Translating Higher-Order Problems to First-Order Clauses

Jia Meng[1], Lawrence C. Paulson[2]
[1]National ICT, Australia
jiameng@nicta.com.au
[2]Computer Laboratory, University of Cambridge, U.K.
LP15@cam.ac.uk

Abstract

Proofs involving large specifications are typically carried out through interactive provers that use higher-order logic. A promising approach to improve the automation of interactive provers is by integrating them with automatic provers, which are usually based on first-order logic. Consequently, it is necessary to translate higher-order logic formulae to first-order form. This translation should ideally be both sound and practical. We have implemented three higher-order to first-order translations, with particular emphasis on the translation of types. Omitting some type information improves the success rate, but can be unsound, so the interactive prover must verify the proofs. In this paper, we will describe our translations and experimental data that compares the three translations in respect of their success rates for various automatic provers.

1 Introduction

Interactive theorem provers, such as HOL4 [GM93], Isabelle [NPW02] and PVS [ORR+96] are widely used for formal verifications and specifications. They provide expressive formalisms and tools for managing large scale proof projects. However, a major weakness of interactive provers is the lack of automation. In order to overcome the problem, we have integrated Isabelle with automatic theorem provers (ATPs) [MQP06]. ATPs combine a variety of reasoning methods and do not require users' instructions on how to use an axiom or when to use an axiom. For example, they do not require equalities to be oriented, but use them as undirected equations.

Among the many logics used by interactive provers, higher-order logic (HOL) is the most popular one because of its expressiveness. One of the most widely used logics in Isabelle is Isabelle/HOL. In contrast, most of the powerful ATPs are based on first-order logic (FOL). Therefore, it is important to translate HOL problems into FOL format.

Those HOL problems that do not involve function variables, predicate variables or λ-abstractions can be translated directly to FOL format. However, we must be careful about how to translate HOL problems that are truly higher-order. In particular, we need to include the problem's type information to preserve soundness. A sound approach is to include all types for all terms. Unfortunately, this will result in large terms and clauses, and much of the type information is redundant. A more compact type representation could enhance the performance of ATPs. Omitting some type information could lead to unsoundness, which ultimately we will prevent through proof reconstruction (Sect. 2.5).

We have implemented three HOL to FOL translations, and we believe two of them are new. We have also carried out extensive experiments in order to assess their effectiveness with the provers E [Sch04], SPASS [Wei01] and Vampire [RV01].

Paper outline. We first describe three HOL to FOL translations and discuss their soundness (Sect. 2). We then describe the experiments we ran (Sect. 3) and finally offer some conclusions (Sect. 4).

2 Background

Higher-order logic (HOL) extends first-order logic (FOL) in several respects. The main difference is that HOL terms can denote truth values and functions. Function values can be expressed using λ-abstractions or by *currying*: that is, by applying a function to fewer than the maximum number of arguments. In FOL, a function must always be supplied the same number of arguments. In translating from HOL to FOL, the only way to reconcile this difference is to regard all HOL functions as constants while providing a two-argument function (called `app` and is abbreviated by `@` below) to express function application. In addition, we need a predicate `B` to convert all top-level FOL terms of boolean type to predicates. These translations allow first-order provers to solve many problems that contain higher-order features, though they do not yield the full power of higher-order logic.

For example, the HOL formula $\forall F\, p(F(x))$ is translated to

```
{++B(@(p,@(F,x)))}
```

We use the combinators **I**, **K**, **C**, **B** and **S** to represent λ-abstractions, asserting the combinator reduction equations as axioms. (Although **K** and **S** suffice in theory, the resulting translation is exponential in the number of abstractions.) Another axiom we assert is function extensionality:

$$\forall f g\, [(\forall x\, f(x) = g(x)) \rightarrow f = g].$$

It has the following clause form, where `e` is a reserved Skolem function symbol, yielding some x such that $f(x) \neq g(x)$.

```
{--equal(@(F,@(@(e,F),G)),@(G,@(@(e,F),G))),
 ++equal(F,G)}
```

Finally, equality in Isabelle may appear in a λ-abstraction, and thus is treated as an ordinary function symbol. We use a new function symbol `fequal` to represent the function version of equality. Through λ-reduction, this equality may be promoted to predicate level, becoming an ordinary equality. Promotion requires two additional axioms:

```
{++B(@(@(fequal,X),Y)),--equal(X,Y)}
{--B(@(@(fequal,X),Y)),++equal(X,Y)}
```

Not all subgoals require the full power of HOL. Often the initial steps of the proof replace complicated constructions by simple ones. Of the remaining subgoals, many are purely first-order. Others are higher-order but use no λ-abstractions. These special cases admit more efficient translations into first-order clauses, though naturally we must also provide a translation that accommodates the general case.

2.1 Types in Isabelle/HOL

We have just seen how to represent HOL formulae in FOL form. A more important issue is to embed type information of HOL formulae in FOL clauses. Let us review how this works for problems that are already first-order [MP04, MQP06]. Before that, we give a brief overview of Isabelle/HOL's polymorphically sorted type system. We refer readers to the two papers above for more information.

Isabelle/HOL supports *axiomatic type classes*, where a type class is a set of types. For example, the type for real numbers `real` is a member of type class `linorder`. A type class is axiomatic because it may have a set of properties—specified by axioms—that all its member types should satisfy. A type may belong to several type classes and an intersection of type classes is a *sort*. Moreover, each type constructor has one or more *arities*, which describe how the result type class depends upon the arguments' type classes. For example, type constructor `list` has an arity that says if its argument is a member of class `linorder` then the resulting list's type is also a member of `linorder`.

Constants can be overloaded and types can be *polymorphic*, allowing instantiation to more specific types. For example, the $\leq$ operator has the polymorphic type $\alpha \to \alpha \to bool$; when it has type $nat \to nat \to bool$ it denotes the usual ordering of the natural numbers, and when it has type $\alpha\ set \to \alpha\ set \to bool$ it denotes the subset relation. The latter type is still polymorphic in the type of the set's elements. Isabelle's overloading cannot be eliminated by preprocessing because polymorphic theorems about $\leq$ are applicable to all instances of this function, despite their different meanings.

When we translate Isabelle formulae to FOL clauses, we need to formalize types, especially in view of Isabelle's heavy use of overloading. We need to ensure that Isabelle theorems involving polymorphic functions are only used for appropriate types. To accomplish this, polymorphic functions carry type information as additional arguments and we translate Isabelle types to FOL terms. For example, we translate $x \leq y$ where x and y are $\alpha\ set$ to $le(x, y, set(\alpha))$. We also translate Isabelle's axiomatic type classes into first-order clauses. For this, we translate type classes to FOL predicates and types to FOL terms.

This translation is reasonably compact, and it enforces overloading ($\leq$ on sets is not confused with $\leq$ on integers), but it does not capture other aspects of types. For example, if we declare a two-element datatype `two`, then we obtain the theorem

$$\forall x\, [x = a \vee x = b].$$

The corresponding clause does not mention type `two`:

```
{++equal(X,a), ++equal(X,b)}
```

It therefore asserts that the universe consists of two elements; given our other axioms, ATPs easily detect the inconsistency. We simply live with this risk for the moment, pending the implementation of proof reconstruction. A simple way of detecting such issues is to check whether a proof refers to at least one conjecture clause: if not, then the axiom clauses by themselves are inconsistent. This method detects some invalid proofs, but not all of them.

Since HOL problems require currying, we need a different type embedding method from first-order ones. We have implemented three type translations, namely fully-typed, partial-typed and constant-typed.

2.2 The Fully-Typed Translation

The fully-typed translation, which resembles Hurd's translation [Hur02], is sound. The special function `typeinfo`, which we abbreviate to `T`, pairs each term with its type. For instance, the formula $P < Q$ is translated to

```
{++B(T(@(T(@(T(<, a=>a=>bool), T(P,a)), a=>bool), T(Q,a)), bool))}
```

This translation is sound because it includes types for all terms and subterms, right down to the variables. When two terms are unified during a resolution step, their types are unified as well. This instantiation of types guarantees that terms created in the course of a proof continue to carry correct types. Isabelle unifies polymorphic terms similarly. In fact, the resolution steps performed by an ATP could in principle be reconstructed in Isabelle. Each FOL axiom clause corresponds to an Isabelle theorem. If two FOL clauses are resolved, then the resolvant FOL clause will correspond to the Isabelle theorem produced by Isabelle's own resolution rule.

The fully-typed translation introduces much redundancy. Every part of a function application is typed: the function's type includes its argument and result types, which are repeated in the translation of the function's argument and by including the type of the returned result. Through experiments (Sect. 3), we have found that these large terms degrade the ATPs' performance. A more compact HOL translation should improve the success rate.

Hurd [Hur03] uses an untyped translation for the same reason. No term or predicate has any type information. Because this translation can produce unsound proofs, Hurd relies on proof reconstruction to verify them. If reconstruction fails, Hurd calls the ATP again, using a typed translation. Hurd says that this happens less than one percent of the time. This combination of an efficient but unsound translation with a soundness check achieves both efficiency and soundness. We intend to take the same approach.

If we are to achieve a compact HOL translation, we will have to omit some types, potentially admitting some unsound proofs. We cannot use a completely untyped translation because our requirements differ from Hurd's. His tactic sends to ATPs a few theorems that are chosen by users. In contrast, we send ATPs hundreds of theorems, many involving overloading. Omitting the types from this large collection would result in many absurd proofs, where for example, the operator $\leq$ simultaneously denoted "less than" on integers and the subset relation. We have designed and experimented with two compact HOL translations: the partial-typed and constant-typed translations. These translations attach the most important type information (such as type instantiations of polymorphic constants) that can block some incorrect resolutions.

2.3 The Partial-Typed Translation

The partial-typed translation only includes the types of functions in function calls. The type is translated to a FOL term and is inserted as a third argument of the application operator (`@`). Taking the previous formula $P < Q$ as an example, we translate it to

```
{++B(@(@(<, P, a=>a=>bool), Q, a=>bool))}.
```

Here, the type of `<` is `a=>a=>bool`, and we include this type as an additional argument of function application `@`.

In a HOL formula, a function may be passed to another function as an argument. If a function appears without arguments, we do not include its type. The FOL clauses are derived from Isabelle formulae, which we know to be well-formed and type correct. The partial-typed translation avoids the redundancy of the fully-typed translation. Most of the time, this type encoding also ensures correct treatments of Isabelle overloading: Isabelle overloaded constants are most likely to appear as operators (functions and predicates) in formulae, whose types are inserted by the partial-typed encoding.

However, the partial-typed translation can still yield unsound proofs. It is vulnerable to the example involving datatype `two`, described in Sect. 2.1 above.

2.4 The Constant-Typed Translation

In the constant-typed translation, we include types of polymorphic constants only. Furthermore, we do not include a constant's full type but only the instantiated values of its type variables. Monomorphic constants do not need to carry types because their names alone determine the types of their arguments. A polymorphic constant is translated to a first-order function symbol. Its arguments, which represent types, are obtained by matching its actual type against its declared type. This treatment of types is similar to the one we use for problems that are already first-order.

Again considering our standard example, if P and Q are natural numbers (type `nat`), we translate the formula $P < Q$ to

```
{++B(@(@(<(nat), P), Q))}.
```

Similarly, if P and Q are sets (type α `set`), it becomes

```
{++B(@(@(<(set(a)), P), Q))}.
```

As for equality, if it appears as a predicate, then we do not insert its type. However, if it appears as a constant in a combinator term, then we include its argument's type as its argument. Similarly, we translated the equality axiom above to the two clauses

```
{++B(@(@(fequal(T),X),Y)),--equal(X,Y)}
{--B(@(@(fequal(T),X),Y)),++equal(X,Y)}
```

This translation can reduce the size of terms significantly. However, like the partial-typed one, it can be unsound.

2.5 Which Translation to Use and Soundness Issues

Of the three HOL to FOL translations above, the fully-typed one is sound but produces excessively large terms. The partial-typed and constant-typed translations are more compact, but may introduce unsound proofs. If we use either of the compact translations, then we must verify proofs in Isabelle to ensure soundness.

There are several factors that affect the decision about which translation should be used as the default.

- Can we verify the proofs for partial-typed and constant-typed translations? The answer is yes. Although the FOL clauses carry insufficient type information, the clauses still correspond to Isabelle lemmas and goals. Our approach to proof reconstruction, which is currently being implemented, involves following the low-level resolution steps. If two clauses cannot be resolved due to incompatible types, Isabelle will detect this.
- What benefit do we obtain from using the compact translations? Our experimental results (Sect. 3) show that the compact translations can boost the success rate significantly. Therefore, it is worthwhile to use a compact translation, even if occasional unsound proofs require retrying the problem using the fully-typed translation.

Moreover, we aim to reconstruct proofs in Isabelle even if we use fully-typed translation. This is so that proofs can go through Isabelle kernel. Since proof reconstruction is needed regardless of which translation is used, the only potential extra cost involved in using a compact translation is that of occasional retries.

3 Experiments

It is obvious that the constant-typed translation is the most compact, while the fully-typed one is the least compact. We can therefore predict that the constant-typed translation will deliver the best results with ATPs, while the fully-typed one will turn out to be the worst. However, such claims need to be backed up by observations, especially given that the fully-typed translation is the best for soundness.

For our experiments, we took 79 problems generated by Isabelle, most of which are higher-order. Since our HOL translation can also be used for purely FOL problems and our experiments were aimed at testing efficiency of the translation methods, we translated all problems (both HOL and FOL) using the three translation methods we mentioned in the previous section. We used our relevance filter [MP06] to reduce the sizes of the problem. We ran these tests on a bank of Dual AMD Opteron processors running at 2400MHz, using Condor[1] to manage our batch jobs.

Each graph compares the success rates of the three translations, for some prover, as the runtime per problem increases from 10 to 300 seconds. These short runtimes are appropriate for our application of ATPs to support interactive proofs, We tested three provers: E (Fig. 1), SPASS 2.2 (Fig. 2) and Vampire 8 (Fig. 3). We used E version

[1]`http://www.cs.wisc.edu/condor/`

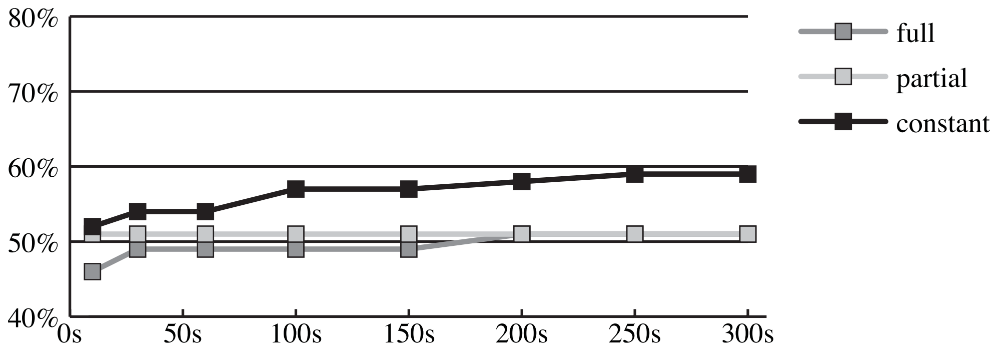

Figure 1: E, Version 0.91dev001

0.91dev001, a development version that surpasses E 0.9 “Soom”. SPASS ran in automatic mode and, in a second run, with SOS enabled and splitting disabled.[2] Vampire ran in its default mode and with its CASC option.

On the whole, the constant-typed translation did indeed yield the highest success rate, while the fully-typed translation yielded the lowest, especially when runtime is increased. This is as we would expect, but a glance at the graphs shows that the situation is more complex than this. Against expectations, the partial-typed translation frequently outperforms the constant-typed one. For runtimes below about 200 seconds, the partial-typed translation gives the best results with SPASS (default settings) and Vampire. As runtime increases to 300 seconds, the constant-typed translation comes out top (or nearly) in all five graphs.

At its default settings, SPASS does not perform well with any translation. It is safe to conjecture that only trivial problems are being proved, where the translation makes little difference. By enabling SOS, which makes the proof search more goal-directed, SPASS delivers excellent results with the constant-typed translation.

Readers may ask whether the fully-typed translation has a lower success rate because the other translations are finding unsound proofs for our problems. We have found that three of the 79 problems have unsound proofs. For one of the problems, incorrect axioms cause all three of its translations to be unsound, so this error does not bias the results. For the other two problems, the compact translations are indeed to blame, giving a bias of 2/79 or 2.5% against the fully-typed translation. The advantage given by the compact translations is much greater than this. We do regard this unsoundness rate as too high, and we are considering a number of options for reducing it.

To obtain a quantitative picture of the differences between the three translations, we chose one of the problems and used tptp2X [SS04] to summarize its syntactic features. This problem is of median size in our problem set. It has 1150 clauses after relevance filtering, of which 105 are non-trivial: the remainder constitute a monadic Horn theory that describes Isabelle’s type class system. Table 1 shows the figures common to all three translations. Table 2 shows variations among the translations.

A major difference is the maximal term depth, where fully-typed appears to have the greatest maximal term depth while constant-typed has the least. Shallower terms

[2]The precise option string is `-Splits=0 -FullRed=0 -SOS=1`.

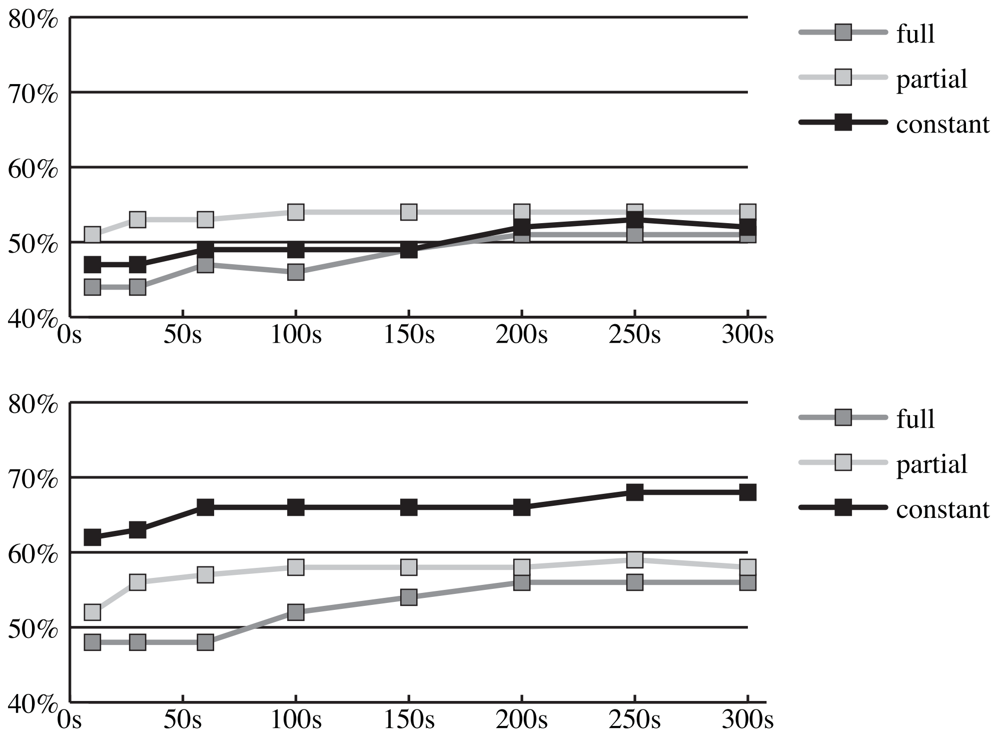

Figure 2: SPASS, Default Settings and with SOS

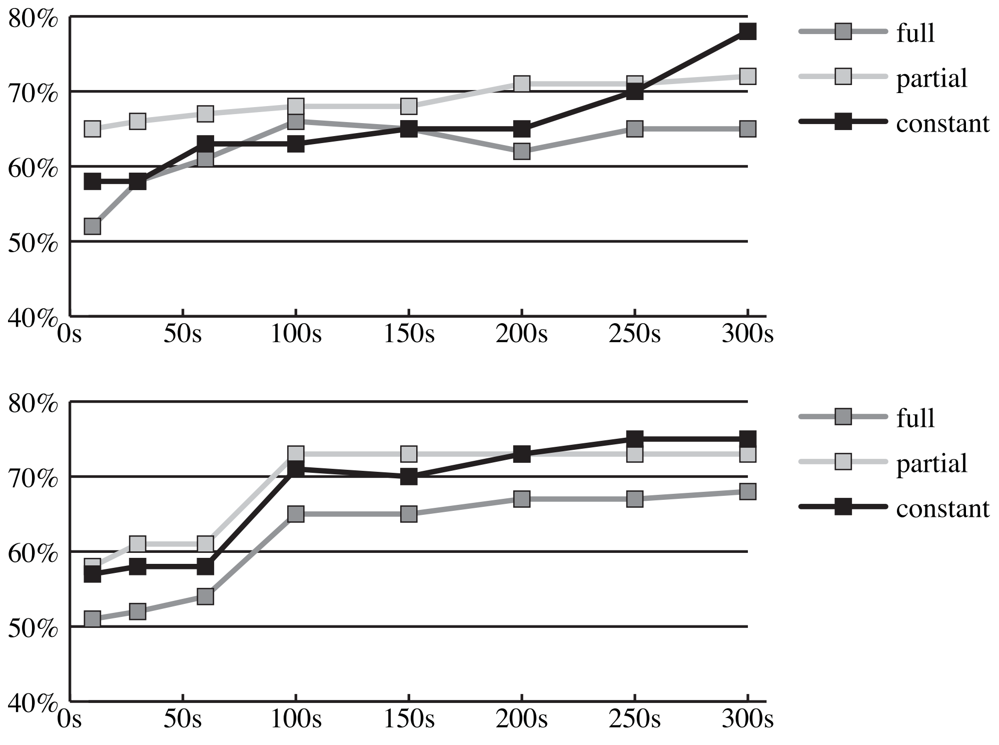

Figure 3: Vampire, Default Settings and in CASC Mode

Number of clauses	1150 (6 non-Horn; 198 unit)
Number of literals	2105 (152 equality)
Maximal clause size	3 (1 average)
Number of predicates	74 (0 propositional; 1–2 arity)

Table 1: Common to All Translations

	No. of functors	*No. of variables*	*Max. term depth*	*File size*
full	51 (42 constant)	1292 (69 singleton)	19 (1 average)	223460
partial	50 (42 constant)	1265 (66 singleton)	13 (1 average)	196663
constant	49 (10 constant)	1292 (106 singleton)	8 (1 average)	171724

Table 2: Differences between Three Translations

may be easier to process by ATPs, which may be a reason why constant-typed performs better overall. In addition, constant-typed produces the smallest problem file. Finally, although the fully-typed and constant-typed translations have the same number of variables, the latter has more singleton variables, which are variables that occur in a clause only once.

These statistics again suggest that the constant-typed translation should give the best results, so we have no explanation for the many situations in which the partial-typed translation performs best. The dips in the graphs, where the success rate drops as the runtime goes up, are also puzzling. A notable one is with Vampire, default settings, with the fully-typed translation (Fig. 3). Vampire's *limited resource strategy*, which discards clauses that cannot be used within the time limit, may explain its dips. For E and SPASS we have no explanation, but we have no doubt that the dips are real. Similar dips appear in our other experiments [MP06], and all of these measurements are made by automatic procedures.

4 Conclusions

We have described three HOL to FOL translations, which differ in their treatment of types. We have carried out extensive experiments to evaluate the effectiveness of the three translations. We have also obtained some statistics concerning how compact our translations are. Of the three translations—fully-typed, partial-typed and constant-typed—the constant-typed translation produces the most compact output.

Naturally, we would expect a provers' success rate to increase with a more compact clause form. This is what we have seen with E and with SPASS (provided SOS is enabled). However, we are surprised to see the simplest and most compact format does not always yield the best results with Vampire. Given that Vampire gives the best overall results, the partial-typed translation is worth considering. Because the more compact translations can be unsound, proofs found using them must be validated in some way, such as by proof reconstruction.

The higher-order logic we have investigated is Isabelle/HOL. However, our translations should be equally applicable to the similar logic implemented in the HOL4 system.

Any translations for PVS would have to take account of predicate subtyping, but their treatment of basic types might be based on our techniques.

Acknowledgements

The research was funded by the EPSRC grant GR/S57198/01 *Automation for Interactive Proof* and by the L4.verified project of National ICT Australia. Joe Hurd has given much helpful advice on how to translate from HOL to FOL. The referees made many useful suggestions for improving this paper.

References

[BR04] David Basin and Michaël Rusinowitch, editors. *Automated Reasoning — Second International Joint Conference, IJCAR 2004*, LNAI 3097. Springer, 2004.

[GM93] Michael J. C. Gordon and Thomas F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge Univ. Press, 1993.

[Hur02] Joe Hurd. An LCF-style interface between HOL and first-order logic. In Andrei Voronkov, editor, *Automated Deduction — CADE-18 International Conference*, LNAI 2392, pages 134–138. Springer, 2002.

[Hur03] Joe Hurd. First-order proof tactics in higher-order logic theorem provers. In Myla Archer, Ben Di Vito, and César Muñoz, editors, *Design and Application of Strategies/Tactics in Higher Order Logics*, number NASA/CP-2003-212448 in NASA Technical Reports, pages 56–68, September 2003.

[MP04] Jia Meng and Lawrence C. Paulson. Experiments on supporting interactive proof using resolution. In Basin and Rusinowitch [BR04], pages 372–384.

[MP06] Jia Meng and Lawrence C. Paulson. Lightweight relevance filtering for machine-generated resolution problems. These proceedings, 2006.

[MQP06] Jia Meng, Claire Quigley, and Lawrence C. Paulson. Automation for interactive proof: First prototype. *Information and Computation*, 2006. in press.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer, 2002. LNCS Tutorial 2283.

[ORR$^+$96] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M.K. Srivas. PVS: Combining specification, proof checking, and model checking. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer Aided Verification: 8th International Conference, CAV '96*, LNCS 1102, pages 411–414. Springer, 1996.

[RV01] Alexander Riazanov and Andrei Voronkov. Vampire 1.1 (system description). In Rajeev Goré, Alexander Leitsch, and Tobias Nipkow, editors, *Automated Reasoning — First International Joint Conference, IJCAR 2001*, LNAI 2083, pages 376–380. Springer, 2001.

[Sch04] Stephan Schulz. System description: E 0.81. In Basin and Rusinowitch [BR04], pages 223–228.

[SS04] Geoff Sutcliffe and Christian Suttner. The TPTP problem library for automated theorem proving. On the Internet at `http://www.cs.miami.edu/~tptp/`, 2004.

[Wei01] Christoph Weidenbach. Combining superposition, sorts and splitting. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume II, chapter 27, pages 1965–2013. Elsevier Science, 2001.