by assuming P and deriving Q. The key point is that rule ($\to$-intr) *discharges* its assumption: the assumption P is used to prove $Q \to P$, but is no longer available once we conclude $P \to (Q \to P)$. The notation $[P]$ indicates that any uses of the assumption P are discharged.

Assumptions are used as one might expect, though their scope is sometimes unclear. In the proof on the slide, assumption P is used to prove $Q \to P$ and then discharged in the proof of $P \to (Q \to P)$. We could have assumed Q but did not need to.

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.

$$\frac{P}{P \vee Q}\ (\vee\text{-intr1}) \qquad \frac{Q}{P \vee Q}\ (\vee\text{-intr2}) \qquad \frac{P \vee Q \qquad \begin{array}{c}[P]\\ R\end{array} \qquad \begin{array}{c}[Q]\\ R\end{array}}{R}\ (\vee\text{-elim})$$

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.

Slide 108

Generalized Horn Clauses

Formalize $\to$-intr as $(P \Longrightarrow Q) \Longrightarrow P \to Q$

Resolution, *lifting* over assumption H. Example:

$$[\![X_1;\, X_2]\!] \Longrightarrow X \quad + \quad [\![Y_1;\, H \Longrightarrow Y_2]\!] \Longrightarrow Y$$

$$= \text{instance of } [\![Y_1;\, H \Longrightarrow X_1;\, H \Longrightarrow X_2]\!] \Longrightarrow Y$$

Formalizing $\to$-intr as $(P \Longrightarrow Q) \Longrightarrow P \to Q$ means we regard assumption discharge as the same sort of entailment as that from premises to conclusion. Put another way, $\to$-intr takes a premise that is itself a rule, namely $\frac{P}{Q}$. This idea can be made formal [40].

We must augment resolution to allow for nesting of $\Longrightarrow$. Let us consider why. To prove $P \to (Q \to P)$, resolution with $\to$-intr yields the subgoal $P \Longrightarrow Q \to P$; as expected, the step adds P to the assumptions. Now we need to apply $\to$-intr again, to add Q to the assumptions. But the subgoal has the form $\cdots \Longrightarrow \cdots \to \cdots$ instead of just $\cdots \to \cdots$. 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 $\to$-intr into the Horn clause

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

This replaces our subgoal by $P \Longrightarrow (Q \Longrightarrow P)$, which may be written more concisely as $[\![P; Q]\!] \Longrightarrow P$. 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.

The resolution shown above is an instance of the following schema, where X matches Y_i:

$$[\![X_1; \ldots ; X_m]\!] \Longrightarrow X \qquad + \qquad [\![Y_1;\ \ldots\ ;\ Y_{i-1};\ [\![\vec{H}]\!] \Longrightarrow Y_i;\ Y_{i+1};\ \ldots\ ;\ Y_n]\!] \Longrightarrow Y$$

$$= \text{instance of } [\![Y_1;\ \ldots\ ;\ Y_{i-1};\ [\![\vec{H}]\!] \Longrightarrow X_1; \ldots ; [\![\vec{H}]\!] \Longrightarrow X_m;\ Y_{i+1};\ \ldots\ ;\ Y_n]\!] \Longrightarrow Y$$

This augments Horn clause resolution to lift of the rule into the subgoal's context; here $\vec{H} \equiv [\![H_1;\ \ldots\ ;\ H_k]\!]$.

Slide 109

Summary

Tools must be trustworthy, powerful and flexible.

They should support **forward** and **backward** styles.

Rules can be represented as **nested Horn clauses**.

Proofs can be built using Horn clause **resolution**.

Rules and assumptions can be matched automatically.

Please do not confuse Isabelle with classical resolution theorem provers such as Otter [20]. Isabelle does not employ proof by refutation, but instead 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.

The next lecture will discuss more radical generalizations of resolution. Isabelle uses the typed λ-calculus as its syntactic framework, and therefore bases resolution on higher-order unification.

As a matter of policy, Isabelle instantiates both rules and assumptions automatically. Asked to prove a subgoal by assumption, Isabelle searches assumptions that match; we do not have to specify one by number. Isabelle considers all matching assumptions, not just the first one. The same holds for rules, if Isabelle is provided with a list of rules to match against a subgoal. The aim is to allow the easy use of known facts.

Exercise 1 Starting with the right instance of the trivial rule, write down the Horn clause resolution steps corresponding to the backward proof of $\mathcal{P}(A \cap B) \subseteq \mathcal{P}(A) \cap \mathcal{P}(B)$.

Exercise 2 Give the Isabelle forms of the rules $\vee$-intr1, $\vee$-intr2 and $\vee$-elim.

Exercise 3 Comment on these alternative introduction rules for $\wedge$ and $\vee$. Why are they correct? What are they good for?

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

Slide 201

Typed λ-Calculus

$a, b, c, \ldots$	constants
$x, y, z, \ldots$	variables
$\lambda x.M$	abstraction over x
$M\ N$	application of M to N

Types prop, α, $\sigma \to \tau$, $[\sigma_1, \ldots, \sigma_k] \to \tau$

Isabelle uses the typed λ-calculus to represent the syntax of terms, formulæ and rules.

Here is a capsule review of the typed λ-calculus. Capital letters like L, M, N, ... stand for terms. In $(\lambda x.M)$, we call x the *bound variable* and M the *body*. Every occurrence of x in M is *bound* by the abstraction. An occurrence of a variable is *free* if it is not bound by some enclosing abstraction. For example, x occurs bound and y occurs free in $(\lambda z.(\lambda x.(yx))$.

Let $M[L/y]$ be the result of substituting L for all free occurrences of y in M. The β-conversion $((\lambda x.M)N) \to_\beta M[N/x]$ substitutes the argument, N, into the abstraction's body, M. We shall assume that bound variables in M are renamed to prevent clashes with free variables of N.

This λ-calculus is typed. 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, ... , N_k have types σ_1, ... , σ_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*. In use, 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. These instances are worked out automatically, with no user intervention.

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.

Polymorphism involves some complications. Type variables need to be constrained, using a mechanism similar to Haskell's type classes [14, 26]. In typed logic, this supports overloading, say to use $+$, $-$ and $\times$ over a range of arithmetic types. But Isabelle does not use polymorphism in its treatment of ZF set theory.

Slide 202

Declaring Types and Connectives

$$\begin{aligned} &\textbf{type}\ \text{bool} \\ \text{isTrue} &: \text{bool} \to \text{prop} \\ \text{True}, \text{False} &: \text{bool} \\ \text{And}, \text{Or}, \text{Implies} &: [\text{bool}, \text{bool}] \to \text{bool} \end{aligned}$$

$$\text{isTrue}(P \wedge Q) \Longrightarrow \text{isTrue}(P)$$

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 formulæ. 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*, which 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 $\wedge$, $\vee$, $\to$ with the usual precedences, and associating to the right. Thus the formula $P \wedge Q \wedge R \to Q$ can be represented by the λ-term

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

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

Common to all theories is that of the meta-logic itself, which provides constants for meta-level connectives such as $\Longrightarrow$, and special syntax such as $[\![X_1; \ldots ; X_m]\!] \Longrightarrow X$.

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

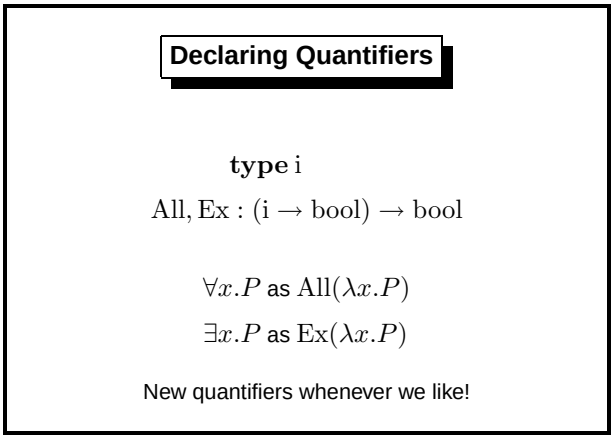
$$\text{isTrue}(P \wedge Q) \Longrightarrow \text{isTrue}(P)$$

but most people prefer to leave isTrue implicit:

$$P \wedge Q \Longrightarrow P$$

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 $[\text{bool}, \text{bool}] \to \text{bool}$.

Slide 203



Notations involving free and bound variables 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.

Here i is some arbitrary type of individuals. For now it does not matter what these individuals are — numbers, sets, etc. Isabelle actually uses polymorphic declarations:

$$\mathrm{All}, \mathrm{Ex} : (\alpha \to \mathrm{bool}) \to \mathrm{bool}$$

Here α ranges over some but not all types. Precisely which types depends upon type classes, which we are not going to discuss here. But the choice determines whether we allow quantifications over booleans and functions. If we allow them, we get higher-order logic; otherwise we get many-sorted first-order logic [23].

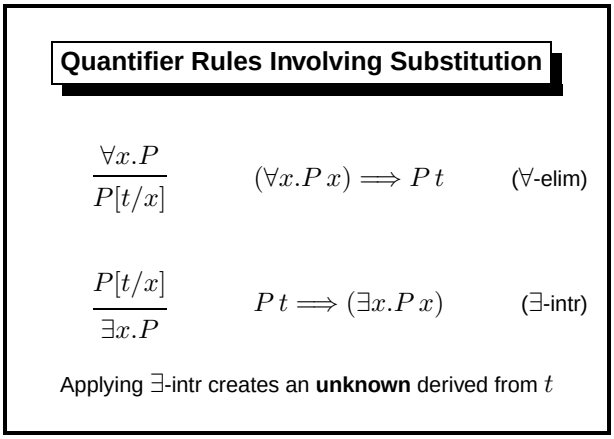
Write $\forall x.P$ for $\mathrm{All}(\lambda x.P)$ and $\exists x.P$ for $\mathrm{Ex}(\lambda x.P)$. As with $\wedge$, $\vee$, $\to$, Isabelle hides the representation from the user. Example:

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

The original formula might conventionally be written $\forall x.\, P[x] \wedge Q[x]$. Here P and Q are really formulæ that have a hole, and $P[t]$ is the result of putting t into the hole — or substituting t for x in $P[x]$. Isabelle represents such formulæ by function variables, with types like $\mathrm{i} \to \mathrm{bool}$, and uses β-conversion to perform substitutions.

To define a new quantifier such as $\forall x \in A.\, P$, meaning $\forall x.\, x \in A \to P$, we must declare a new constant, specify its notation, and make a logical definition like the one above. No programming is required, and only four lines in a theory file.

Slide 204



The typed λ-calculus handles the substitution in these rules. All Isabelle rules are schematic: we may instantiate their variables to generate new rules. In both the rules shown above, P has type i $\to$ bool and stands for a formula with a hole. We may replace it by the 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 can instantiate and simplify such rules automatically. Users do not need to be aware that β-reductions are occurring. 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 a variable that can be instantiated; this theorem is schematic.

In Isabelle, such schematic variables are normally written $?t$ for emphasis. These notes often omit question marks, to avoid clutter; should be present on all the free variables in Isabelle rules discussed here.

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 leave it as an unknown. Then we can split the subgoal in two (by applying $\wedge$-intr). Proving $Q\,?t$ will probably instantiate $?t$ to something, say 3. The other subgoal will become $R\,3\,y$.

Thus we can strip quantifiers without specifying how to instantiate the bound variables. During the proof, the variables may get instantiated automatically. This is a boon in interactive proof; it is also the foundation of automatic proof procedures for quantifiers.

Slide 205

Quantifier Rules Involving Parameters

$$\frac{P}{\forall x.P} \quad (x \text{ not free in assumptions})$$

Premise: 'P is true for all x'

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

Applying the rule creates a ***parameter***

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

The proviso aims to ensure that the premise may not make assumptions about the value of x, and therefore holds for all x. We formalize $\forall$-intr by

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

This means, 'if Px is true for all x, then $\forall x.Px$ is true'. Hiding isTrue, we get the axiom shown on the slide.

Applying this rule in backwards proof creates a subgoal prefixed by !!, binding a parameter. Resolution must be able to lift rules over parameters, as well as over assumptions. 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! Consider the existential quantifier rule

$$\frac{\exists x.P \qquad \begin{array}{c}[P]\\ Q\end{array}}{Q} \quad x \text{ not free in } Q \text{ or assumptions} \qquad (\exists\text{-elim})$$

The Isabelle version is $[\![\exists x.P\, x;\ !!x.P\, x \Longrightarrow Q]\!] \Longrightarrow Q$.

Slide 206

Induction Rules and Parameters

$$\frac{P(0) \qquad \begin{array}{c}[\,P(x)\,]\\ P(x+1)\end{array}}{P(n)} \quad (x \text{ not free } \ldots)$$

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

Like the HOL formula

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

This is another example of the use of !!. In higher-order logic (HOL) one can quantify over formulæ and predicates, and therefore can express induction as shown above. We could also have used !! at the outer level, to close the rule over its free variables:

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

This, again, is analogous to the closed HOL formula

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

Isabelle provides the meta-level connectives !! and $\Longrightarrow$ so that users are not forced to work in HOL. Sometimes HOL is too strong, as in this version of set theory's separation axiom:

$$\forall P\ x.\ x \in \{x \in A.\, P(x)\} \leftrightarrow x \in A \wedge P(x)$$

This would make P range over higher-order predicates. Isabelle can express the axiom in first-order logic, which is normal usage:

$$!!P\ x.\ x \in \{x \in A.\, P(x)\} \leftrightarrow x \in A \wedge P(x)$$

Because Isabelle provides !! and $\Longrightarrow$, we can add this axiom or the induction rule to weak logics, such as equational or quantifier-free logic.

Isabelle's treatment of rules recognizes !! and $\Longrightarrow$ but not $\forall$ and $\rightarrow$. Thus it behaves properly even if the induction formula itself contains $\rightarrow$ and $\forall$.

Slide 207

Higher-Order Unification

To join rules, Isabelle must solve equations like

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

Isabelle makes guesses like

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

Isabelle resolves rules by **unifying** λ-terms. Unification is equation solving; the solution of $f(?x, c) \stackrel{?}{\equiv} f(d, ?y)$ is $?x \equiv d$ and $?y \equiv c$.

Higher-order unification is equation solving for typed λ-terms. To handle β-conversion, it must reduce $(\lambda x.t)u$ to $t[u/x]$. But it must guess the unknown function $?f$ in order to solve equations such as that shown on the slide. Huet's [15] search procedure solves equations by guessing the leading symbol of $?f$, simplifying, then recursively unifying the result.

Isabelle's unification procedure is polymorphic: it solves for type unknowns as well as for term unknowns. But it is incomplete, never guessing that a type unknown may be a function type. (Huet's procedure is complete, as it does not handle type unknowns.) A more serious problem is that higher-order unification is undecidable; there may be infinitely many unifiers, and the search need not terminate.

Such problems seldom arise in practice. But do be careful with **function unknowns**. Terms such as $?f\,?x\,?y$ and $?f(?g\,x)$ match anything in countless ways. They obviously should be avoided. Isabelle lets you instantiate unknowns before attempting unification.

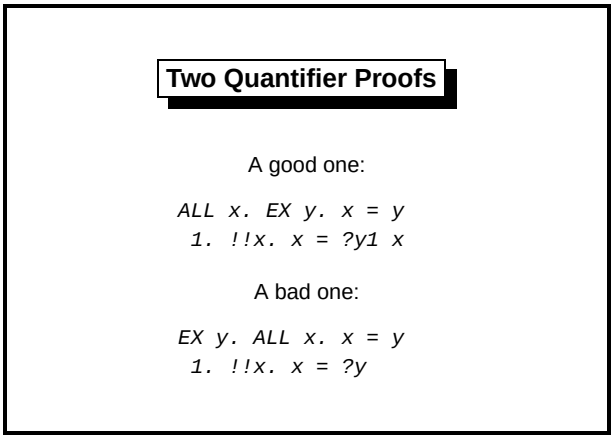
The term $?f\,a$ matches $a + a$ in four ways. Isabelle generates them lazily. Solutions that use the function's argument appear first, as they are usually preferable:

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

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

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

Slide 208



These tiny examples provide a glimpse of Isabelle's syntax. Both proofs involve stripping two quantifiers from an initial goal. Due to the order of the quantifiers, one goal is provable and the other is not. We see how Isabelle handles quantifiers in backward proof.

In the good proof, we start with the goal $\forall x.\, \exists y.\, x = y$. The first inference applies $\forall$-intr. This yields the subgoal $\exists y.\, x = y$, where x is bound (by !!) in that subgoal.

Next we should remove the existential quantifier, to get a subgoal containing unknown. Applying $\exists$-intr does this. But lifting has the effect of making 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 instantiated to any term containing x. The term we require is x itself, with $?y1$ instantiated to $\lambda x.x$. Here is the last step, which proves the goal by reflexivity.

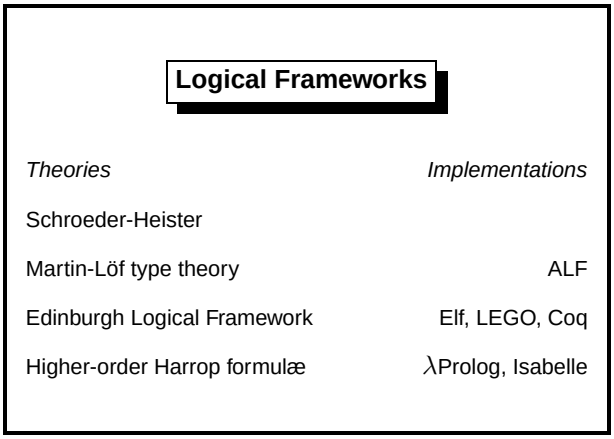
```
by (resolve_tac [refl] 1);
 Level 3
 ALL x. EX y. x = y
 No subgoals!
```

In the bad proof, we start with the invalid goal $\exists y.\, \forall x.\, x = y$. The first inference applies $\exists$-intr, yielding the subgoal $\forall x.\, x = ?y$. The unknown is just $?y$; it may not be instantiated to terms containing x, as x is a bound variable.

Removing the universal quantifier, by $\forall$-intr, yields the subgoal $x = ?y$. Still x is bound, though by !! instead of $\forall$. If we attempt to use reflexivity, unification will fail.

The workings of lifting and higher-order unification are complicated, but we do not need to know about them. We see ordinary formulæ, and instantiations happen automatically. The method can be proved correct [27]. It is reasonably fast: Isabelle can do hundreds of such steps per second. It works however deeply quantifiers are nested. And it works just as well for user-defined quantifiers like $\forall x \in A.\, Px$ or $\bigcup x \in A.\, Bx$. This is the foundation for Isabelle's automatic proof procedures for quantification.

Slide 209



Logical frameworks are theories of proofs in general logics. Best known is the Edinburgh Logical Framework [12], which is based on unpublished work by Per Martin-Lf. The proof assistants LEGO [18] and Coq [7] implement related type theories, but to formalize mathematics directly, rather than to formalize logics.

Isabelle's representation of rules was originally naive, partly inspired by Schroeder-Heister's [40] 'rules of higher level' in natural deduction. The current version borrows ideas from type-theoretic logical frameworks. It is essentially the higher-order hereditary Harrop (hohh) formulæ of Felty and Miller, who have exhibited [9] a translation between it and the Edinburgh Logical Framework.

λProlog is a logic programming language based on higher-order unification and hohh. Felty [8] has used it to implement theorem provers. This work is closely related to Isabelle. Elf [36] is an analogous language based on the Edinburgh Logical Framework.

The most visible difference between hohh and type-theoretic frameworks is that the latter represent propositions as types. Instead of having the assumption P you have the bound variable $x \in P$. The additional bound variables make it impractical to handle quantifier reasoning as Isabelle does. Type-theoretic frameworks require a more complicated form of higher-order unification, and their models are hard to visualize.

The orthodox viewpoint is that the λ-calculus terms are just syntactic codes of logical formulæ. Isabelle's approach lends itself to a semantic viewpoint. Hohh is a fragment of higher-order logic, which (ignoring polymorphism) has simple models based on ordinary set theory. We may regard bool as denoting the two-element set of truth values. Connectives $\wedge$, $\vee$, $\rightarrow$ denote the obvious functions over truth values. Even All and Ex, which represent quantification, are just infinitary truth tables. The Isabelle formulations of the usual rules can be seen to be true under this semantics.

Slide 210

Summary

- λ-abstraction expresses **variable binding**.
- Application expresses **substitution**; β-reduction performs it.
- **Meta-quantification** expresses quantifier provisos.
- **Unknowns** arise in quantifier proofs.

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 kinds of variables is pragmatic. Unknowns may be instantiated during unification; free variables remain unchanged. Having two kinds of variables allows us to distinguish, when stating a goal, which parts of it should be regarded as fixed.

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.

Exercise 4 Express this substitution rule, where P serves as a template for substitution, in Isabelle form:

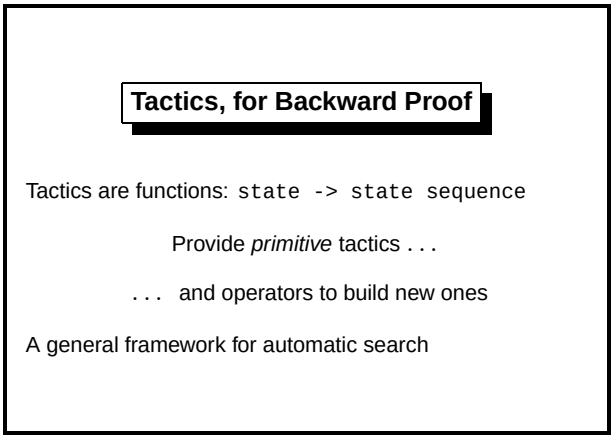
$$\frac{t = u \qquad P[t/x]}{P[u/x]}$$

Exercise 5 Suggest a possible representation of $\forall x \in A.\, P$ in the λ-calculus, where occurrences of x in P are bound.

Exercise 6 Express this rule for a bounded universal quantifier in Isabelle form:

$$\frac{\begin{array}{c}[x \in A] \\ P\end{array}}{\forall x \in A.\, P} \quad (x \text{ not free in assumptions other than } x \in A)$$

Slide 301

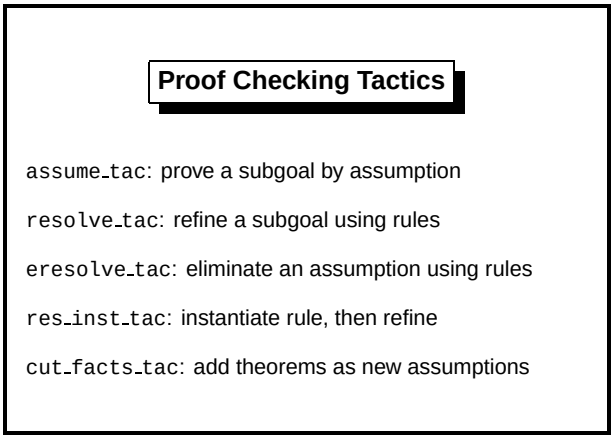


Tactics are the basic unit of backward proof. They can be expressed by combining primitive tactics, using operators called *tacticals*. Robin Milner introduced these concepts in Edinburgh LCF. Regarding automatic proof as impossible, he intended instead to provide a convenient language in which users could express their proof methods, which might be variations on the built-in tactics, or heavy-duty 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 from those of LCF and its successor HOL in many respects. Isabelle represents proof states by theorems of the form "subgoals imply goal", so there is no need to reconstruct the actual proof at the end of the top-down development (in HOL terminology, there are no validations). An Isabelle tactic looks at the entire proof state: at all outstanding subgoals instead of just one. This is inherent in the representation of proof states by theorems. While it sacrifices the tree structure of the tactical proof, something like this is essential in order to support unification, where proving one goal may affect others. An Isabelle tactic can instantiate unknowns in other subgoals or the main goal; it can search for proofs of all remaining subgoals.

Isabelle tactics return sequences (lazy lists) of next states, enumerating alternatives for backtracking. Operations on sequences can express many kinds of searches. (See my ML book [33], chapters 5 and 10.) 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.

Slide 302



These provide proof checking at the lowest level. Proofs expressed entirely in terms of these would be extremely lengthy. Sometimes they are used in this form, but mostly they are invoked via higher-level strategies. The first lecture described the concepts behind `assume_tac` and `resolve_tac`. The others are essentially variations on those themes; there are quite a few of them.

`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. Discrimination nets are available to speed up the search, if there are hundreds of rules.

`eresolve_tac` is suitable when a rule applies to an assumption and makes it obsolete. The tactic uses and then deletes the assumption. Its effect with $\vee$-elim is to search for an assumption of the form $P \vee Q$ and create two new subgoals with assumptions P and Q respectively. Backtracking makes `eresolve_tac` search for another matching assumption.

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

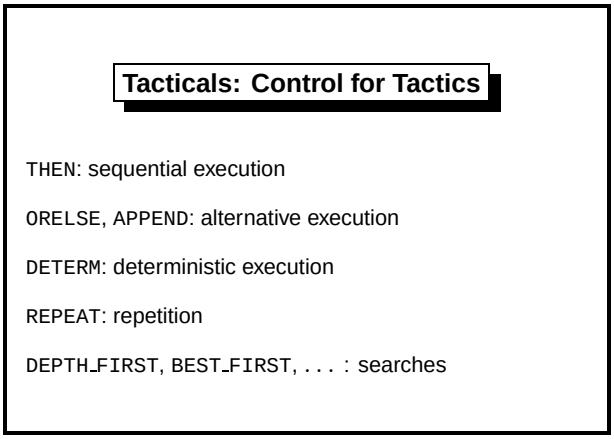
As for `assume_tac`: shouldn't Isabelle do proof by assumption automatically? One can arrange for tactics of any sort to be applied after any proof command. But proof by assumption is less trivial than it looks. Consider this proof state:

```
Level 1
(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.

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

Slide 303



Isabelle's tacticals provide a rich control language for tacticals. They express new tactics that can combined further. They achieve the desired behaviour by operating on lazy lists:

tac1 `THEN` *tac2* returns all states reachable by applying *tac1* then *tac2*

tac1 `ORELSE` *tac2* tries *tac1*; if this fails, uses *tac2*

tac1 `APPEND` *tac2* calls both *tac1* and *tac2*, appending their results

`DETERM` *tac* returns the first state reachable by applying *tac*

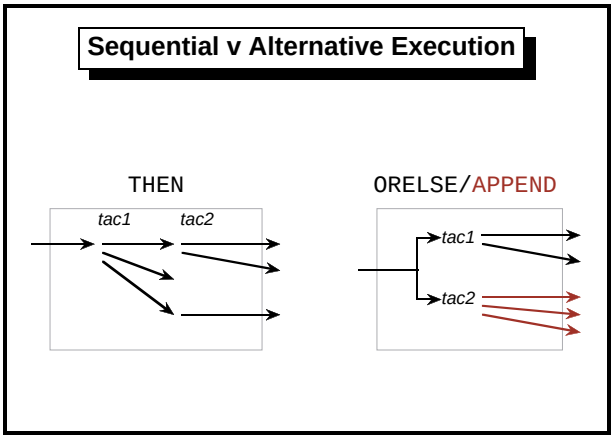
`REPEAT` *tac* returns all states reachable by applying *tac* as long as possible

`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; it is appropriate when the choice is between operations that have the same effect when applied in any order. The difference between `ORELSE` and `APPEND` is that `ORELSE` forbids backtracking from its first argument to its second; it is appropriate when the second argument specifies operations to be tried only as a last resort.

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. Artificial Intelligence textbooks [39] 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.

Slide 304



Consider *tac*1 `THEN` *tac*2. Here *tac*1 returns a sequence of three possible next states.

1. Given the first of these, *tac*2 returns two next states.
2. Given the second of these, *tac*2 returns no next states; thus this possibility contributes nothing to the final sequence of outcomes.
3. Given the third of these, *tac*2 returns one next state.

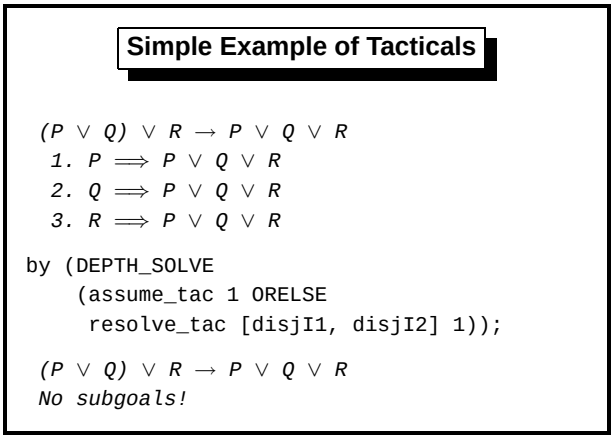
A total of three states can arise from this call to *tac*1 `THEN` *tac*2.

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.

Variations on `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.

Slide 305



Here is a demonstration of our primitives. Trying to prove that $\vee$ is associative has yielded three subgoals, each to prove $P \vee (Q \vee R)$ from the assumption P, Q or R. At our disposal are Isabelle versions of the disjunction rules:

$$P \Longrightarrow P \vee Q \qquad\qquad Q \Longrightarrow P \vee Q$$

The command shown above attempts proof by assumption, if possible, and otherwise applies one of the disjunction rules. The first disjunction rule is always chosen initially, the second after 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. Isabelle's *classical reasoner* is a set of tactics designed to handle the most common cases automatically, so that we do not have to make such decisions.

Slide 306

The Classical Reasoner

`fast_tac`, `best_tac`: prove goal using supplied rules

- *Safe* rules can be applied at any time.
- *Unsafe* rules lose information or create unknowns.

Negated assumptions simulate sequents.

Tactics work with *analytic* rules.

Analytic rules are those that break up a conclusion or assumption into smaller parts. We may distinguish between *safe* and *unsafe* rules. Safe rules do not require backtracking; they represent logical equivalences, and are analogous to rewrite rules such as $m \times (i+j) = m \times i + m \times j$. Unsafe rules may require backtracking.

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

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

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.

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

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 can handle large numbers of rules, packaged together using discrimination nets for fast retrieval. It is implemented using Isabelle 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 particularly good at reasoning about inductively defined relations. When it fails, it often fails quickly, and can be single-stepped to locate the trouble spot.

Slide 307

Examples of the Classical Reasoner

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

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

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

diamond parcontract

The first theorem is #40 from Pelletier's problem set [35]. It is rather easy; its proof requires only 0.5 seconds on a fast SPARCstation. The harder problems from this set are too big to fit on the slide.

The second theorem is typical of many set-theoretic identities that `fast_tac` proves automatically. To do this, `fast_tac` uses rules proved specifically about the primitives of set theory, rather than expanding the definitions to primitive logical concepts. Thus it reasons about unions and intersections at a high level. This allows users to follow what is happening, should the proof fail. It also promotes efficiency; the theorem shown is proved in 0.3 seconds.

The third example is the deduction theorem. It comes from a proof of the soundness and completeness of propositional logic, by Tobias Nipkow and myself [32]. Unlike the other examples, it is not proved by `fast_tac` alone. The first step of the proof is induction on the derivation of $\{p\} \cup H \vdash q$; 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 [34]. Again it consists of induction followed by classical reasoning. But this time `fast_tac` (equipped with rules about the behaviour of reductions) needs 43 seconds to prove the four subgoals.

Slide 308

A Generic Simplifier

Rewriting can be conditional, permutative, . . .

. . . and *take account of context*:

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

User-supplied context and search strategies

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 $e_1 + \cdots + c_n$. But few of them provide generic support to adapt rewriting to particular theories.

The rule shown on the slide describes a way of extracting context from conjunctions. 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. If we define bounded quantification over sets, we can prove and install the analogous rule for it:

$$\frac{A = A' \qquad \begin{matrix}[x \in A'] \\ P(x) \leftrightarrow P'(x)\end{matrix}}{(\forall x \in A.\, P(x)) \leftrightarrow (\forall x \in A'.\, P'(x))}$$

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; these too are described by inference rules. Isabelle's simplifier also lets users describe how rewrite rules are extracted from general formulæ. For example, a formula of the form $\forall x \in A.\, P(x)$ might yield rewrite rules conditional upon $x \in A$.

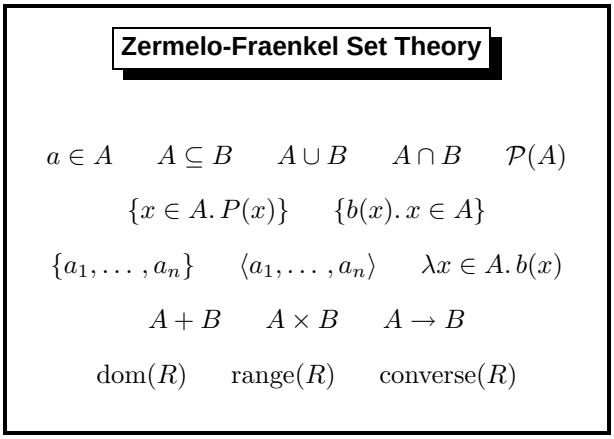
These extensions yield sound results because they work by performing proofs. A user error cannot make the simplifer replace everything by true.

Slide 309

Summary

- Built-in tactics support proof checking.
- Complex tactics can be built using tacticals.
- Several search strategies are available.
- The classical reasoner works with any analytic rules.
- The simplifier can be extended for new connectives.

Slide 401



Isabelle implements two set theories. Isabelle/ZF [28] is built upon first-order logic using the standard Zermelo-Fraenkel axioms [42]. 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.

Why use set theory at all? We need collections of some sort and the only alternative is a type theory. Simple type theories (such as HOL) are in themselves too rigid; one cannot, for example, form the collection of lists of a given length. Predicate subtyping (as in PVS) can help. But even in the Calculus of Constructions, a highly expressive type system, one usually defines some sort of set theory. Set-theoretic primitives can express complex systems. They are found in the Z and B specification languages; many practitioners have advocated their use.

Finite sets are written $\{a_1, \ldots, a_n\}$, tuples $\langle a_1, \ldots, a_n \rangle$ and functions $\lambda x \in A.\, b(x)$.

Isabelle/ZF includes other primitives not shown above. 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 $\bigcup_{x \in A} B(x)$ are defined in terms of $\bigcap(A)$ and $\bigcup(A)$, where A is a set of sets. The operators $\times$ and $\to$ are generalized to the dependent constructions $\sum_{x \in A} B(x)$ and $\prod_{x \in A} B(x)$.

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, ..., 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.

Slide 402

High-Level Rules

$$(!!x.\, x \in A \Longrightarrow x \in B) \Longrightarrow A \subseteq B$$

$$[\![A \subseteq B;\ a \in A]\!] \Longrightarrow a \in B$$

$$[\![a \in A;\ b \in B(a)]\!] \Longrightarrow b \in \left(\bigcup_{x \in A} B(x)\right)$$

$$[\![f \in A \to B;\ a \in A]\!] \Longrightarrow f`a \in B$$

$$[\![a \in A;\ P(a)]\!] \Longrightarrow a \in \{x \in A.\, P(x)\}$$

These are examples of the rules used in Isabelle/ZF proofs. They have been derived from the Zermelo-Fraenkel axioms. Proofs directly in terms of those axioms would be barbaric. Some of the rules have long and complicated proofs. These are performed once and for all when Isabelle/ZF is built. Such rules behave identically to built-in ones in use; there is no efficiency penalty for using them, and they work with all the same commands.

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

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

In Isabelle, the axiom above behaves as a scheme. We may instantiate $?P$ with various λ-expressions, and the result is automatically β-reduced to obtain the required instances.

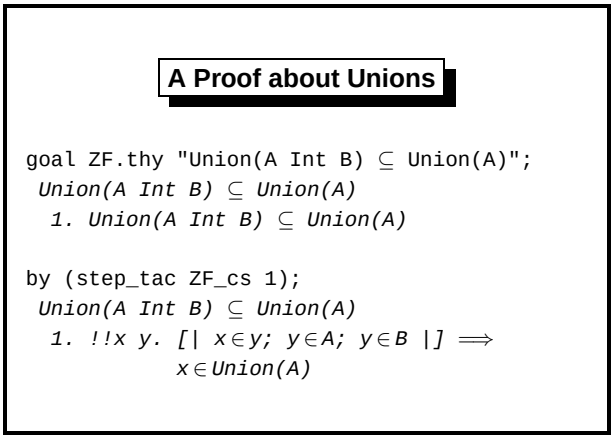
In the current version, the "axiom" of separation is actually a theorem. It is proved from the axiom of replacement. With Isabelle we need not care whether something is an axiom or a theorem. Theorem schemas pose no more difficulties than axiom schemas do.

The rules shown above are couched in terms of membership ($\in$). Rules involving $\subseteq$ can yield particularly high-level proofs, when they apply. Recall the first lecture's use of rules such as $\mathcal{P}$-mono and $\cap$-greatest:

$$A \subseteq B \Longrightarrow \mathcal{P}(A) \subseteq \mathcal{P}(B)$$

$$[\![C \subseteq A;\ C \subseteq B]\!] \Longrightarrow C \subseteq A \cap B$$

Slide 403



This theorem can be proved in one step using `fast_tac`. For demonstration, let us prove it in smaller steps using a tactic called `step_tac`. This tactic is normally called via `fast_tac`; calling it directly allows for "single-step" reasoning, especially useful when the proof fails. I have slightly modified the output to improve readability.

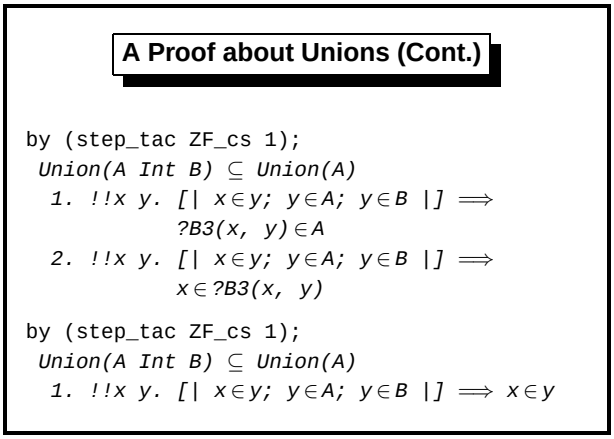
Calling `step_tac` either applies "safe" inferences to the entire proof state, or applies one "unsafe" inference to the specified subgoal. Safe inferences are analogous to rewriting by rules such as

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

Typical natural deduction rules express each direction of such an equivalence separately. Unsafe inferences are not equivalences at all.

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

Slide 404



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 $x \in U$. Isabelle displays the unknown set U as `?B3(x,y)`. It appears in two goals; in general, different proofs of a goal instantiate its unknowns differently, and we may have to search for an instantiation 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.

It is rather hard to find examples simple enough to present on a slide, and hard enough to require more than one call to `step_tac`. More interesting, but with a slightly longer proof, is the monotonicity rule for $\bigcup$:

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

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

Slide 405

Functions and Injections

$$\mathrm{func}(f) \equiv \forall x\, y.\langle x,y\rangle \in f \rightarrow \forall z.\langle x,z\rangle \in f \rightarrow y = z$$

$$A \rightarrow B \equiv \{f \subseteq A \times B.\, A \subseteq \mathrm{dom}(f) \wedge \mathrm{func}(f)\}$$

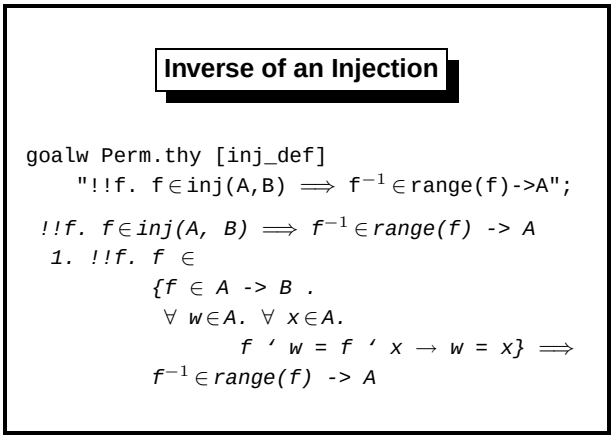
$$\mathrm{inj}(A,B) \equiv \{f \in A \rightarrow B. \forall w, x \in A. f`w = f`x \rightarrow w = x\}$$

These definitions are taken from the standard Isabelle/ZF libraries. The representation of functions in terms of ordered pairs is cumbersome; most proofs are conducted in terms of higher-level rules about λ-abstraction and function application. Users need only know the third definition, which defines in the standard way the set of injections from A to B. These definitions illustrate the use of the notation and form the basis for the examples following.

Here $f`x$ denotes application to the function f to the argument x. It is object-level application, just as $\rightarrow$ is object-level implication. These differ from meta-level application, written $f(x)$, and meta-level implication, written $\Longrightarrow$. 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 just a term of function type. Many meta-level functions, such as $\bigcup$ and $\mathcal{P}$, are defined over the entire universe of sets and thus cannot be represented as object-level functions.

Due to space limitations I have abbreviated the operators "function" and "domain", and written $f \subseteq A \times B$ instead of $f \in \mathcal{P}(A \times B)$. To simplify the presentation I use the function space $A \rightarrow B$ where Isabelle/ZF uses the general product $\prod_{x \in A} B(x)$. The general product, and the general sum $\sum_{x \in A} B(x)$, are as useful in set theory as their counterparts are in dependent type theories.

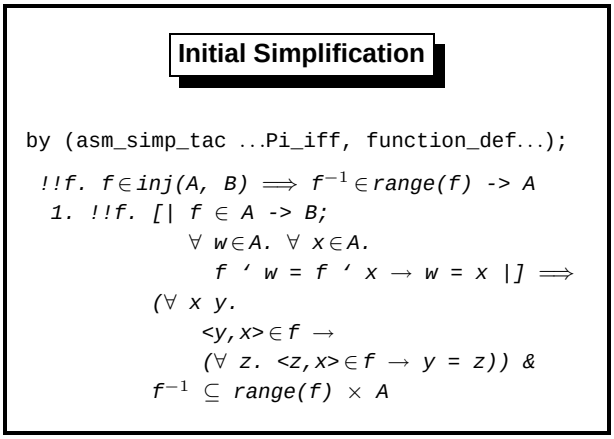
Slide 406



Injections, surjections and bijections are fundamental to discrete mathematics and computer science. Many papers on formal methods take a simple databases as their example. Most databases label each record by a unique primary key; thus there is an injection from records to primary keys. The theorem above implies that the inverse of injection is a function from primary keys to records. (In fact it is a bijection, but we don't prove that here.)

This example illustrates use of Isabelle's simplifier and classical reasoner on nontrivial theorems. Here we see the initial proof state after expanding the definition of $\mathrm{inj}(A, B)$. I have inserted some mathematical symbols (like $\forall$ for `ALL` and f^{-1} for converse(f)) to improve readability and save space.

Slide 407



The full command given to Isabelle was

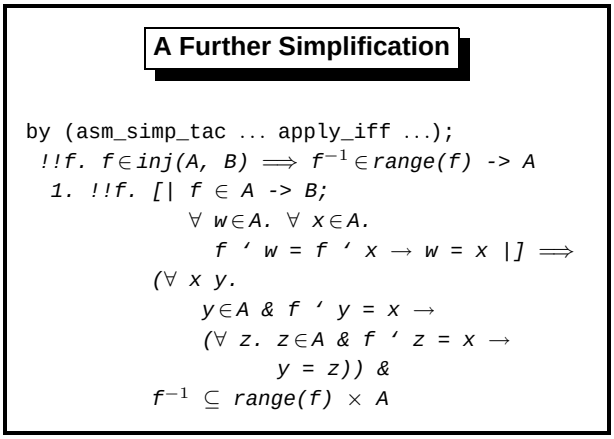
```
by (asm_simp_tac (ZF_ss addsimps [Pi_iff, function_def]) 1);
by (eresolve_tac [CollectE] 1);
```

The new proof state is the result of unfolding the definitions of function space and the "function" predicate, and then splitting up an assumption of the form $\{f \in A \to B. \ldots\}$. Here, rewriting affects only the subgoal's conclusion, not its assumptions; an occurrence of the function space operator remains in the assumptions as $A \to B$.

Default rules (stored in `ZF_ss`) perform most of the reasoning. Furthermore we use `Pi_iff` and `function_def` as rewrite rules, to expose the representation of functions. Normally we should not want to do that but this proof needs it.

Rewriting has proved part of the unfolded subgoal, namely $\mathrm{dom}(f^{-1}) \subseteq \mathrm{range}(f)$. We still have to show that f^{-1} is a function and is included in $\mathrm{range}(f) \times A$.

Slide 408



The command shown above rewrites with `apply_iff`, which stands for the clause

$$f \in A \to B \Longrightarrow \langle a, b\rangle \in f \leftrightarrow (a \in A \wedge f`a = b).$$

The effect is to replace $\langle a, b\rangle \in f$ by $f`a = b$ when we know that f is a function. The connection between the quantified assumption and the first conjunct is now clear. We still have to prove it, and the second conjunct, $f^{-1} \subseteq \mathrm{range}(f) \times A$. It sounds hard but it turns out 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 is not the sort of rule that one would normally use, but this proof is about how functions are represented as sets of pairs. Isabelle's `fast_tac` can manage with quantified formulæ such as those on the slide, and returns quickly if they are reasonably simple. With the present subgoal, `fast_tac` needs only 1.3 seconds, of which one second is devoted to the first conjunct.

Slide 409

Survey of Isabelle/ZF

Recursive functions — using well-founded relations

Recursive data structures — with infinite branching

Several computational case studies

Standard set theory: ordinals, cardinals, . . .

Proofs in advanced set theory: equivalents of AC

Slide 410

Summary

- Set theory constructs are useful in specifications.
- Reasoning about sets requires high-level rules.
- Unknowns arise naturally in set theory proofs.
- The simplifier and classical reasoner are effective in set theory.

Exercise 7 Using high-level rules such as those demonstrated above, prove the monotonicity of $\bigcup$:

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

Exercise 8 Use the theorem $[\![f \in \text{inj}(A, B);\ b \in \text{range}(f)]\!] \Longrightarrow f`(f^{-1}`b) = b$ to strengthen our result to

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

Slide 501

Declaring Recursive Types

`datatype` $(\alpha_1, \ldots, \alpha_m)t = constr_1 \mid \ldots \mid constr_n$

Each $constr$ has the form $C\ \tau_1\ \ldots\ \tau_k$

For lists, trees, . . . , equipped with primitive recursion

Example, binary trees:

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

A datatype is a disjoint sum, which may be recursive. The syntax is based upon that of Standard ML [33]. Datatypes can model lists, trees, finite enumerations, etc.

[For this lecture 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. Even with this, I have had to simplify much of the Isabelle text in order to fit it on the slides.]

Datatype declarations may be placed in theory files along with declarations of other types, constants, definitions, etc. Each declaration causes Isabelle to make appropriate definitions and derive the properties usually expected of a datatype. We can refer to the properties as Isabelle rules, which are bound to ML identifiers.

The slide declares a polymorphic type of labelled binary trees. The ML identifier `bt.distinct` is bound to a list of theorems stating that the constructors are *distinct* (leaves are not branches):

```
Lf ~= Br a t1 t2      Br a t1 t2 ~= Lf
```

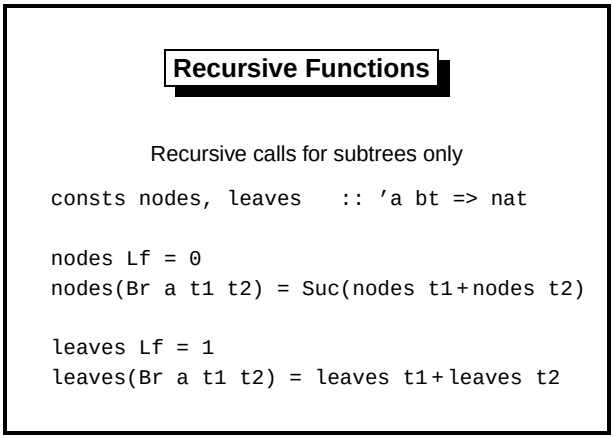
The identifier `bt.inject` is bound to a list of theorems stating that the constructors are *injective* (we can extract a branch's subtrees):

```
(Br a t1 t2 = Br a' t1' t2')  =  (a = a'  &  t1 = t1'  &  t2 = t2')
```

The identifier `bt.induct` is bound to a rule for *structural induction*:

```
[| P Lf;
   !!a t1 t2. [| P t1; P t2 |] ⟹ P(Br a t1 t2)
|] ⟹ P bt
```

Slide 502

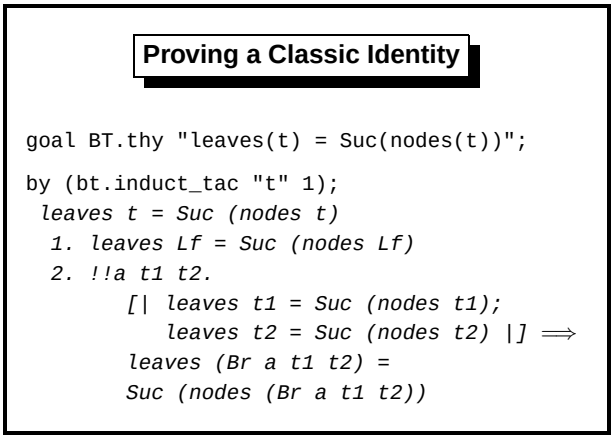


Isabelle (in ZF or HOL) supports function definition by well-founded recursion. Any relation that can be proved to be well-founded may be used to show termination of recursive calls. We may even interleave the proofs of termination and those of other correctness properties; this is essential for reasoning about certain nested recursive functions, such as the unification algorithm. Konrad Slind has recently written a tool to automate much of this process; its accepts function definitions expressed using pattern-matching [41].

The overhead shows a simpler form of function definition: primitive recursion. Isabelle/HOL provides special syntax for declaring functions that make recursive calls only to *immediate subparts* of their argument. Examples of immediate subparts are the predecessor of a natural number, the tail of a list or the two children of a binary tree.

Most algorithms can be expressed as recursive functions, using the techniques of functional programming. The chief exceptions are concurrent algorithms and those that are inherently imperative, such as the union-find data structure.

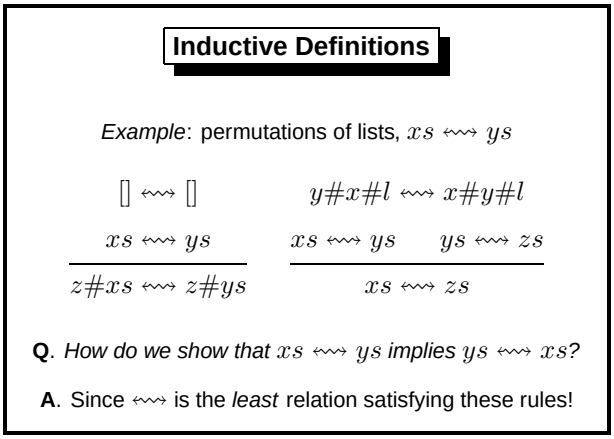
Slide 503



This example illustrates how inductive proofs are performed in Isabelle. The user must specify the induction variable and rule. Isabelle does not contain heuristics for inventing induction schemes, as they are difficult to implement do not save users much effort. The correct induction scheme is usually obvious.

The slide shows the proof state after applying induction. Isabelle's simplifier, using default rewrites, can prove both subgoals automatically. Many properties of recursive functions are proved with equal ease.

Slide 504



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 [21].

A desired collection of rules may be given to Isabelle (ZF and HOL) to specify a (co)inductive definition. Isabelle reduces it to a fixedpoint definition, using the Knaster-Tarski theorem. An inductive definition is a least fixedpoint; a coinductive definition is a greatest fixedpoint. The rules may involve any operators that have been proved monotone. Definitions may be iterated: one (co)inductive set may be used to define another.

Given a (co)inductive definition, Isabelle returns (as theorems bound to ML identifiers) the main properties of the (co)inductively 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.

To demonstrate inductive definitions, the slide defines the "permutation of" relation for lists. We could model permutations more generally using bijections, but the inductive definition is easy to use.

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 swapping the first two elements of a list creates a permutation. Of the lower two rules, one says that adding identical elements to both lists preserves the "permutation of" relation; the other says that "permutation of" is transitive.

"Permutation of" is an equivalence relation. Transitivity is given. Reflexivity is trivial: prove $xs \leftrightsquigarrow xs$ by list induction, using the two left-hand rules. Symmetry seems harder; we can prove it using rule induction.

Slide 505

Rule Induction for Permutations

$xs \leftrightsquigarrow ys$ implies $P\,xs\,ys$ if P satisfies the $\leftrightsquigarrow$ rules!

- $P\,[]\,[]$
- $P\,(y\#x\#l)\,(x\#y\#l)$
- if $P\,xs\,ys$ then $P\,(z\#xs)\,(z\#ys)$
- if $P\,xs\,ys$ and $P\,ys\,zs$ then $P\,xs\,zs$

Rule induction [45] is a powerful inference rule for proving consequences of $xs \leftrightsquigarrow ys$. Recall that $\leftrightsquigarrow$ is the least set closed under the given rules. If some predicate P is also closed under those rules then $xs \leftrightsquigarrow ys$ implies $P\,xs\,ys$ for all x and y. The slide shows the corresponding subgoals. (Isabelle actually derives a stronger induction rule, but the details don't concern us here.)

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

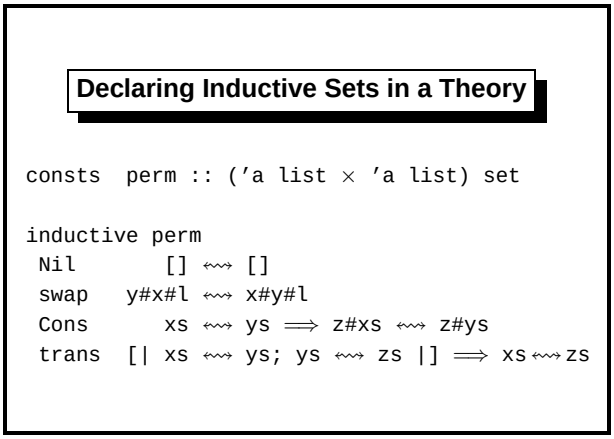
- $\mathrm{length}[] = 0 = \mathrm{length}[]$
- $\mathrm{length}\,(y\#x\#l) = 2 + \mathrm{length}\,l = \mathrm{length}\,(x\#y\#l)$
- If $\mathrm{length}\,xs = \mathrm{length}\,ys$ then

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

- if $\mathrm{length}\,xs = \mathrm{length}\,ys$ and $\mathrm{length}\,ys = \mathrm{length}\,zs$ then $\mathrm{length}\,xs = \mathrm{length}\,zs$ 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 swap the operands of $\leftrightsquigarrow$. The Isabelle versions of the declaration of $\leftrightsquigarrow$ and the proof of symmetry appear on slides below.

Slide 506



This illustrates Isabelle's syntax, though 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`. We specify the inductive definition using this notation.

(Co)inductive definitions can specify other information. If the definition involves other operators then they must be monotonic; Isabelle must be given the corresponding monotonicity theorems. In ZF, the definition requires rules to perform a kind of type checking.

Isabelle returns the list of introduction rules, bound to the ML identifier `perm.intrs`:

```
[] ⇜⇝ []      y#x#l ⇜⇝ x#y#l      other two as above
```

The elimination rule, bound to `perm.elim`, allows case analysis.

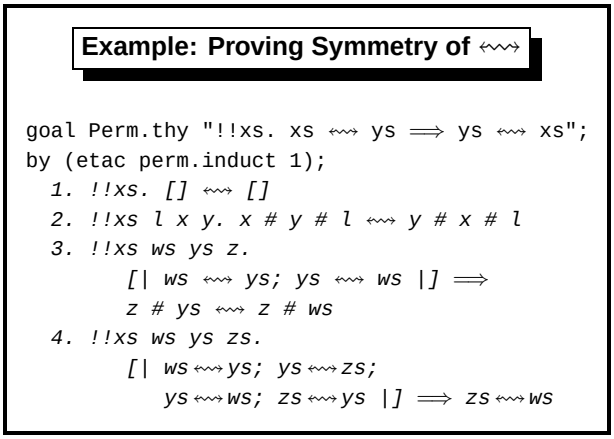
```
[| a ∈ perm;
   a = ([], []) ⟹ Q;
   !!l x y. a = (y#x#l, x#y#l) ⟹ Q;
   !!xs ys z. [| a = (z#xs, z#ys); xs ⇜⇝ ys |] ⟹ Q
   ...
|] ⟹ Q
```

This rule resembles induction but has no induction hypotheses, and is generated even for coinductive definitions. Variants of it can be used with `fast_tac` to perform a kind of backward reasoning known as *rule inversion*.

The induction rule, as described on a previous slide, is bound to `perm.induct`.

(Co)inductive definitions are conservative. The rules they return are proved as the consequences of a low-level definition — they are not merely asserted. Most definitions can be processed in a few seconds.

Slide 507

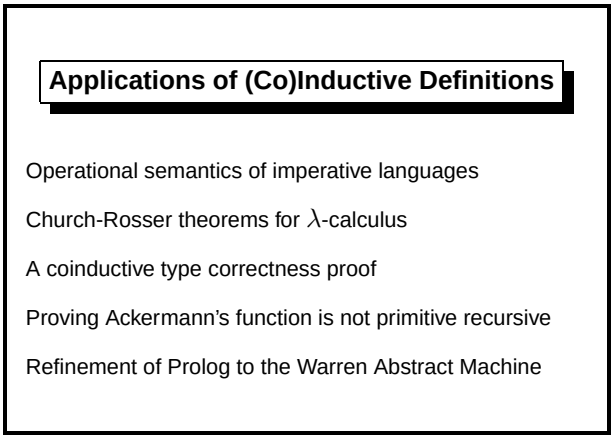


Here we see the result of using rule induction to prove symmetry. Compare the subgoals shown above with those described earlier. They are all instances of the introduction rules. We can prove them trivially using a single command:

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

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

Slide 508



Several large studies use inductive definitions. Ltzbeyer, Sandner and Nipkow [17, 24] have proved several properties relating the operational and denotational semantics of Winskel's toy programming language IMP [45]. Using different techniques, Nipkow [25] and Rasmussen [38] have both proved the Church-Rosser theorem. A datatype specifies the set of λ-terms, while inductive definitions specify several reduction relations.

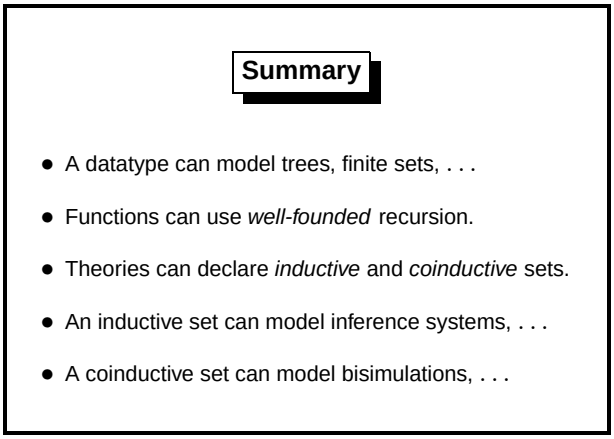
To demonstrate coinductive definitions, Frost [10] has proved the consistency of the dynamic and static semantics for a small functional language. The example, by Milner and Tofte [22], concerns a coinductively defined typing relation.

Isabelle/ZF supports *codatatypes*, which are like datatypes but admit infinitely deep nesting: trees need not be well-founded. (Doing this in the presence of the foundation axiom requires the use of variant pairs and function [31].) Frost uses a codatatype definition to specify 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 [29] demonstrates the flexibility of inductive definitions in Isabelle. The set of primitive recursive functions is difficult to define formally, because 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.

Cornelia Pusch [37] 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.

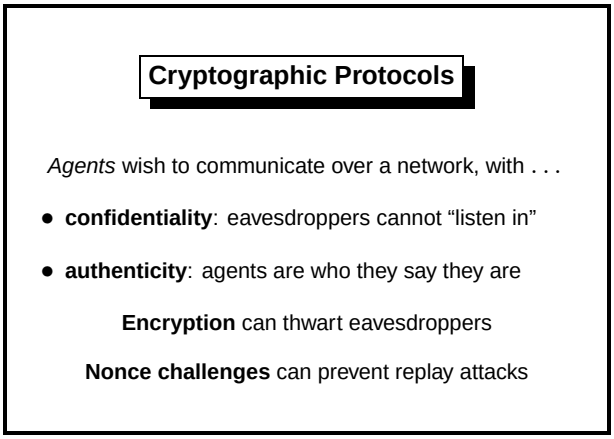
Slide 509



Exercise 9 Using recursion and an if-then-else construct, define a function $\mathrm{count}\, l\, z$ to count how many times z occurs in the list l.

Exercise 10 Using rule induction, prove that $xs \leftrightsquigarrow ys$ implies $\mathrm{count}\, xs\, z = \mathrm{count}\, ys\, z$.

Slide 601



We look at two applications in detail. Safety properties of cryptographic protocols can be proved using Isabelle's support for inductive definitions. Interactive program derivation makes use of Isabelle's treatment of unknowns in goals. Finally we have a brief survey of other work.

There are too many combinations for each agent to have a shared key with each potential partner. Instead a **Key Server** assigns fresh keys for this purpose. Each agent shares a key with the server.

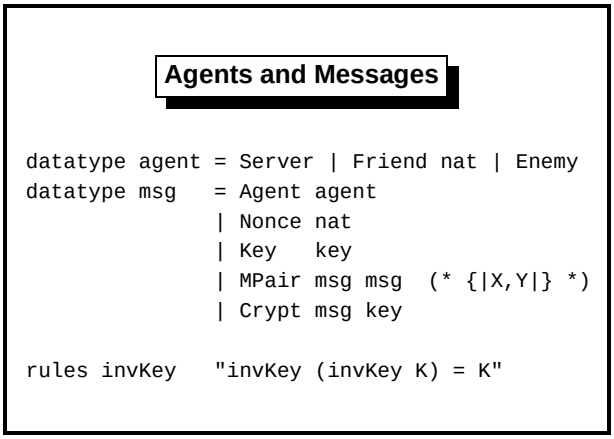
Authenticity involves assuring somebody at the other end of the network of your identity. What if an enemy reads a message sent by agent A, and later replays it (or something composed from it) to agent B? He might manage to fool B into thinking he is really A, and then cause havoc in A's name.

A *nonce* is a sort of password, used to prevent such replay attacks. An agent can include a fresh nonce in a message, expecting the other party to send that nonce back again (as part of a new message). If the nonces do not agree then the attempted connection will fail. Old messages previously seen by the enemy will have out-of-date nonces, and thus will be useless.

We do not suppose that agents are conscious, capable of being "fooled" or "convinced". Rather they operate a *protocol* and behave precisely as the protocol dictates.

Encryption *does not* guarantee authenticity or security. Many protocols based on encryption have been shown to be vulnerable, often in subtle ways. Rigorous correctness proofs seem essential. One popular method is based upon logics of belief [4]. The method outlined below is entirely different. Inductive definitions are used to specify the elements of messages and possible traces of protocol runs.

Slide 602



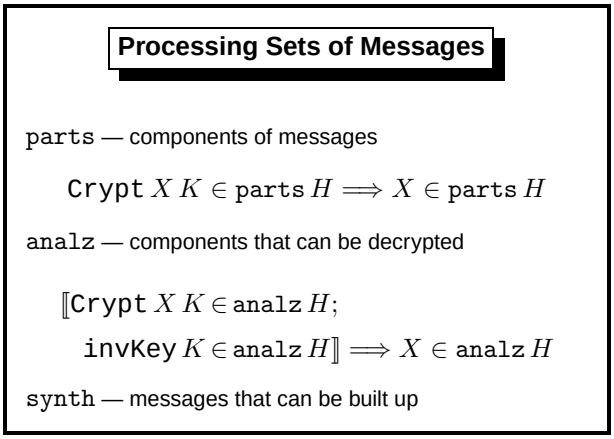
Agents include the Server, the friendly agents and the Enemy. We can model attacks in which the Enemy takes part in protocols is if he were a friendly agent. A more restrictive alternative is to build the Enemy into the network.

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 a special notation.

Our datatype makes the assumption that encryption is injective. This is false in the real world: one may "decrypt" a message using various keys. Standard practice, however, is to include redundancy in all messages so that uses of the wrong key can be detected.

Constant `invKey` models public-key encryption. Each public key K has an inverse, written K^{-1}, which is kept private. To model conventional (shared key) encryption we just assume $K^{-1} = K$.

Slide 603



If H is a set of messages then

- `parts` H is the set of all components of H (going down recursively). It concerns the structure of messages alone.
- `analz` H is similar, but one needs the key to get at the body of an encrypted message. Thus it relates to security of messages.
- `synth` H is the set of all messages that can be built using elements of H as components

The set operations all have straightforward inductive definitions, e.g.

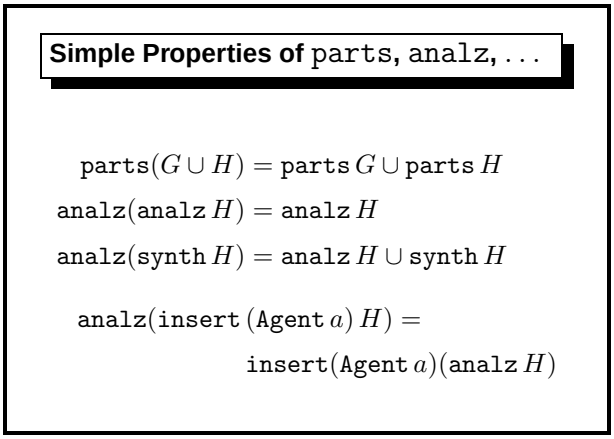
```
consts  analyze   :: msg set => msg set
inductive "analyze H"
  intrs
    Inj     "X∈H ⟹ X∈analyze H"

    Fst     "|X,Y| ∈analyze H ⟹ X ∈analyze H"

    Snd     "|X,Y| ∈analyze H ⟹ Y ∈analyze H"

    Decrypt "[| Crypt X K ∈analyze H; Key(invKey K)∈analyze H
             |] ⟹ X ∈analyze H"
```

Slide 604



Many properties of `parts`, `analz`, `synth` are easily proved by rule induction, classical reasoning and rewriting. Rule inversion plays a minor role except with `synth`. The overhead presents just a few of the approximately 70 theorems proved about these and similar set operations.

Of the three set operations, `parts` is the easiest to reason about. It is the only one of the operations that distributes over union. Given a specific argument it can be evaluated using obvious rules. Atomic members of its argument are simply extracted, becoming members of the result; other members are broken down and the new argument recursively evaluated.

On the other hand, `analz` is the hardest to reason about. During evaluation, we cannot extract a key without first showing that there are no encrypted messages that it could decrypt. Simiarly one cannot extract an encrypted message without knowing whether or not there is a key available to decrypt it. The fourth equation on the slide shows the treatment of agents names in the argument: they can be extracted.

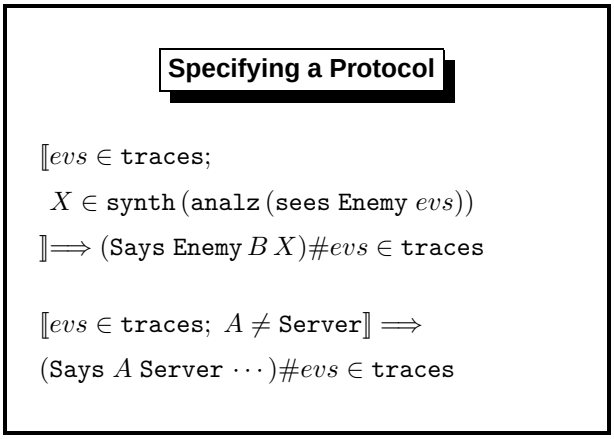
We have several equations that help break down expressions in which the operations are nested. All the operations are idempotent. The equation

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

has a difficult (though largely automatic) proof. It is easily understood, especially with the help of a diagram. An analogous equation holds for `parts`.

There is no equation to break down the combination $\texttt{synth}(\texttt{analz}\,H)$, which is the set of messages that can be built from everything that can be decrypted from H. "Fake" messages from the Enemy are of this form.

Slide 605



An *event* is something of the form `Says` $A\,B\,X$, namely A says X to B. Other events could be envisaged, corresponding to internal actions of agents.

The function `sees` describes what an agent sees from a list of events. The enemy 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.

A *trace* is a list of events that could occur during execution of a protocol. We try to prove that all possible traces are safe — where safety is expressed in terms of keys the Enemy can see, for instance.

Here are just two rules of the inductive definition of traces. The empty list is also a trace: the initial trace. Further rules describe the protocol in terms of what message an agent receives, and what message is sent in response.

The first rule shown describes Enemy behaviour. The enemy may say anything he can say. We do not expect him to invent new nonces here, but he can also participate in protocol runs as if he were friendly.

One normally assumes that the Enemy can block messages. We do not need to model this, as nothing in the framework forces agents to respond to messages anyway.

In early experiments I have formalized part of the Needham-Schroeder protocol with shared keys. Safety properties are proved by rule induction over `traces`. I have managed to prove that, even if the Enemy knows a friendly agent's key, session keys issued for other agents remain secure for ever — from the Enemy and even from other agents. This amounts to both a security property and an authenticity property.

One limitation of the present formalization is that protocol runs may not be interleaved. This limitation could be restricted but it might make already difficult proofs harder still.

Slide 606

Derivation of Functional Programs

Aim: derive correct programs from specifications

Prove a theorem like

$$?f \in \{f \mid \forall x \,.\, P(x) \rightarrow Q(fx)\}$$

Derive a function with precondition P, postcondition Q

Interleave design and verification

Dijkstra [6] is usually credited with the idea that we should transform specifications into correct programs, instead of writing programs and afterwards trying to prove their correctness. The fact remains that people often have intuitions about correct code that is not directly driven by such a formal process. An effective tool for correct-program derivation should allow programmers to interleave programming with proving. Then, their guesses about the program structure can immediately be submitted for verification, and used to decompose the original specification to specifications of the remaining program components.

In the context of functional programming, there are two main approaches. Manna and Waldinger's deductive synthesis [19] is based on classical first order logic. The "proofs as programs" paradigm is based on constructive type theories [43].

Research of this sort can be done in Isabelle, using its ability to prove goals containing unknowns. The unknowns stand for parts of the program that remain to be written, and can be instantiated incrementally. This might happen by direct user command, or perhaps automatically, guided by a logical specification of what that piece of code has to accomplish.

An impressive implementation of Manna and Waldinger's approach is Martin Coen's PhD work on interactive derivation of functional programs [5]. Also relevant is work done at the Max Planck Institute, Saarbrcken [2, 3], on deriving logic and functional programs.

Slide 607

The Elements of CCL

CCL = FOL + *typed sets* + *functional language*

Constructors for pairs, functions and recursion

Types defined set theoretically to include Π, Σ, recursion

– can express *Bool*, *Nat*, $A + B$, $A \times B$, etc.

Tactics for type-checking, proving and instantiation

Examples: sorting, unification, etc.

One can easily use Isabelle to prove theorems of the form $P(?f)$, where P is a specification, and instantiate $?f$ to some function in the logic. This is unsatisfactory for two reasons: (1) typically P will mention $?f$ more than once, resulting in blow-up, and (2) not all functions expressible in ZF or HOL are executable.

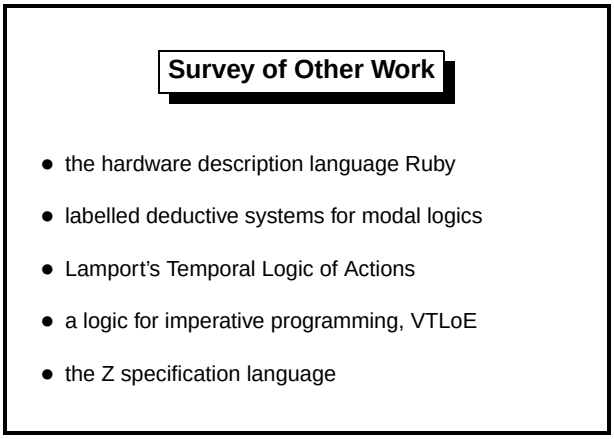
Point (1) can be addressed using the $\{f \mid \forall x\,.\,P(x) \to Q(fx)\}$ notation. Point (2) is best dealt with by formalizing a programming language, instead of identifying programs with mathematical functions.

Coen's Classical Computational Logic (CCL) extends first-order logic with a functional language, defined by an operataional semantics. His approach synthesis derives programs in this language, rather than mathematical functions. *Types* are sets of equivalences classes of terminating programs. They include the Π and Σ types found in constructive type theories, recursive types and subset types. Their definitions are fairly routine, using set theory.

Termination arguments are expressed using well-founded relations, not in terms of the restrictive framework of primitive recursion. Later, unpublished work extended the approach to lazy functional programs. Termination is replaced the standard notion of canonical form (or weak head normal form).

Coen did some extended examples, deriving functional programs for subtractive division, insertion sort and unification. Subtractive division is a simple demonstration of why we need well-founded recursion instead of primitive recursive, while unification is an example of a short program with a complex correctness proof. Much of this work is distributed with Isabelle.

Slide 608



Ole Rasmussen has embedded the relational hardware description language Ruby in ZF.

David Basin, Sen Matthews and Luca Vigan are applying Isabelle to study combinations of logics (via labelled deductive systems). As a first example of modular presentation of logics, they have implemented a wide variety of modal logics (including K, D, T, B, KD45, S4, S4.2 S5).

Sen Matthews is using Isabelle to implement Feferman's theory of finitary inductive definitions, FS0.

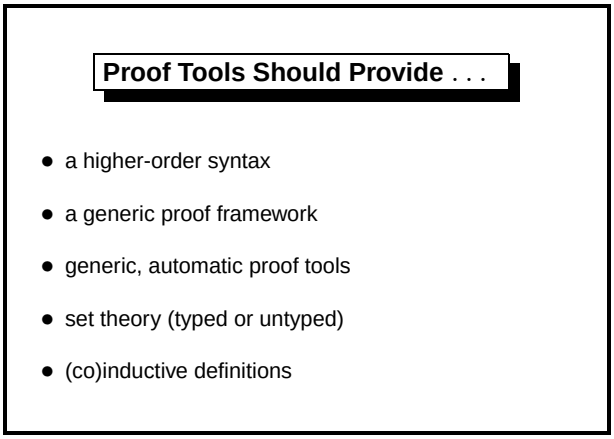
Krzysztof Grąbczewski has mechanized the first two chapters of *Equivalents of the Axiom of Choice* by Rubin and Rubin, in ZF.

Ongoing work involving the Universities of Cambridge, Warwick and Nancy concerns developing support for Lamport's Temporal Logic of Actions and applying it to correctness proofs for concurrent programs.

Jacob Frost has implemented the Variable-Typed Logic of Effects, which is intended to support reasoning about imperative programming at a low level.

The TokiZ research project aims to build a toolkit that supports the use of the formal specification language Z and to study the theoretical issues involved. They have implemented a prototype including a deductive system for Z and a substantial part of the mathematical library associated with Z. The prototype is a deep semantic embedding of Z in Isabelle.

Slide 609



To conclude, let us recall those features of Isabelle that have turned out to be particularly successful. They should be borne in mind when one designs a new tool.

A higher-order syntax is essential to support variable binding. This does not mandate the use of higher-order logic, though it is not a bad choice.

Many formal methods involve specialized formalisms, so a generic framework is valuable. This includes strong support for various notations, as well as an ability to mechanize various forms of reasoning. Researchers using other tools (such as HOL and PVS) often embed formalisms into them.

Automatic proof support is absolutely essential. Proof checking cannot cope with proofs of realistic size. Decision procedures are useful, provided they are not too brittle. Isabelle's classical reasoner and simplifier cope with many classical logics.

PVS is an interesting experiment in the use of an elaborate type system. Its subtypes do many of the same jobs as typed sets. PVS type-checking ensures that many checks are made that users might otherwise forget about. Nonetheless, this type system is very complex, and does not obviate the need for set-theoretic constructs. Users of higher-order logic have often, in the past, reasoned about sets in ad-hoc ways in terms of predicate variables. We need proper support for set theory, including all the usual operators, and the ability to reason about their properties.

Many computational phenomena can be modelled using inductive or coinductive definitions.

Acknowledgements. Michael Jones, Fabio Massacci, Chris Owens, Mark Staples and Myra VanInwegen commented on these notes. 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 Nol. The research was funded by several grants including EPSRC GR/K57381 "Mechanising Temporal Reasoning" and ESPRIT 6453 "Types."

References

[1] Peter Aczel. An introduction to inductive definitions. In J. Barwise, editor, *Handbook of Mathematical Logic*, pages 739–782. North-Holland, 1977.

[2] Penny Anderson and David Basin. Deriving and applying logic program transformers. In *Algorithms, Concurrency and Knowledge (1995 Asian Computing Science Conference)*, LNCS 1023, pages 301–318, Pathumthani, Thailand, December 1995. Springer.

[3] Abdelwaheb Ayari and David Basin. Generic system support for deductive program development. In T. Margaria and B. Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems: second international workshop, TACAS '96*, LNCS 1055, pages 313–328. Springer, 1996.

[4] M. Burrows, M. Abadi, and R. M. Needham. A logic of authentication. *Proceedings of the Royal Society of London*, 426:233–271, 1989.

[5] Martin D. Coen. *Interactive Program Derivation*. PhD thesis, University of Cambridge, November 1992. Computer Laboratory Technical Report 272.

[6] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[7] Gilles Dowek et al. The Coq proof assistant user's guide. Technical Report 154, INRIA-Rocquencourt, 1993. Version 5.8.

[8] Amy Felty. Implementing tactics and tacticals in a higher-order logic programming language. *Journal of Automated Reasoning*, 11(1):43–82, 1993.

[9] Amy Felty and Dale Miller. Encoding a dependent-type λ-calculus in a logic programming language. In Mark E. Stickel, editor, *10th International Conference on Automated Deduction*, LNAI 449, pages 221–235. Springer, 1990.

[10] Jacob Frost. A case study of co-induction in Isabelle. Technical Report 359, Computer Laboratory, University of Cambridge, February 1995.

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

[12] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993.

[13] Matthew Hennessy. *The Semantics of Programming Languages: An Elementary Introduction Using Structural Operational Semantics*. Wiley, 1990.

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

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

[16] R. E. Korf. Depth-first iterative-deepening: an optimal admissible tree search. *Artificial Intelligence*, 27:97–109, 1985.

[17] Heiko Lötzbeyer and Robert Sandner. Proof of the equivalence of the operational and denotational semantics of IMP in Isabelle/ZF. Project report, Institut für Informatik, TU München, 1994.

[18] Zhaohui Luo and Randy Pollack. LEGO proof development system: User's manual. Technical Report ECS-LFCS-92-211, Department of Computer Science, University of Edinburgh, 1992.

[19] Zohar Manna and Richard Waldinger. Fundamantals of deductive program synthesis. *IEEE Transactions on Software Engineering*, 18(8):674–704, August 1992.

[20] W. McCune. Otter 3.0 Reference Manual and Guide. Technical Report ANL-94/6, Argonne National Laboratory, Argonne, IL, 1994.

[21] Robin Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[22] Robin Milner and Mads Tofte. Co-induction in relational semantics. *Theoretical Computer Science*, 87:209–220, 1991.

[23] Tobias Nipkow. Order-sorted polymorphism in Isabelle. In Gérard Huet and Gordon Plotkin, editors, *Logical Environments*, pages 164–188. Cambridge University Press, 1993.

[24] Tobias Nipkow. Winskel is (almost) right: Towards a mechanized semantics textbook. In V. Chandru and V. Vinay, editors, *Foundations of Software Technology and Theoretical Computer Science*, volume 1180 of *LNCS*, pages 180–192. Springer, 1996.

[25] Tobias Nipkow. More Church-Rosser proofs (in Isabelle/HOL). *Journal of Automated Reasoning*, 26:51–66, 2001.

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

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

[28] Lawrence C. Paulson. Set theory for verification: I. From foundations to functions. *Journal of Automated Reasoning*, 11(3):353–389, 1993.

[29] Lawrence C. Paulson. A fixedpoint approach to implementing (co)inductive definitions. In Alan Bundy, editor, *Automated Deduction — CADE-12 International Conference*, LNAI 814, pages 148–161. Springer, 1994.

[30] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*. Springer, 1994. LNCS 828.

[31] Lawrence C. Paulson. A concrete final coalgebra theorem for ZF set theory. In Peter Dybjer, Bengt Nordström, and Jan Smith, editors, *Types for Proofs and Programs: International Workshop TYPES '94*, LNCS 996, pages 120–139. Springer, 1995.

[32] Lawrence C. Paulson. Set theory for verification: II. Induction and recursion. *Journal of Automated Reasoning*, 15(2):167–215, 1995.

[33] Lawrence C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 2nd edition, 1996.

[34] Lawrence C. Paulson. Generic automatic proof tools. In Robert Veroff, editor, *Automated Reasoning and its Applications: Essays in Honor of Larry Wos*, chapter 3. MIT Press, 1997.

[35] F. J. Pelletier. Seventy-five problems for testing automatic theorem provers. *Journal of Automated Reasoning*, 2:191–216, 1986. Errata, JAR 4(2) (1988), 235–236 and JAR 18(1) (1997), 135.

[36] Frank Pfenning. Logic programming in the LF logical framework. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.

[37] Cornelia Pusch. Verification of compiler correctness for the WAM. In von Wright et al. [44].

[38] Ole Rasmussen. The Church-Rosser theorem in Isabelle: A proof porting experiment. Technical Report 364, Computer Laboratory, University of Cambridge, May 1995.

[39] Elaine Rich and Kevin Knight. *Artificial Intelligence*. McGraw-Hill, 2nd edition, 1991.

[40] Peter Schroeder-Heister. A natural extension of natural deduction. *Journal of Symbolic Logic*, 49(4):1284–1300, December 1984.

[41] Konrad Slind. Function definition in higher-order logic. In von Wright et al. [44].

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

[43] Simon Thompson. *Type Theory and Functional Programming*. Addison-Wesley, 1991.

[44] J. von Wright, J. Grundy, and J. Harrison, editors. *Theorem Proving in Higher Order Logics: TPHOLs '96*, LNCS 1125. Springer, 1996.

[45] Glynn Winskel. *The Formal Semantics of Programming Languages*. MIT Press, 1993.