Isabelle-91*

Tobias Nipkow† and Lawrence C. Paulson‡

Isabelle is a generic theorem prover. Object-logics are formalized within higher-order logic, which is Isabelle's meta-logic. Proofs are performed by a generalization of resolution, using higher-order unification. The latest incarnation of Isabelle, *Isabelle-91*, features a type system based on order-sorted unification; this supports polymorphism and overloading in logic definitions.

1 Defining logics

Isabelle's meta-logic is intuitionistic higher-order logic with implication ($\Longrightarrow$), universal quantifiers ($\bigwedge$), and equality ($\equiv$) [5, 6]. The presentation of an object-logic consists of a signature introducing the types and constants of the logic, i.e. its abstract syntax, and axioms describing the inference rules. As a tiny example, consider the following definition of minimal logic.

$$\begin{array}{lll} \mathcal{M}in & = & \textbf{types } \mathit{form} \\ & & \textbf{consts } \_ \longrightarrow \_ : \mathit{form} \to (\mathit{form} \to \mathit{form}) \\ & & \qquad\qquad [\![\_]\!] : \mathit{form} \to \mathit{prop} \\ & & \textbf{rules } ([\![P]\!] \Longrightarrow [\![Q]\!]) \Longrightarrow [\![P \longrightarrow Q]\!] \\ & & \qquad [\![P \longrightarrow Q]\!] \Longrightarrow [\![P]\!] \Longrightarrow [\![Q]\!] \end{array}$$

$\mathcal{M}in$ introduces the type *form* (of object-formulae) and the infix constant $\longrightarrow$ (for object-implication). The constant $[\![\_]\!]$ maps *form* to the predefined type *prop* of meta-logic propositions. The proposition $[\![P]\!]$ should be read as "the formula P is true"; we usually distinguish object-level formulae (*form*) from meta-level formulae (*prop*). In practice the brackets can be dropped; parser and pretty-printer take care of such matters.

The two rules for minimal logic are typical natural deduction rules for implication introduction and elimination, which are usually written as follows:

$$\frac{\begin{array}{c}[P] \\ \vdots \\ Q\end{array}}{P \longrightarrow Q} \qquad\qquad \frac{P \longrightarrow Q \quad P}{Q}$$

*Appeared in D. Kapur (editor), *11th International Conf. on Automated Deduction*, (Springer LNAI 607, 1992), 673–676. Research supported by ESPRIT BRA 3245, *Logical Frameworks*.

†Author's address: Institut für Informatik, TU München, Postfach 20 24 20, 8000 München 2, Germany. E-mail: `Tobias.Nipkow@Informatik.TU-Muenchen.De`.

‡Author's address: University of Cambridge, Computer Laboratory, Pembroke Street, Cambridge CB2 3QG, England. E-mail: `Larry.Paulson@cl.cam.ac.uk`.

Logics can be combined (taking their union) and can be extended with new types, constants and rules. A minimal predicate logic can be defined as an extension of $\mathcal{M}in$ by adding a type of terms and a quantifier:

$$\begin{array}{rcl} \mathcal{P}red & = & \mathcal{M}in \;+\; \textbf{types}\ term \\ & & \quad \textbf{consts}\ \forall : (term \to form) \to form \\ & & \quad \textbf{rules}\ (\bigwedge x.[\![P(x)]\!] \Longrightarrow [\![\forall(P)]\!] \\ & & \qquad\qquad [\![\forall(P)]\!] \Longrightarrow [\![P(t)]\!] \end{array}$$

Because Isabelle is based on higher-order logic, its expressions are simply typed λ-terms. The λ-calculus notions of free and bound variables handle quantifiers. The formula $\forall x.P(x)$, where P is of type $term \to form$, is internally represented as $\forall(\lambda x.P(x))$. The parser and pretty-printer translate between the concrete syntax and the internal form.

The two inference rules formalize the usual rules of quantifier introduction and elimination:

$$\frac{P}{\forall x.P} \qquad\qquad \frac{P[t/x]}{\forall x.P}$$

The introduction rule is subject to the proviso that x is not free in the assumptions. In Isabelle, this proviso is automatically enforced because the premise $[\![P(x)]\!]$ is in the scope of a local $\bigwedge x$ [5]. Similar techniques handle existential quantifiers and expressions such as $\Sigma_{i=0}^{n} k_i$ (summations), $\Pi_{x \in A} B(x)$ (dependent types) and $\bigcup_{\alpha<\gamma} H_\alpha$ (large unions).

2 Order-sorted polymorphism

$\mathcal{P}red$ formalizes a single sorted predicate logic. To support many-sorted and polymorphic logics, Isabelle-91 introduces *order-sorted polymorphism* [3]. This is ML-polymorphism where the algebra of types is order-sorted — there is a new level of partially ordered *sorts* classifying the types. Type variables are qualified by sorts, thus restricting the set of types they range over. This is a generalization of Standard ML's equality types [2] and is closely related to Haskell's type classes [4]. As an example consider the following definition of a polymorphic first-order logic:

$$\begin{array}{rcl} \mathcal{FOL} & = & \textbf{sorts}\ i < \top \\ & & \quad \textbf{types}\ form : \top \\ & & \quad \textbf{consts}\ \_ \longrightarrow \_ : form \to form \to form \\ & & \qquad\qquad \forall : (\alpha_i \to form) \to form \end{array}$$

The first line introduces a sort i ("individuals") which is a subsort of the predefined sort $\top$ of all types. The type of formulae is classified as being of sort $\top$. Implication is as before, whereas $\forall$ has acquired a polymorphic type: the type variable α_i ranges over all types of sort i. In particular α_i does not range over $form$ and function types, because neither are of sort i. This rules out quantification over predicates and functions, thus ensuring that FOL is indeed a first-order logic and not a higher-order one in disguise.

The sort i is initially empty but further extensions may change this:

$$\begin{array}{rcl} \mathcal{N}at & = & \mathcal{FOL} \;+\; \textbf{types}\ nat : i \\ & & \quad\ \textbf{consts}\quad 0 : nat \\ & & \qquad\quad succ : nat \to nat \end{array}$$

The formula $\forall x.P(x) \longrightarrow P(succ(x))$ is legal, with x having the inferred type nat which is of sort i.

A polymorphic equality operator would have type $\alpha_i \to (\alpha_i \to form)$. For higher-order logic, *form* would be declared to have sort i, to permit quantification over formulae [3].

2.1 Overloading

Order-sorted polymorphism can also be used to specify *ad-hoc* polymorphism or *overloading*. Suppose we would like to use the symbol + at more than one type, for example both natural numbers and strings (where + might denote concatenation). Isabelle's type system does not allow the simultaneous declaration

$$\begin{array}{rl} \textbf{consts} & \_+\_ : nat \to nat \to nat \\ & \_+\_ : string \to string \to string \end{array}$$

However, + can be declared once and for all as a polymorphic operator:

$$\begin{array}{rcl} \mathcal{FOL}_+ & = & \mathcal{FOL} \;+\; \textbf{sorts}\; a < i \\ & & \quad\; \textbf{consts}\; \_+\_ : \alpha_a \to \alpha_a \to \alpha_a \end{array}$$

The sort a is meant to represent those types which provide addition (+). More precisely, any type τ of sort a automatically gives rise to a constant $+ : \tau \to \tau \to \tau$. So far there is only the generic operation but there are no instances. The latter are created by asserting that some type is of sort a:

$$\begin{array}{rcl} \mathcal{N}at_+ & = & \mathcal{FOL}_+ \;+\; \mathcal{N}at \;+\; \textbf{types}\; nat : a \\ & & \quad\; \textbf{rules}\; 0 + n = n \\ & & \qquad\quad succ(m) + n = succ(m + n) \end{array}$$

3 Theorem proving in Isabelle

Object-logic proofs can be performed by applying one rule at a time, or in large tactic steps.

3.1 Resolution

The central notion in Isabelle is that of a theorem. All axioms in a logic definition are available as theorems; new theorems can only be derived by combining existing ones. Ignoring $\bigwedge$ and $\equiv$, the general form of a theorem is $\phi_1 \Longrightarrow \ldots \Longrightarrow \phi_m \Longrightarrow \phi$ which we abbreviate as $[\phi_1, \ldots, \phi_m] \Longrightarrow \phi$. Theorems are combined by resolution. Given theorems

$$[\phi_1, \ldots, \phi_m] \Longrightarrow \phi \quad \text{and} \quad [\psi_1, \ldots, \psi_n] \Longrightarrow \psi$$

and a substitution s such that $s(\phi) = s(\psi_k)$, we can infer the new theorem

$$s([\psi_1, \ldots, \psi_{k-1}, \phi_1, \ldots, \phi_n, \psi_{k+1}, \ldots, \psi_n] \Longrightarrow \psi)$$

Isabelle computes the substitution s by *higher-order unification* [1].

If theorems are viewed as derived rules, resolution corresponds to forward proof. Backward proof is obtained by a change of perspective: theorem $[\psi_1, \ldots, \psi_n] \Longrightarrow \psi$ can be read

as an intermediate state in a backward proof where ψ is the overall goal and the ψ_i are the subgoals yet to be proved. In this case, resolution with rule $[\phi_1, \ldots, \phi_m] \Longrightarrow \phi$ corresponds to Prolog-style backward chaining. The initial state of a backwards proof is the trivially valid implication $\psi \Longrightarrow \psi$. Identifying proof states with theorems means that intermediate proof states are also valid theorems, albeit hypothetical ones.

Due to the presence of universal quantifiers and the possibility of natural deduction style proofs, resolution may also have to "lift" one rule over the quantifiers and local assumptions of the other one [5].

3.2 Tactics

Tactics are the functional programmer's answer to the tedium of single-step theorem proving. They combine arbitrary algorithmic sequences of proof steps (and searches) into a single function. Since Isabelle identifies proof states with theorems, tactics are simply functions over theorems. To allow for backtracking, tactics are in fact functions from theorems to streams of theorems. Backtracking can occur over the choice of rule, like in Prolog, and also over the choice of unifier (because of higher-order unification).

Tactics can be written from scratch or can be assembled from existing tactics with tacticals like THEN, ORELSE, REPEAT, DEPTH_FIRST, BEST_FIRST, etc. The tactic writer is supplied with a rich language of control structures, including search strategies.

Isabelle comes with several generic packages for writing tactics. There are two packages to support rewriting with user-defined object-level equalities; more generally, they can rewrite with equivalence relations and reduction systems. Another package supports classical predicate calculus reasoning, and can prove many of the problems in Pelletier [8]; this same package supports automated reasoning in set theory.

4 General information

Isabelle is written in Standard ML and should run on any Standard ML system. Interaction with Isabelle is through ML. Logics, theorems and tactics are ML values; they are manipulated by calling ML functions for extending a logic, resolving two theorems, applying a tactic, or whatever.

Isabelle comes with 8 different logics, including LCF and some modal logics. The most substantially developed logics are first-order logic, set theory, and higher-order logic. Many non-trivial theorems have been proved in them, including the Schröder-Bernstein theorem, the well-founded recursion theorem, and soundness and completeness of propositional logic.

Isabelle can be obtained by anonymous ftp from `ftp.cl.cam.ac.uk`. Using binary mode, get the file `isabelle.tar.Z` from directory `ml`. The LaTeX-sources for the manual [7] are included.

References

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

[2] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[3] T. Nipkow. Order-sorted polymorphism in Isabelle. In G. Huet, G. Plotkin, and C. Jones, editors, *Proc. 2nd Workshop on Logical Frameworks*, pages 307–321, 1991.

[4] T. Nipkow and G. Snelting. Type classes and overloading resolution via order-sorted unification. In *Proc. 5th ACM Conf. Functional Programming Languages and Computer Architecture*, pages 1–14. LNCS 523, 1991.

[5] L. C. Paulson. The foundation of a generic theorem prover. *J. Automated Reasoning*, 5:363–397, 1989.

[6] L. C. Paulson. Isabelle: The next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–385. Academic Press, 1990.

[7] L. C. Paulson and T. Nipkow. Isabelle tutorial and user's manual. Technical Report 189, University of Cambridge, Computer Laboratory, 1990.

[8] F. J. Pelletier. Seventy-five problems for testing automatic theorem provers. *J. Automated Reasoning*, 2:191–216, 1986. Errata, JAR 4 (1988), 236–236.