View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Elsevier - Publisher Connector

ELSEVIER

Available online at www.sciencedirect.com

Information and Computation 204 (2006) 1575–1596

Information and Computation

www.elsevier.com/locate/ic

Automation for interactive proof: First prototype

Jia Meng, Claire Quigley, Lawrence C. Paulson *

Computer Laboratory, University of Cambridge, UK

Received 1 January 2005
Available online 30 June 2006

Abstract

Interactive theorem provers require too much effort from their users. We have been developing a system in which Isabelle users obtain automatic support from automatic theorem provers (ATPs) such as Vampire and SPASS. An ATP is invoked at suitable points in the interactive session, and any proof found is given to the user in a window displaying an Isar proof script. There are numerous differences between Isabelle (polymorphic higher-order logic with type classes, natural deduction rule format) and classical ATPs (first-order, untyped, and clause form). Many of these differences have been bridged, and a working prototype that uses background processes already provides much of the desired functionality.
© 2006 Elsevier Inc. All rights reserved.

1. Introduction

Automatic theorem provers (ATPs) such as Vampire [19], which work by resolution, are impressive in their power. Interactive proof tools such as Isabelle [14] and PVS [5] provide much less automation; proofs require substantial user effort. However, interactive tools are better suited for verification projects. They admit complicated definitions and specifications, including recursive definitions of types, functions, and relations.

An obvious step is to gain the best of both worlds by integrating resolution provers with interactive ones, but many complications make this task difficult. Most automatic theorem provers

* Corresponding author. Fax: +1 44 1223 334678.
E-mail address: lcp@cl.cam.ac.uk (L.C. Paulson).

0890-5401/$ - see front matter © 2006 Elsevier Inc. All rights reserved.
doi:10.1016/j.ic.2005.05.010

work in first-order logic, while interactive provers typically use higher-order logic. Most resolution theorem provers are untyped, while interactive provers often have complicated type systems. Resolution theorem provers are designed to handle one-shot problems, while interactive provers are designed to support lengthy developments. Resolution provers are designed to run for minutes or hours, while the user of an interactive tool expects to get a response in seconds.

Our approach integrating the two types of system is based on a few simple principles. The guiding idea is that user interaction should be minimal. The system should invoke automatic provers spontaneously or in response to a trivial gesture such as a mouse click. These proof attempts should run in the background, not disturbing the user unless a proof is found. Proofs should refer to a large library of known lemmas: users should not have to select the relevant ones. The automatic prover should not be trusted; instead, proofs should be translated back into the formalism of the interactive prover. Proofs should be delivered in source form to the user, who can simply paste them into her proof script.

Governed by these principles, we have taken a systematic approach to reconciling the differences between an interactive tool (Isabelle) and automatic theorem provers (Vampire, SPASS). We have simply enumerated the differences—types, clause form, higher-order concepts, etc.—and dealt with each one in turn. The implementation is essentially complete, although still in need of many refinements to make it generally usable.

There is much related work. Many others have attempted to integrate interactive and automatic provers.

- Coq has been integrated with Bliksem [4].
- HOL has integrated with various first-order provers, including Gandalf [9] and Metis [11], the latter designed specifically for that integration.
- Isabelle has been integrated with a purpose-built prover, `blast` [17].
- KIV has been integrated with a tableau prover, $3T^AP$ [1].

Closest to our conception is Ωmega [21]. It shares with our work the idea that automatic provers can run in the background without being invoked by the user. However, there are also some important differences between the two projects. Ωmega is a novel architecture, using proof planning to invoke external reasoners. The objective of the Ωmega project is to identify techniques that can assist mathematicians. Isabelle is an established verification tool with many users and a huge amount of already formalized material. Our objective is to strengthen our existing framework rather than to create a new one. The only external reasoners we consider are resolution provers, and from this narrow focus we hope to provide the best possible integration.

The rest of the paper is as follows. We begin by describing Isabelle (Section 2) and our target resolution provers, Vampire and SPASS (Section 3). We describe how to formalize Isabelle's type system in first-order logic (Section 4) and how to translate lemmas and goals from Isabelle's higher-order logic into clauses (Section 5). We describe the overall process management framework (Section 6) and the two stages of proof reconstruction: how we check that the proof works in Isabelle (Section 7) and how we turn it into a proof script (Section 8). We describe the prototype's current status (Section 9) and finally give brief conclusions (Section 10).

2. Isabelle

Isabelle [14] is an interactive proof tool. Like others based on the LCF architecture, it allows proofs to be constructed only within a small kernel, which defines the basic inference rules. All decision procedures and other proof mechanisms must ultimately reduce their deductions to basic inference rules and axioms. Such an architecture makes proof procedures more difficult to implement, but it greatly improves their reliability. A difference between Isabelle and other LCF-based provers is that Isabelle's built-in logic, the *meta-logic*, is intended only for the formalization of other logics, the *object-logics*.

The LCF approach uses type-checking of the underlying programming language (typically a dialect of ML) to enforce soundness. The definition of `thm`, the abstract type of theorems, constitutes the inference kernel. A value of type `thm` can only be constructed by applying inference rules (which are built-in functions with types such as `thm -> thm`) ultimately to axioms (typically constants of type `thm`). Because ML's type-checker prevents arbitrary formulae from being assigned type `thm`, any expression having this type represents a correct proof.

Isabelle is *generic*: it supports a wide range of formalisms. The most important object-logic is higher-order logic (HOL), but several others are available, including Zermelo–Fraenkel set theory (ZF) [18]. The version of higher-order logic in Isabelle has polymorphic types (here we mean logical types, not ML types). Unlike other implementations of HOL, Isabelle also provides the concept of *axiomatic type class*: a collection of types, each possessing a specified set of operations that satisfy specified axioms. Typical type classes include *partially ordered set* and *ordered field*. A type class is open-ended: any new type that meets the specification can be admitted to the class. Unlike PVS, Isabelle does not support predicate subtyping, where the combination of a type and a predicate yields a new type.

Isabelle's most basic inference mechanism is a form of Horn clause resolution, which must not be confused with the resolution performed by the automatic provers we invoke. A typical Isabelle theorem is a nested implication of the form

$$\phi_1 \Longrightarrow (\cdots(\phi_n \Longrightarrow \theta)\cdots),$$

with implicit universal quantification over its free variables. This theorem is abbreviated as

$$[\![\phi_1; \cdots \phi_n]\!] \Longrightarrow \theta,$$

and represents the object-logic inference rule

$$\frac{\phi_1 \quad \cdots \quad \phi_n}{\theta.}$$

Isabelle's resolution combines such rules in the obvious way. For example, the two Isabelle theorems $\phi \Longrightarrow \theta$ and $\theta \Longrightarrow \psi$ can be resolved to obtain $\phi \Longrightarrow \psi$; more generally, either clause may have multiple negative literals, and the complementary literals may undergo unification. [1] The Isabelle proof state has much in common with a Prolog goal clause. When a user performs single-step

[1] Isabelle uses higher-order unification [8].

proof checking, applying some rule to reduce a goal to subgoals, she is actually performing Isabelle resolution between that rule and the proof state [16].

Isabelle provides a variety of automatic tools—known as *tactics*—that can be used to construct proofs. They include

- `simp`, which simplifies subgoals using rewriting and decision procedures,
- `blast`, which proves subgoals by classical reasoning (see below),
- `auto`, which combines simplification and classical reasoning,
- and other classical reasoning tools such as `fast` and `clarify`.

The rewriting engine and arithmetic decision procedures are similar to those found in competing systems such as PVS [5] and HOL [7]. Unique to Isabelle is its generic classical reasoner, which searches for proofs using tableau methods [17]. Within the realm of first-order logic, the classical reasoner is much weaker than resolution theorem provers. However, the classical reasoner is not restricted to first-order logic: it uses any supplied collection of lemmas to perform forward or backward chaining, governed by depth-first iterative deepening. The classical reasoner can prove many theorems that are difficult for most automatic provers, such as $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$, and theorems that cannot easily be expressed in first-order logic at all, such as

$$\left(\bigcup_{i \in I \cup J} A_i\right) = \left(\bigcup_{i \in I} A_i\right) \cup \left(\bigcup_{i \in J} A_i\right).$$

The classical reasoner's other great advantage is that it can utilize a large set of lemmas without suffering a combinatorial explosion. Although certain proofs do require the user to name crucial lemmas, hundreds of other lemmas are available to the classical reasoner at all times. These are lemmas that were previously designated as being useful for classical reasoning, and they constitute a knowledge base for the user's application domain. Typically omitted from this knowledge base are transitivity laws and similar lemmas that would blow up the search space. While integrating Isabelle with automatic theorem provers, we have sought to preserve this advantage: the user should only have to identify a few crucial lemmas, while the resolution search automatically finds other needed facts from the knowledge base. Separating relevant facts from irrelevant ones is a task that resolution theorem provers find difficult [13].

Isabelle provides two proof styles, linear and structured. Linear proofs resemble the tactic scripts of HOL and PVS, and consist of commands that manipulate the proof state. Structured proofs (of the Isar language [26]) are an attempt at formalizing mathematical style. We have concentrated on supporting structured proofs, chiefly because they work on different subgoals independently. In the linear style, an Isabelle tactic can affect all subgoals, so they are likely to change before an automatic prover manages to prove any of them. Our methods are thus applicable to traditional LCF-style systems such as HOL, where even linear proof scripts respect the goal-subgoal tree structure.

2.1. Notational remark

This paper follows the usual convention in the logic and theorem proving communities, where the scope of quantifiers is as small as possible. For example, $\forall x\, A \wedge B$ abbreviates $(\forall x\, A) \wedge B$ rather

than $\forall x\ (A \wedge B)$. This convention differs from that used in interactive tools such as Isabelle, where the quantifier syntax includes a dot (as in $\forall x.\ A \wedge B$) and the scope of quantifiers extends to the right as far as possible.

3. Automatic theorem provers

Most automatic theorem provers implement some form of resolution. They work in untyped first-order logic; they use clause form; their main inference rules are resolution, factoring, and paramodulation; they output a summary of the proof found. The better ones utilize advanced indexing data structures and subsumption, and they can prove exceptionally complex theorems.

We hope that integration with a resolution prover will equip Isabelle with an effective combination of equational and classical reasoning, via paramodulation. One of the most popular Isabelle tactics, `auto`, performs a naive combination of rewriting and classical reasoning. It begins by performing obvious classical reasoning steps (such as reducing the goal $A \wedge B$ to the separate subgoals A and B interleaved with simplification; it attempts to prove the resulting subgoals using classical reasoning. What it cannot do is interleave equational reasoning with search as paramodulation does.

We performed much experimentation using Vampire [19], which has repeatedly won at CASC (the CADE ATP System Competition). [2] We have even used a version of Vampire modified (by its developers) to support forward and backward inference, as Isabelle does. However, we have also used SPASS [24], largely because its proof output is easier to interpret. We intend that our work should be applicable to most resolution provers. Standardization of inputs and outputs would make this objective attainable. For input, the tptp2X utility can translate from TPTP (thousands of problems for theorem provers [22]) format into the input languages of all major resolution provers. Output of proofs is more problematical because systems differ in their inference rules, simplification steps, and problem transformations. We hope that many theorem provers will eventually produce complete, explicit proofs in TSTP (thousands of solutions from theorem provers [23]) format.

4. Coding Isabelle types in first-order logic

Isabelle/HOL implements classical higher-order logic, whose complex type system is not supported by standard ATPs. Therefore, we need to model Isabelle's type system within first-order logic. The purpose of encoding type information is to ensure soundness, but it also turns out to reduce the ATP's search space: theorems will take part in proof attempts only if the types are appropriate.

4.1. Type classes

In Isabelle, a *type class* is a set of types for which certain operations are defined [25]. An *axiomatic* type class has a set of axioms that must be satisfied by its *instances*, namely the types belonging to that class. If a type τ belongs to a class C then it is written as $\tau :: C$. A type class C is a *subclass*

[2] See http://www.cs.miami.edu/~tptp/CASC/.

of another type class D provided all axioms of D can be proved in C; if a type τ is an instance of C then it is an instance of D as well. Furthermore, a type class may have more than one direct superclass. If C is a subclass of both D_1 and D_2, then C is a subset of the intersection of D_1 and D_2. The intersection of type classes is called a *sort*.

We formalize types as first-order terms. For each type class, we introduce a unary predicate. If a type τ is an instance of a class C, then $C(\tau)$ will be true. The subclass relation "C is a subclass of D" is expressed by the universal implication $\forall\tau\,[C(\tau) \rightarrow D(\tau)]$. Similarly, the sort constraint $\tau :: C_1, \ldots, C_n$ is expressed by the conjunction $C_1(\tau) \wedge \cdots \wedge C_n(\tau)$.

Isabelle provides compound types through the use of type constructors. Each type constructor has one or more *arities*, which describes the type class information of the arguments and the result of this type constructor. For instance, the list type constructor `list` may have an arity written as

```
list :: (type) order
```

This means, if the argument of `list` is an instance of class `type`, then the resulting type (of lists) belongs to class `order`. (It can be realized by the prefix ordering on lists, which does not require an ordering to be defined on list elements.) We formalize this arity by the first-order formula

$$\forall\tau\,[\mathit{type}(\tau) \rightarrow \mathit{order}(\mathit{list}(\tau))].$$

In general, for a type constructor op, each arity of the form

$$op :: (C_1, \ldots, C_n)C$$

is translated into a Horn clause

$$\forall\tau_1, \ldots, \tau_n\,[C_1(\tau_1) \wedge \cdots \wedge C_n(\tau_n) \rightarrow C(op(\tau_1, \ldots, \tau_n))].$$

4.2. Embedding type information in clauses

Isabelle's predicates and functions are typed—many of them are polymorphic—and this information must be conveyed to the ATP. We embed the types of predicates and functions, formalized as shown above, in clauses:

- A function or predicate (other than equality) takes its type as an additional argument.
- Equality is discussed below. However, equalities between boolean values—which are legal in HOL—are simply replaced by two implications.
- Any type class constraints on type variables occurring in a clause are included as preconditions, in the form of additional negative literals.

The equality predicate requires special treatment because it is built into most ATPs to support inference rules such as paramodulation. It takes two arguments only, so adding a third argument is out of the question. We briefly experimented with embedding type information into the two-argument equality relation by pairing each operand with its type. Thus $A = B$ became

$$\mathit{equal}(\mathit{typeinfo}(A, \tau), \mathit{typeinfo}(B, \tau)),$$

where τ is the type of A and B and *typeinfo* effectively behaves as a pairing function. To make equality reasoning work, we included an axiom for stripping away type information:

$$equal(typeinfo(A, \tau), typeinfo(B, \tau)) \rightarrow equal(A, B).$$

This approach is useful in certain situations, such as proving the set equality $A = B$ by separate consideration of the cases $A \subseteq B$ and $B \subseteq A$, or proving the integer equality $i = j$ by separate consideration of the cases $i \leqslant j$ and $j \leqslant i$. It prevents the (untyped) ATP from attempting to prove absurdities like $A \leqslant B$ or $x \subseteq y$. However, this representation harms performance, probably because it complicates equality reasoning. We now simply regard equality as untyped.

Isabelle's type information exists in various places and needs to be extracted and converted to first-order format. The type information in Isabelle goals and lemmas are translated into additional clauses. Type information such as subclass relationships are global facts, and hence are converted to axiom clauses. The type information from Isabelle rules are translated into extra literals of the rule clause. Unfortunately, these extra clauses and literals complicate proof reconstruction: they represent Isabelle's type system rather than actual Isabelle inferences. When reconstructing a proof, we have to identify and remove subproofs that perform type checking.

4.3. Examples

Here are two examples taken from Isabelle-generated TPTP files in order to illustrate the formalization of types. To improve readability, we have reformatted the output, simplifying the computer-generated names. The first example expresses the theorem $A \subseteq B \wedge B \subseteq A \rightarrow A = B$. Isabelle identifies the subset relation with the overloaded constant $\leqslant$, whose third argument below restricts it to sets.

```
input_clause(cls_0_Set_subset_antisym, axiom,
 [++equal(V_A, V_B),
  --lessequals(V_B, V_A, fun(set(T), fun(set(T), bool))),
  --lessequals(V_A, V_B, fun(set(T), fun(set(T), bool)))]).
```

The second example expresses the theorem $\neg(x < x)$. It refers to the overloaded constant $<$. Its type argument, together with the clause's first literal, restricts it to elements whose type belongs to the type class of partial orderings, `order`.

```
input_clause(cls_0_Orderings_order_less_irrefl, axiom,
 [--class_order(T),
  --less(V_x, V_x, fun(T, fun(T, bool)))]).
```

5. Preparing Isabelle goals for ATPs

Recall that invocation of the ATP should be invisible to the user. She works normally, refining goals to subgoals; proof attempts on open subgoals take place in the background, and any successes are reported. The automatic prover must receive

- existing lemmas,
- assumptions local to the current proof, and
- the subgoals to be proved.

The automatic theorem prover must be given clause form, with Isabelle's conventions for natural deduction removed. Isabelle types must also be removed, as described above. Finally, these translations must be integrated with Isabelle's interactive proof language.

5.1. Translating existing lemmas to clause form

Lemmas that were previously proved in Isabelle need to be converted to clause normal form (CNF). This transformation must be performed using Isabelle inferences in order to allow proof reconstruction. As a result, our CNF transformation is a function of type `thm -> thm`, which in LCF-based provers is the type of derived inference rules. This CNF transformation is performed by two functions.

- `skolem_axiom` converts an Isabelle theorem into negation normal form (NNF), Skolemizes it and finally removes all existential variables.
- `cnf_axiom` performs the same steps, then converts the result into conjunction normal form (CNF), represented by a list of clauses. Each clause has type `thm`, and is therefore an Isabelle theorem.

Skolemization takes several steps and treats lemmas differently from negated conjectures. It begins with a formula whose outermost universal quantifiers have been discarded. It then moves every existential quantifier to the front of the theorem by applying the rewrite rule

$$(\forall x\, \exists y\, P(x, y)) \iff (\exists f\, \forall x\, P(x, f(x))).$$

This equivalence expresses the axiom of choice, which appears to be necessary when performing Skolemization by inference. [3] A single application of this equivalence yields a function of one variable. Repeated application—to move an existential variable past several universal variables—results in a function of all of those variables. Rewriting with this equivalence, along with others to extract existential quantifiers from conjunctions, disjunctions, etc., yields a formula in which all existential quantifiers are lined up at the front. If we were to reinstate the outermost universal quantifiers, we would obtain a $\forall\exists$-prefix.

These existential quantifiers must now be removed altogether. The procedure depends upon whether the clauses have been produced from the negated conjecture or from existing theorems. Skolemization of the negated conjecture is easy: we transform an Isabelle goal (the conjecture) into one that is headed by existential quantifiers. These can be removed using the standard treatment of existentially quantified assumptions, namely application of the rule of $\exists$-elimination. Skolemization of theorems is harder: we transform an Isabelle theorem into one that is headed by existential quantifiers. Removing those quantifiers requires a further use of the axiom of choice, in the form of Hilbert's ϵ-operator. The term $\epsilon x\, P(x)$ denotes some value x such that $P(x)$ is true, if such exists;

[3] In simple cases, we can avoid the use of the axiom of choice by instead pulling out and discarding the quantifier $\forall x$ whenever we encounter a formula of the form $\forall x\, \exists y\, P(x, y)$, at the cost of substantially increased proof complexity [3].

otherwise, it denotes any value of the appropriate type. If we have transformed a lemma into the form $\exists x\, P(x)$, then we may conclude $P(\epsilon x\, P(x))$; this inference is trivial in Isabelle, using the basic properties of Hilbert's ϵ-operator. This step can be repeated for each existential quantifier.

For instance, the Isabelle lemma `subsetI` expresses the natural deduction rule for introducing the subset relation: to show $A \subseteq B$, it suffices to show that for arbitrary x, if $x \in A$ then $x \in B$. This lemma is equivalent to the first-order formula

$$\forall x\, (x \in A \to x \in B) \to A \subseteq B.$$

The variables A and B are implicitly universal, but since the outermost universal quantifiers are discarded, there is no prefix $\forall AB$. Transforming this formula into NNF and Skolemizing yields

$$\exists x\, [(x \in A \wedge x \notin B) \vee A \subseteq B].$$

The replacement of existential variables by ϵ-terms yields a large formula.

$$\underbrace{\epsilon x\, [(x \in A \wedge x \notin B) \vee A \subseteq B]}_{\epsilon\text{-}term} \in A \wedge \underbrace{\epsilon x\, [(x \in A \wedge x \notin B) \vee A \subseteq B]}_{\epsilon\text{-}term} \notin B \vee A \subseteq B$$

The two ϵ-terms are identical, representing the eliminated $\exists x$. It would obviously be preferable to define a Skolem function by

$$f(A, B) \stackrel{\text{def}}{=} \epsilon x\, [(x \in A \wedge x \notin B) \vee A \subseteq B].$$

because the result would be more compact:

$$(f(A, B) \in A \wedge f(A, B) \notin B) \vee A \subseteq B$$

Causing such function declarations to be generated presents some practical problems: Isabelle's concept of "theory" makes it difficult to declare constants on the fly. For the present, proof reconstruction in Isabelle uses the versions containing ϵ-terms. Efficiency concerns will probably force us to replace these by automatically-defined Skolem functions.

Obviously, the ϵ-terms must be replaced by proper Skolem terms such as $f(A, B)$ before the clauses are delivered to an ATP. For each ϵ-term, we generate a unique Skolem term. The universally quantified variables that cover the scope of a Skolem term are exactly those variables that appear inside the ϵ-term. Therefore it is enough to inspect each ϵ-term separately in generating a Skolem term. This step (which is not performed by Isabelle inferences) yields clauses that are ready to be delivered to ATPs.

5.2. Preprocessing of elimination rules

The code that converts formulas into clauses raises an exception if it detects that the formula is higher-order, since these clauses will be used by first-order ATPs. However, one class of apparently higher-order theorems can be converted. Many Isabelle lemmas are expressed to resemble the introduction and elimination rules of natural deduction [16]. The format of Isabelle elimination rules can be expressed directly in higher-order logic:

$$\forall P\,[A \to (\forall \mathbf{x}_1\, \mathbf{B}_1 \to P) \to \cdots \to (\forall \mathbf{x}_n\, \mathbf{B}_n \to P) \to P]$$

Here, A is a formula that contains an operator to be eliminated. Each $\mathbf{x}_i$ is a list of universally quantified variables, while $\mathbf{B}_1, \ldots, \mathbf{B}_n$ are lists of formulae, regarded as conjunctions. (Any of the lists may be empty.) Finally, P is a predicate variable.

Clearly any theorem of this form should be transformed into an equivalent first-order formula, removing the predicate variable, before we transform it into CNF. For an elimination rule like this, we transform it to

$$A \to (\exists \mathbf{x}_1\, \mathbf{B}_1 \vee \cdots \vee \exists \mathbf{x}_n\, \mathbf{B}_n).$$

If $n = 0$, the elimination rule is simply $\neg A$.

The transformation described above is difficult to perform simply by applying Isabelle inference rules. An alternative approach was adopted that still ensures correctness. This approach can be divided into two major steps.

(1) From the elimination rule, we construct an Isabelle term that represents the first-order formula equivalent to the rule. This is straightforward programming.
(2) We then invoke an Isabelle function that takes a term and a tactic, yielding an Isabelle theorem. The tactic applies the elimination rule under consideration, then delivers the resulting subgoals to Isabelle's classical reasoner.

As an example, consider the elimination rule `UnionE`. Intuitively, it says that if $A \in \bigcup C$, then there is some x such that $A \in x$ and $x \in C$. It can be expressed in higher-order logic as follows:

$$\forall P\, A\, C\,[A \in \bigcup C \to \forall x\,(A \in x \wedge x \in C \to P) \to P].$$

The result of our transformation is a first-order theorem:

$$\forall A\, C\,[A \in \bigcup C \to \exists x\,(A \in x \wedge x \in C)].$$

This formula is finally converted to CNF as two clauses

$$A \notin \bigcup C \vee [\epsilon x\,[A \notin \bigcup C \vee (x \in C \wedge A \in x)]] \in C,$$

and

$$A \notin \bigcup C \vee A \in \epsilon x\,[A \notin \bigcup C \vee (x \in C \wedge A \in x)],$$

where A and C are implicitly universally quantified. Since the procedure above is carried out by Isabelle inferences, each clause has the type `thm`, which means it is an Isabelle theorem. The versions of these clauses given to the ATP will replace $\epsilon x\,(A \notin \bigcup C \vee x \in C \wedge A \in x)$ by a Skolem term of the form $f(A, C)$.

5.3. A generic clause datatype

Although ATPs accept clauses in a textual format, it is useful to have an internal representation of clauses that captures all of the essential information. We define type `clause` to represent clauses derived from Isabelle lemmas and goals. A `clause` contains several fields.

- A unique identifier. If the clause is derived from an Isabelle rule, then the rule's name is recorded as well.
- An indication of whether the clause should be labelled as an axiom or negated conjecture clause. This distinction is important for heuristics such as SOS, which attempt to focus the proof search on the conjecture. (Recall discussion in Section 3.)
- A list of literals in this clause. Since, our inputs to ATPs are typed, predicates' and functions' types are included.
- Additional type information. This includes type classes of type variables that occur in the clause.

During the conversion of an Isabelle lemma or negated subgoal into the `clause` datatype, type information is gathered and stored in the form of additional literals, as described in Section 4.2 above. A first-order Horn clause represents the arity of each type constructor. Every such clause is represented by a type `arityClause`, which is analogous to `clause` but simpler. Fields included in `arityClause` are

- a unique identifier,
- a positive literal (the type class of the type constructor's result), and
- a list of negative literals (the type classes of the type constructor's arguments).

A similar datatype, `classrelClause`, stores information about the subclass relationship between type classes. Each such relationship is formalized as a Horn clause.

The clause datatypes defined above can easily be translated to any ATP-specific clause syntax. We have implemented the conversion to the widely-used TPTP format. By default, the clauses include type information, as described in the previous section. Users can modify this behaviour (for instance, on including type information for equalities) by setting boolean flags.

5.4. Summary of preliminary experiments

Before undertaking any implementation, we carried out a series of experiments in order to examine whether our approach would be practical [12,13]. The experiments consisted of taking basic tactic invocations (the tactics were *`blast`*, *`fast`*, *`clarify`*, *`auto`*, and *`simp`*) from existing proofs.

We attempted to reproduce them using Vampire and SPASS, with a time limit of 60 s per proof. In each case, clauses from the negated conjecture were combined with axiom clauses obtained from the default classical and simplifier rules: the rules that Isabelle tactics such as `auto` would use. In some of the examples, we used *formula renaming* [15] before the CNF transformation in order to minimize the number of clauses.

Overall, our experiments showed that our methods of translating ZF and HOL into first-order clauses were effective. Most of the goals that were proved by Isabelle's `blast` and `auto` were proved by Vampire and SPASS. In addition, the inclusion of type information in HOL was shown to be important: rules involving polymorphic operators did not increase the search space, probably because the type information constrained the search.

Since, the aim of this integration is to improve automation, it is important to know whether Vampire and SPASS can prove goals that were not proved by Isabelle's automatic tools. For this, we took 15 lemmas that were proved in Isabelle by a short sequence of proof steps and gave them to Vampire. Vampire proved ten of these.

However, our experiments also showed that having a large number of axioms—which is inevitable if we include the default classical and simplifier rules—often overwhelms ATPs. Many of those failed proof goals could be proved if we removed some of the irrelevant axioms. Thus, we have had advance warning of the need to implement heuristics for coping with redundant information.

6. Architecture overview

Fig. 1 presents a block diagram of the system. Isabelle interacts with the user through a generic interface, Proof General. In order to send subgoals to resolution provers while the user continues with what she is doing, a *watcher* process is created. This is another Isabelle process, running concurrently with the Isabelle process the user interacts with, and which can communicate with that process by two Unix pipes, one for input, the other for output.

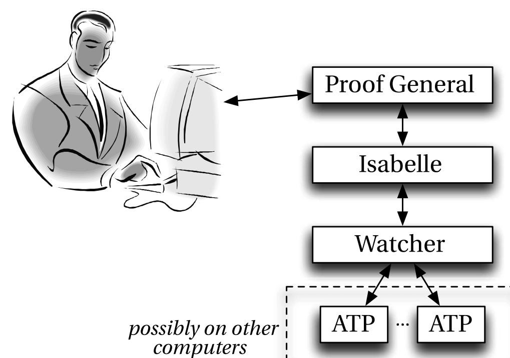

Fig. 1. System architecture.

POSIX is an international standard that defines how an application obtains the basic services of an operating system. [4] The watcher calls the POSIX functions specified in the Standard ML Basis Library [6]. Both Poly/ML and Standard ML of New Jersey implement these functions.

6.1. Watcher process

From the main Isabelle process, a watcher process can be created with the command

```
val (watcherIn, watcherOut, watcherPid) = createWatcher();
```

This uses the POSIX `fork()` command to create an identical copy of the original Isabelle process, and then sets up communication pipes between the two. The process runs concurrently with the original process, and the user should be unaware of its existence unless it finds a proof.

Once the watcher has been created, it polls its input pipe every 100 ms for commands from the main Isabelle process. If it sees a command to prove a subgoal, it calls an automatic prover using a modified version of the `execute` function from the Unix structure. This creates a process running the resolution prover on the desired subgoal.

Each time a prover is called, a data structure containing information on that process is added to a list in the watcher. This data structure contains

- the process ID of the new process,
- the name of the prover,
- the file containing the subgoal to be proved, and
- the file descriptors of the input and output pipes connecting the watcher to the resolution proof process.

All processes in the watcher's list are polled for output, each poll timing out after 100 ms. If the watcher sees that a child has responded, it checks to see whether a proof has been found and, if so, reads in the proof and carries out a reconstruction. It then signals the Isabelle process that a proof has been found and transferring the commands necessary for reconstruction via the output pipe. Once a proof has been found, reconstructed, and transmitted to Isabelle, the information for that proof process is removed from the watcher's list.

Resolution provers can run either on the same machine as the main Isabelle process and watcher, or remotely using `ssh`, the secure shell. Once an `ssh` key has been set up on the calling machine, `ssh` connections can be authenticated automatically.

6.2. Interaction with the user

The standard user interface for Isabelle is Proof General [2]. Based on the Emacs text editor, Proof General connects the evaluation of a proof to the editing of a proof script containing a series of commands. Once a command has been executed by Isabelle, the corresponding line in the script is locked, preventing any modification.

[4] See http://www.pasc.org/.

Communication with Proof General takes place in three buffers.

- The *script buffer* contains the text of the proof script; unlocked portions can be edited.
- The *goals buffer* displays the current list of subgoals to be proved.
- The *response buffer* displays output from the proof tool.

Users tell the system to execute parts of the proof script by clicking an icon at the top of the Proof General window. In order to provide for output from an ATP via the watcher process, we have added a *resolution-response* buffer to Proof General. This works like the *trace* buffer, which displays any tracing output. Both of these buffers, along with procedures to print warning messages in the response buffer, rely on the output from the proof tool being enclosed in *urgent response* characters. These are non-printing characters that direct Proof General to do something special with this output, instead of displaying it in the response buffer.

What Proof General does next with the output depends on the character following the urgent response character. In the case of an ATP-based proof, this is another non-printing character telling Proof General to put the output in the resolution-response buffer, highlight it, and bring the buffer to the front. Fig. 2 shows an example. The upper window holds an interaction buffer where the user has entered a conjecture followed by the command `proof -`. A proof found by SPASS is displayed, for the user to insert into her proof script. Note that the reconstructed proof emulates resolution rather than attempting to be intuitive.

6.3. Interaction between proof general and ATPs

Since, the aim of our integration is to reduce interaction, the invocation of ATPs should be as easy as possible. In our current prototype, invocation is automatic when Isabelle enters a mode in which subgoals are available to be proved. [5] Experience suggests that we should adopt a new mode of interaction, ATPs are invoked by a single mouse gesture. Explicit invocation prevents the waste of unnecessary ATP processes being spawned, and it keeps the user in control. However invocation takes place, the following steps are taken:

- First, a file containing clauses derived from each subgoal is produced.
- Next, a watcher process is created and a request to call an ATP passed to it, consisting of
  - the ATP to be called, with any necessary settings
  - the path and filename of the input file, and
  - a textual representation of the negated, Skolemized goal (including types).

  This initiates a call to the ATP and translation of any proof that is found, and will be discussed in Section 7.
- Finally, the Isabelle process responds to a POSIX signal from the watcher process by reading the proof found, transforming it into an Isar script, and sending it to Proof General for display in the resolution-response buffer. This will be described in Section 8.

[5] Precisely, this occurs at the entry to *State* mode, which signals the start of a structured proof.

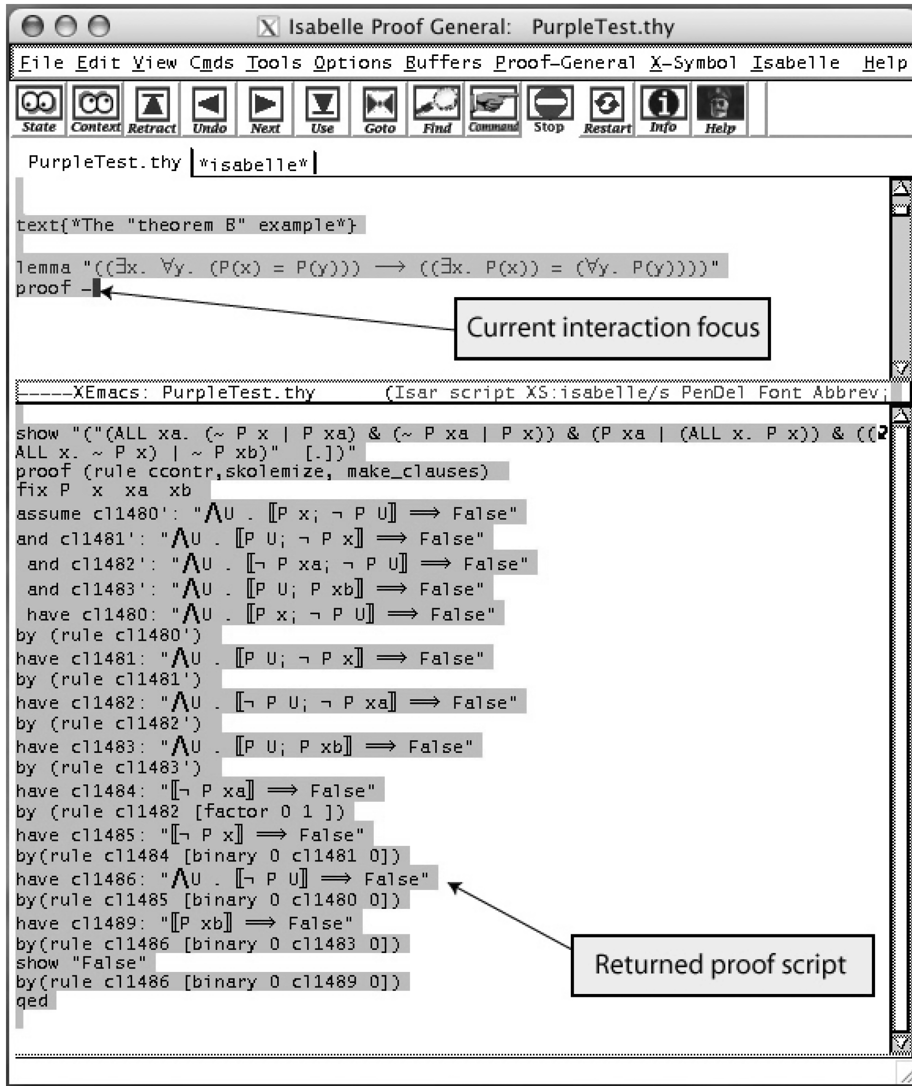

Fig. 2. Proof General returning a proof script (shaded) to the user.

7. Finding and translating an ATP proof

To illustrate the process of finding and translating an ATP proof, we follow the progress of the proof of the following formula using the SPASS prover.

$$\exists x\, \forall y\, (P(x) = P(y)) \longrightarrow (\exists x\, P(x) = \forall y\, P(y)).$$

We begin by transforming this subgoal into negation normal form, followed by Skolemization and conversion to clauses. This allows us to generate ATP-specific problem files for each subgoal.

The pathnames of these files, along with settings for the ATP to be called, are then sent to the watcher process, which in turn calls an automatic prover. At present, the choice of prover is determined by flag settings. If necessary, Isabelle uses tptp2X [22] to convert the problem file from TPTP format into the format required by the chosen prover.

Once a proof has been found by the resolution prover (Fig. 3), the watcher reads it in. Resolution proof steps are represented in the watcher by the following datatype:

```
datatype Proofstep =  Axiom
                     | Binary of (int * int) * (int * int)
                     | MRR of (int * int) * (int * int)
                     | Factor of (int * int * int)
                     | Para of (int * int) * (int * int)
                     | Rewrite of (int * int * int)
                     | Unusedstep of unit
```

Inference rules currently emulated in Isabelle include binary resolution, matching resource resolution, factoring, paramodulation, and rewriting. The SPASS proof is now parsed, yielding a list:

```
val proof_steps =
   [(1, Axiom, ["P x", "~  P U"]), (3, Axiom, ["P U", "~  P x"]),
    (5, Axiom, ["~  P U", "~  P xa"]), (7, Axiom, ["P U", "P xb"]),
    (9, Factor (5, 0, 1), ["~  P xa"]),
    (10, Binary ((9, 0), (3, 0)), ["~  P x"]),
    (11, Binary ((10, 0), (1, 0)), ["~  P U"]),
    (12, Factor (7, 0, 1), ["P xb"]),
    (14, Binary ((11, 0), (12, 0)), [])]
```

Each element of the list contains

- the line number of the SPASS proof step,
- a proof step as an element of type `Proofstep`, and
- a list of strings representing the clause ordering in the SPASS proof.

The last element is necessary because SPASS reorders the literals in each clause; we must reorder the literals in the Isabelle clauses to match this. Additionally, all literals that were inserted to model

```
Here is a proof with depth 4, length 9 :
1[0:Inp] || v_P(tconst_fun(typ__asc39_a,tconst_bool),v_x)*+ ->
            v_P(tconst_fun(typ__asc39_a,tconst_bool),U)*.
3[0:Inp] || v_P(tconst_fun(typ__asc39_a,tconst_bool),U)*+ ->
            v_P(tconst_fun(typ__asc39_a,tconst_bool),v_x)*.
5[0:Inp] ||  -> v_P(tconst_fun(typ__asc39_a,tconst_bool),U)*
                v_P(tconst_fun(typ__asc39_a,tconst_bool),v_xa)*.
7[0:Inp] || v_P(tconst_fun(typ__asc39_a,tconst_bool),U)*+
            v_P(tconst_fun(typ__asc39_a,tconst_bool),v_xb)* -> .
9[0:Fac:5.0,5.1] ||  -> v_P(tconst_fun(typ__asc39_a,tconst_bool),v_xa)*.
10[0:Res:9.0,3.0] ||  -> v_P(tconst_fun(typ__asc39_a,tconst_bool),v_x)*.
11[0:Res:10.0,1.0] ||  -> v_P(tconst_fun(typ__asc39_a,tconst_bool),U)*.
12[0:Fac:7.0,7.1] || v_P(tconst_fun(typ__asc39_a,tconst_bool),v_xb)* -> .
14[0:Res:11.0,12.0] ||  -> .
```

Fig. 3. A proof returned by SPASS.

the Isabelle type system (recall the discussion in Section 4.2) are removed, leaving only the literals that appear in the Isabelle representation of the clause.

As the watcher starts off as an identical copy of the main Isabelle process, it is able to create clauses for each subgoal by the same methods as the calling program when it created the ATP problem files. A contradiction can be derived from this set of clauses by emulating the steps of the resolution proof. Hurd [10] describes a similar reconstruction procedure for HOL, which he applies to his first-order prover, Metis.

In addition to producing the desired theorem, the translation procedure returns a list of reconstruction steps. These consist of

- the line number,
- a proof step,
- an ordered list of the literals in the clause, and
- a list of the variables in the clause.

The watcher process transforms this list of reconstruction steps into a string:

```
val reconstr =
   "[P%x%xa%xb%]1OrigAxiom()[P x%  P U%][U%]3OrigAxiom()[P U%  P x%][U%]
     5OrigAxiom()[  P xa%  P U%][U%]7OrigAxiom()[P U%P xb%][U%]
     1Axiom()[P x%  P U%][U%]3Axiom()[P U%  P x%][U%]
     5Axiom()[  P U%  P xa%][U%]
     7Axiom()[P U%P xb%][U%]9Factor(5,0,1)[  P xa%][]
     10Binary((9,0),(3,0))[  P x%][]
     11Binary((10,0),(1,0))[  P U%][U%]12Factor(7,0,1)[P xb%][]
     14Binary((11,0),(12,0))[][]"
: string
```

This is sent, along with the string representation of the goal, to the Isabelle process. At the same time a POSIX signal is sent, causing the Isabelle process to interrupt what it is currently doing to deal with the information from the watcher. The need to interrupt the main Isabelle process is unfortunate, but this process is the only one that can communicate with Proof General.

8. Transforming the ATP proof to an Isar proof

In order to allow Isar scripts to express resolution proofs, we had to extend the Isar language with the inference rules used in resolution provers. In Isar, operations that transform theorems are called *attributes*. These new attributes could then be used to produce an Isar script from the reconstruction string obtained from the watcher process.

8.1. Resolution proof attributes for Isar

We extended the Isar proof language with the following attributes:

```
proof (rule ccontr,skolemize, make_clauses)
fix P  x  xa  xb
assume cl1': "⋀U . ⟦P x; ¬ P U⟧ ⟹ False"
and cl3': "⋀U . ⟦P U; ¬ P x⟧ ⟹ False"
and cl5': "⋀U . ⟦¬ P xa; ¬ P U⟧ ⟹ False"
and cl7': "⋀U . ⟦P U; P xb⟧ ⟹ False"
have cl1: "⋀U . ⟦P x; ¬ P U⟧ ⟹ False"
  by (rule cl1')
have cl3: "⋀U . ⟦P U; ¬ P x⟧ ⟹ False"
  by (rule cl3')
have cl5: "⋀U . ⟦¬ P U; ¬ P xa⟧ ⟹ False"
  by (rule cl5')
have cl7: "⋀U . ⟦P U; P xb⟧ ⟹ False"
  by (rule cl7')
have cl9: "⟦¬ P xa⟧  ⟹ False"
  by (rule cl5 [factor 0  1 ])
have cl10: "⟦¬ P x⟧  ⟹ False"
  by(rule cl9 [binary 0  cl3 0])
have cl11: "⋀U . ⟦¬ P U⟧ ⟹ False"
  by(rule cl10 [binary 0  cl1 0])
have cl12: "⟦P xb⟧  ⟹ False"
  by (rule cl7 [factor 0  1 ])
show "False"
  by(rule cl11 [binary 0  cl12 0])
qed
```

Fig. 4. Generated Isar proof script.

- **binary**, binary resolution
- **factor**, factoring
- **paramod**, paramodulation
- **demod**, demodulation (rewriting)

Each of these attributes operate on one or more already established facts in a proof. For example, the line

```
by (rule cl9 [binary 0 cl3 0])
```

resolves the first literal (numbered 0) of the fact labelled `cl9` with the first literal of the fact labelled `cl3`. The code for these operations is the same as that used to translate these operations inside the watcher process (Section 6.1).

8.2. Production of an Isar script

After the watcher has checked the correctness of the ATP proof by reconstructing it using Isabelle's inference rules, it generates a structured Isar proof. The user can insert this into her theory file, so that it can be reprocessed subsequently without the support of an ATP.

When the Isabelle process receives the appropriate POSIX signal, a handler function reads the proof reconstruction information from the watcher's output pipe, and parses the reconstruction steps from the reconstruction string.

The Isar proof (Fig. 4) begins with the line

```
proof (rule ccontr, skolemize, make_clauses).
```

This has the effect of applying the classical contradiction tactic to the goal, Skolemizing it and transforming it to a list of clauses. The first element of the reconstruction string is a list of variables that must be fixed, making them effectively constants. Most of them are Skolem functions.

```
fix P x xa xb
```

The rest of the reconstruction string is parsed into a list of elements of the form

(*clause number*, *proof step*, *literals*, *variables*).

We begin by assuming all the clauses produced by `make_clauses` that are not actually used in the resolution proof—Isar will complain otherwise—and then we assume the clauses that were used. The literals in each clause will be in the order given to them by Isabelle, so if SPASS has re-ordered the literals, we need to derive a new instance of the clause with the literals in the order SPASS expects. We then simply emulate the SPASS proof, each time forcing the literals into SPASS's ordering.

9. Current status

The architecture of the system is essentially complete. The implementation is partly complete, but there are major gaps and numerous details to be fixed.

9.1. Proof reconstruction issues

One source of problems has been the difficulty of interpreting the output of ATPs. (Hurd [9] has reported the same difficulty, with the Gandalf prover.) When SPASS is given a set of clauses, it reorders the literals in them and may preprocess and simplify them before attempting a proof, which may contain various simplification steps in addition to the normal inference rules. Coping with such details takes time, and reduces our ability to re-target the system to other ATPs.

Our integration makes previously-proved lemmas available for use in the resolution proofs. The same mechanism should work for facts proved locally within the current scope of the structured proof. We also have a mechanism for temporarily adding global theorems to the search. A declaration such as

```
note order_trans [intro]
```

adds the transitivity of $\leqslant$ to the set of lemmas permitted in proofs.

The proofs given to the user are chains of resolution-style inference rules. Much could be done to make them more concise and more readable. TRAMP [11] generates natural proofs, but integrating it with our system would be a major project. An alternative approach to proof reconstruction is to

use the ATP's output merely to extract the names of the lemmas used in the proof, and perhaps their instantiations; with these hints, a basic prover that works by Isabelle inference could find the proofs afresh. A prover that could be adapted for this purpose is Metis [10].

Despite its complications, we see some form of proof reconstruction as essential. Even if we trust the ATPs, we may not want to trust our translations between typed higher-order logic and untyped first-order logic. Moreover, having proof reconstruction in source form makes the proof script self-contained.

9.2. User interface

At present, any proofs found are displayed in one of Proof General's buffers, in a similar manner to the current goals and proof state. Preliminary experiments suggest that this is quite distracting. The user has no control over when these proofs are displayed and may be irritated by these sudden interruptions to her train of thought.

It is possible that a better model of interaction may be that of the Office Assistant in Microsoft Word. This occupies a separate window from the document the user is working on, occasionally alerting them to possible problems that have just arisen, or offering pertinent help. The proof assistant would, ideally, have different settings for helpfulness; the user could tailor the amount and types of assistance provided, thereby ensuring that it is helpful rather than distracting.

9.3. Prover performance

Isabelle's full lemma library corresponds to about 1400 first-order clauses. Automatic provers often fail to prove even trivial results in the presence of so many clauses. The *set of support* heuristic (SOS) [27] is a classic means of improving performance by ignoring irrelevant axioms: it requires all inferences to involve the negated conjecture, preventing aimless forward inferences involving the axioms alone. Unfortunately, modern theorem provers such as Vampire and SPASS use different heuristics, restricting the application of resolution according to an ordering. The combination of ordered resolution and SOS is incomplete, and in some cases the proof attempt fails quickly. More research is needed to let us combine the goal-orientation of SOS with the high performance of ordered resolution. The hierarchical structure of Isabelle theories may support algorithms for removing irrelevant axioms or generating effective orderings.

10. Conclusions

Our prototype is essentially complete and already demonstrates the key ideas. Automatic provers run in the background. If a proof is found, it is delivered to the user in source form. The user does not have to prepare the problem first, for example by identifying the necessary lemmas. Our prototype still requires much tuning before it will be genuinely useful. We need to boost the performance of the automatic theorem provers, for example by filtering out some irrelevant axioms.

Acknowledgments

We are grateful to the SPASS and Vampire teams for their co-operation and to Gernot Stenz and Geoff Sutcliffe for running some of our problems on their reasoning systems. The referees made numerous helpful suggestions. The research was funded by the EPSRC grant GR/S57198/01 *Automation for Interactive Proof*.

References

[1] Wolfgang Ahrendt, Bernhard Beckert, Reiner Hähnle, Wolfram Menzel, Wolfgang Reif, Gerhard Schellhorn, Peter H. Schmitt, Integrating automated and interactive theorem proving, in: Wolfgang Bibel, Peter H. Schmitt (Eds.), Automated Deduction—A Basis for Applications, volume II. Systems and Implementation Techniques, Kluwer Academic Publishers, Dordrecht, 1998, pp. 97–116.

[2] David Aspinall, Proof general: a generic tool for proof development, in: S. Graf, M. Schwartzbach (Eds.), TACAS '00: Tools and Algorithms for Construction and Analysis of Systems, 6th International Conference, LNCS, vol. 1785, Springer, Berlin, 2000, pp. 38–42, On the Internet at http://proofgeneral.inf.ed.ac.uk/.

[3] Matthias Baaz, Alexander Leitsch, On Skolemization and proof complexity, Fundamenta Informaticae 20 (4) (1994) 353–379.

[4] Marc Bezem, Dimitri Hendriks, Hans de Nivelle, Automatic proof construction in type theory using resolution, Journal of Automated Reasoning 29 (3–4) (2002) 253–275.

[5] Judy Crow, Sam Owre, John Rushby, Natarajan Shankar, Mandayam Srivas, A tutorial introduction to PVS, Technical report, Computer Science Laboratory, SRI International, 1995. First published in Workshop on Industrial-Strength Formal Specification Techniques, Boca Raton, Florida.

[6] in: Emden R. Gansner, John H. Reppy (Eds.), The Standard ML Basis Library, Cambridge University Press, Cambridge, MA, 2004.

[7] Michael J.C. Gordon, Thomas F. Melham, Introduction to HOL: A Theorem Proving Environment for Higher Order Logic, Cambridge University Press, Cambridge, MA, 1993.

[8] G.P. Huet, A unification algorithm for typed λ-calculus, Theoretical Computer Science 1 (1975) 27–57.

[9] Joe Hurd, Integrating Gandalf and HOL, in: Yves Bertot, Gilles Dowek, André Hirschowitz, Christine Paulin, Laurent Théry (Eds.), Theorem Proving in Higher Order Logics: TPHOLs '99, LNCS, vol. 1690, Springer, Berlin, 1999, pp. 311–321.

[10] Joe Hurd, First-order proof tactics in higher-order logic theorem provers, in: Myla Archer, Ben Di Vito, César Muñoz (Eds.), Design and Application of Strategies/Tactics in Higher Order Logics, number NASA/CP-2003-212448 in NASA Technical Reports, pages 56–68, September 2003.

[11] Andreas Meier, TRAMP: transformation of machine-found proofs into natural deduction proofs at the assertion level (system description), in: David McAllester (Ed.), Automated Deduction—CADE-17 International Conference, LNAI, vol. 1831, Springer, Berlin, 2000, pp. 460–464.

[12] Jia Meng, Integration of interactive and automatic provers, in: Manuel Carro, Jesus Correas (Eds.), Second CologNet Workshop on Implementation Technology for Computational Logic Systems, 2003. On the Internet at http://www.cl.cam.ac.uk/users/jm318/papers/integration.pdf.

[13] Jia Meng, Lawrence C. Paulson, Experiments on supporting interactive proof using resolution, in: David Basin, Michaël Rusinowitch (Eds.), Automated Reasoning—Second International Joint Conference, IJCAR 2004, LNAI, vol. 3097, Springer, Berlin, 2004, pp. 372–384.

[14] Tobias Nipkow, Lawrence C. Paulson, Markus Wenzel, in: Isabelle/HOL: A Proof Assistant for Higher-Order Logic, LNCS Tutorial, vol. 2283, Springer, Berlin, 2002.

[15] Andreas Nonnengart, Christoph Weidenbach, Computing small clause normal forms, in: Robinson, Voronkov [20], chapter 6, pp. 335–367.

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

[17] Lawrence C. Paulson, A generic tableau prover and its integration with Isabelle, Journal of Universal Computer Science 5 (3) (1999) 73–87.
[18] Lawrence C. Paulson, Isabelle's logics: FOL and ZF. Technical report, Computer Laboratory, University of Cambridge, 2003. On the Internet at http://isabelle.in.tum.de/dist/Isabelle2003/doc/logics-ZF.pdf.
[19] Alexander Riazanov, Andrei Voronkov, Vampire 1.1 (system description), in: Rajeev Goré, Alexander Leitsch, Tobias Nipkow (Eds.), Automated Reasoning—First International Joint Conference, IJCAR2001, LNAI, vol. 2083, Springer, Berlin, 2001, pp. 376–380.
[20] Alan Robinson, Andrei Voronkov (Eds.), Handbook of Automated Reasoning, Elsevier Science, Amsterdam, 2001.
[21] Jörg Siekmann, Christoph Benzmüller, Armin Fiedler, Andreas Meier, Immanuel Normann, Martin Pollet, Proof development with Ω mega: the irrationality of $\sqrt{2}$, in: Fairouz Kamareddine (Ed.), Thirty Five Years of Automating Mathematics, Kluwer Academic Publishers, Dordrecht, 2003, pp. 271–314.
[22] Geoff Sutcliffe, Christian Suttner, The TPTP problem library: CNF Release v1.2.1, Journal of Automated Reasoning 21 (2) (1998) 177–203.
[23] G. Sutcliffe, J. Zimmer, S. Schulz, TSTP data-exchange formats for automated theorem proving tools, in: W. Zhang, V. Sorge (Eds.), Distributed Constraint Problem Solving and Reasoning in Multi-Agent Systems, number 112 in Frontiers in Artificial Intelligence and Applications, IOS Press, 2004.
[24] Christoph Weidenbach, Combining superposition, sorts and splitting, in: Robinson, Voronkov [20], chapter 27, pp. 1965–2013.
[25] Markus Wenzel, Type classes and overloading in higher-order logic, in: Elsa L. Gunter, Amy Felty (Eds.), Theorem Proving in Higher Order Logics: TPHOLs '97, LNCS, vol. 1275, Springer, Berlin, 1997, pp. 307–322.
[26] Markus M. Wenzel, Isabelle/Isar—A Versatile Environment for Human-readable Formal Proof Documents, Ph.D. thesis, Technische Universität München, 2002.
[27] Lawrence Wos, George A. Robinson, Daniel F. Carson, Efficiency and completeness of the set of support strategy in theorem proving, Journal of the ACM 12 (4) (1965) 536–541.