Translating Higher-Order Clauses to First-Order Clauses

Jia Meng (jia.meng@nicta.com.au)
National ICT, Australia

Lawrence C. Paulson (lp15@cam.ac.uk)
Computer Laboratory, University of Cambridge, U.K.

Abstract. Interactive provers typically use higher-order logic, while automatic provers typically use first-order logic. In order to integrate interactive provers with automatic ones, it is necessary to translate higher-order formulae to first-order form. The translation should ideally be both sound and practical. We have investigated several methods of translating function applications, types and λ-abstractions. Omitting some type information improves the success rate, but can be unsound, so the interactive prover must verify the proofs. This paper presents experimental data that compares the translations in respect of their success rates for three automatic provers.

1. Introduction

Interactive theorem provers, such as HOL4 [5], Isabelle [14] and PVS [15] are widely used for formal specification and verification. They provide expressive formalisms and tools for managing large scale proof projects. However, a weakness of interactive provers is their lack of automation. In order to overcome this weakness, we have integrated Isabelle with automatic theorem provers (ATPs) [12]. ATPs utilize a variety of reasoning methods and do not require hints on how or when to use an axiom. For example, they do not expect users to orient equalities or instantiate quantifiers.

Many interactive provers implement some form of higher-order logic (HOL). Isabelle supports a variety of logics, but most users know it only as Isabelle/HOL. In contrast, the most powerful ATPs are all based on first-order logic (FOL). Therefore, a successful integration requires translating HOL problems into first-order form.

Our criteria for these translations are pragmatic and relate to the requirements of our integration. We do not expect to obtain automation of full higher-order logic. We do not expect the translations to have interesting theoretical properties, such as completeness. We merely seek a translation that achieves a high success rate for problems containing higher-order features. As will be seen, we even consider unsound translations, for our integration can verify the soundness of proofs when importing them into Isabelle (Sect. 2.8).

We build upon the work of Hurd. He has integrated Metis, his own first-order prover, with the HOL4 interactive proof environment [8]. We consider alternatives to Hurd's treatments of function applications, types and λ-abstractions, backing up our choices with extensive experimentation. A translation should preserve type information; a sound approach is to include types for all terms. Unfortunately, full type information takes up much space. A more compact representation yields better results, as we demonstrate below. Omitting some type information can lead to unsound proofs. We outline an algorithm to translate proofs from an unsound translation into a sound one, which can be used to test the soundness of the original proofs.

A contrasting approach is Otter-λ: Beeson [2] has modified the source code of the Otter theorem prover, in particular its unification algorithm, to provide limited higher-order features. Beeson notes, however, that Otter-λ does not implement higher-order logic:

> We do not regard Otter-λ as a "combination of first-order logic and higher-order logic." Lambda logic is not higher-order, it is untyped. ... While there probably are interesting connections to typed logics, some of the questions about those relationships are open at present ... [2, p. 313]

Even with an ATP for higher-order logic [3], we would somehow need to formalize Isabelle/HOL's unusual type system (Sect. 2.2). Our translations permit the use of high-performance ATPs, unmodified. ATP technology is developing rapidly, and we do not wish to be tied to a single system such as Otter.

Bouillaguet et al. [4] have developed a translation from higher-order logic to first-order logic. They eliminate type information and have proved this to be sound and complete. They even use Isabelle/HOL. Their work is impressive in its specific application of data structure verification. However, they do not translate λ-abstractions and their formulas may only refer to the types of integers and Java objects.

We have implemented three HOL to FOL translations, two of which are new. We have also developed an optimization technique, which essentially produces two additional translations. We have also addressed the question of how to remove λ-abstractions from the HOL problems. We have implemented both λ-lifting, which replaces λ-abstractions by newly-defined functions [6], and two different combinator translations [24]. We have carried out extensive experiments on all of these translations, using the provers E [20], SPASS [26] and Vampire 8.1 [19]. One compact translation significantly outperforms the basic, fully-typed one.

Compared with our previous work [10], we test more translations, we use a larger problem set, and we take strong measures to ensure that only sound proofs are counted.

Paper outline. We first describe three approaches to translating types and discuss their soundness (Sect. 2), then describe our experimental results (Sect. 3). Next, we present three approaches to translating λ-abstractions, along with experimental results (Sect. 4). Finally, we offer some conclusions (Sect. 5). Appendix A presents examples of the translations.

2. The Three Translations

Any problem that involves function variables, Boolean variables or λ-abstractions is clearly higher-order, but the precise criteria are surprisingly subtle. Consider that an equation such as `hd(Cons X L) = X` is first-order if `X` is of polymorphic type, but higher-order if its type is Boolean. This situation seems odd: the polymorphic equation encompasses all types, including type `bool`. However, only in the Boolean case can `hd(Cons X L)` be seen as a formula as opposed to a term. In higher-order logic, formulas are simply terms of Boolean type.

True higher-order reasoning requires deductive mechanisms, such as higher-order unification, to generate suitable λ-abstractions and logical formulas as the proof develops. We merely aim to allow reasoning about the first-order aspects of higher-order problems. Our methods allow use of the equation `hd(Cons X L) = X` even for Boolean lists, but we do not expect to prove `hd(Cons X L)` except in trivial cases. Our methods accept problems containing functions expressed in λ-notation, and can simplify the results of applying such functions to arguments, but we do not expect proofs to find interesting instantiations of function variables.

If a higher-order formula is essentially first-order, then translating it to first-order logic is straightforward; otherwise, its higher-order features must be removed as described below. We use the following criteria for a formula to be essentially first-order:

- No function has an argument involving function or Boolean types.
- There are no variables of function or Boolean types.
- There are no higher-type instances of overloaded constants.

An example of the last criterion is the constant 1, which Isabelle/HOL allows to have any type. (Users can overload 1 with multiple definitions, even at function types.) If it appears with a function type, then the problem counts as higher-order, because potentially 1 can appear both as a constant and as a function symbol.

2.1. Features Common to All Translations

Our translations act on the output of the clause form transformation, which takes place inside Isabelle. Thus, their input consists of clauses that contain higher-order features. The translations are designed to preserve the first-order aspects of the problem while making them acceptable to a first-order prover. Higher-order logic extends first-order logic in several respects: chiefly, 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. Not so in HOL, as we can see by considering the function `map`, which applies a function to every element of a list. It can appear with no arguments, with one argument, or with two arguments:

```
map = map
map id = id
map F [] = []
```

In translating from HOL to FOL, the natural treatment of currying is to regard all HOL functions as constants while providing a two-argument function (called `@` below) to express function application.

HOL formulas are simply Boolean terms, but in first-order logic, formulas and terms are distinct. A HOL term such as `X < g(Y)` can appear as an assertion, but it can also be a function argument, as in `f(X < g(Y))`. Our translations address this distinction by providing a predicate, called `B` below, to convert a Boolean term to a formula. Logically, `B(x)` means `x=True`. Ignoring types, we translate the assertion `X < g(Y)` to the formula `B(@(@(less,X),@(g,Y)))` and the term `f(X < g(Y))` to the term `@(f,@(@(less,X),@(g,Y)))`.

Equality requires special treatment: a HOL equality assertion must be translated to use the ATP's built-in equality primitive. However, equality in HOL may appear in a Boolean-valued term, for example `f(X=0)`. We translate such occurrences to a new constant symbol, `fequal`, yielding for our example `@(f,@(@(fequal,X),0))`. Reasoning steps may promote this constant to the predicate level, where it expresses an ordinary equality. To handle such situations, we define `fequal` in Isabelle/HOL using the axiom

$$\forall X\, Y\, [\texttt{fequal}\, X\, Y \iff X = Y].$$

Ignoring types again, this corresponds to the following two clauses:

```
-B(@(@(fequal,X),Y)) | X = Y
B(@(@(fequal,X),X))
```

Following Hurd [8] (based of course on Turner [25]), we remove λ-abstractions by translating them to Curry's combinators **S**, **K**, **I**, **B** and **C**. These are easily defined in higher-order logic by the usual combinator reduction equations. This and alternative approaches are examined in Sect. 4.

A basic axiom of HOL is function extensionality:

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

It has the following clause form, where `ext` is a reserved Skolem function symbol. Given f and g, it yields some x such that $f(x) \neq g(x)$.

```
@(F,@(@(ext,F),G)) != @(G,@(@(ext,F),G)) | F = G
```

There is no need for `ext` to be available as a constant.[1] Converting it to a function would abbreviate this axiom as follows:

```
@(F,ext(F,G)) != @(G,ext(F,G)) | F = G
```

Generalizing this idea, we might eliminate unnecessary uses of the constants `@` and `B` by preprocessing the set of clauses. We examine this optimization in Sect. 2.7 below.

All of the examples shown above ignore types. The translations described below differ in their treatment of types. They translate terms involving the constants `fequal` and `ext` in the same manner as they translate other constants.

2.2. Types in Isabelle/HOL

Isabelle supports polymorphism. This expresses universal quantification over types, though with no explicit type quantifier. For example, a polymorphic identity function might have type `'a=>'a`, allowing it to take on any type of the form T`=>`T by instantiating `'a`. For purposes of deduction, type variables come in two forms. *Schematic* type variables, written `?'a`, `?'b`, …, can be instantiated by types. *Free* type variables, written `'a`, `'b`, …, are not really variables but represent fixed, unknown, types; they typically occur in conjecture clauses, and are essentially Skolem constants for polymorphism.

An example may be illustrative. Consider proving that `map id`, the function that applies the identity function to every element of a list, equals the identity function for lists:

[1] Miller [13] notes that providing Skolem functions as constants yields the effect of the axiom of choice. Isabelle/HOL includes this axiom anyway, so we see this as no danger.

```
map id = id
```

When this equation is stated as a conjecture, the functions have the following types:[2]

- `map`: `('a=>'a) => ('a list => 'a list)`
- `id` (first occurrence): `'a => 'a`
- `id` (second occurrence): `'a list => 'a list`

Proving this equation for the arbitrary, fixed type `'a` establishes it for all types. Isabelle replaces `'a` by `?'a` once the theorem has been proved, to express this generality.

The Isabelle/HOL type system has further refinements. An *axiomatic type class* denotes a set of types. For example, `real` (the type of real numbers) 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, the 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`. This claim is justified by Isabelle declarations that define a lexicographic ordering for lists and prove it to satisfy the axioms of a linear order.

A constant can be overloaded by giving it a polymorphic type, with different definitions for various types. For example, the $\leq$ operator has the polymorphic type `'a=>'a=>bool`; when it has type `nat=>nat=>bool` it denotes the usual ordering of the natural numbers, and when it has type `'a set =>'a set => 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.

2.3. Translating Types and Terms to First-Order Logic

Type information—at least some of it—must be preserved when translating higher-order formulae to first-order logic. This in turn requires that we translate Isabelle types to FOL terms. A translation should

[2] Type constructors in Isabelle use a postfix syntax, so we write `'a list` rather than `list('a)`.

respect overloading, ensuring that Isabelle theorems involving polymorphic functions are only used for appropriate types. Isabelle's axiomatic type class information can be formalized as a collection of simple facts and implications, and easily translated to Horn clauses [12].

Isabelle types are translated to first-order logic as follows:

- Schematic type variables are translated to first-order variables; for example, `?'a` is translated to `T_a`. (An initial capital signifies a variable in most ATPs.)
- Free type variables are translated to first-order constants; for example, `'a` is translated to `t_a`. (Lower case signifies a constant.)
- Compound types are formed by the application of a type constructor to arguments. The type constructor is translated to a first-order function, prefixed with `tc_`, and its arguments are translated recursively. For example, the function type `?'a=>nat` is translated to `tc_fun(T_a,tc_nat)`. This example contains two type constructors, `fun` and `nat`, the latter taking zero arguments. The translation of `'a set set` is `tc_set(tc_set(t_a))`.

Our three translations differ in how much type information they retain. As a starting point, consider a translation that retains no type information at all. The input is a HOL term from which all λ-abstractions have been removed, as discussed in Sect. 4. There are four remaining kinds of terms:

- Schematic variables express generality and are translated to first-order variables.
- Free variables are essentially Skolem constants.
- Constants, even those of function type, are translated to first-order constants.
- Function applications are translated using the `@` operator, as shown in Sect. 2.1 above.

This basic translation resembles the one for types mentioned above. Variable and constant names are given prefixes to ensure correct capitalization and to distinguish entities of different kinds that have the same name; we omit the details.

Omitting types yields a compact result, but the resulting proofs can be *unsound*: that is, meaningless in higher-order logic. In an unsound proof, the empty clause is reached through violations of the (omitted) type constraints rather than by refuting the conjecture, which typically plays no role in the proof. For example, in Isabelle we can declare a two-element enumeration type, `two`.

```
datatype two = a | b
```

The Isabelle theory will then include the theorem

```
∀x::two. x=a ∨ x=b
```

An untyped translation from Isabelle/HOL to first-order logic will discard the constraint to type `two`:

```
X=a | X=b
```

It therefore asserts that the universe consists of two elements; given a few axioms about the natural numbers or lists, ATPs easily detect the inconsistency.

Combinator axioms can also give rise to unsound proofs. If their types are omitted, then they can express fixedpoint operators, thus deriving a formula ϕ such that $\phi = \neg\phi$. This effect can occur not only with the traditional combinators **S**, **K** and **I**, but with any higher-order functions that may be present.

Unsound proofs fail during proof reconstruction, because Isabelle's inference system enforces type constraints. Therefore, they cannot cause Isabelle to accept false theorems. However, they can prevent the discovery of sound proofs. Including type information in the translation can prevent unsound proofs, but it can make problems too large, again preventing the discovery of sound proofs. We have a tradeoff between sound and prolix translations and unsound but compact ones. Below we discuss three possible treatments of types, the fully-typed, partially-typed and constant-typed translations. Finally, we say more about soundness and proof reconstruction.

2.4. The Fully-Typed Translation

The fully-typed translation, due to Hurd [7], is sound. A special function `ti` pairs each term with its type. For instance, the term $X \leq Y$ is translated to

```
ti(@(ti(@(ti(le, T_a => T_a => bool),
          ti(X, T_a)),
        T_a => bool),
     ti(Y, T_a)),
   bool).
```

For clarity of presentation, we omit the predicate `B`, needed if $X \leq Y$ occurs as a literal. We also leave the type constructors `=>` and `bool` in their Isabelle form rather than translating them as `tc_fun (T_a, tc_bool)`, etc. Although the equality predicate built into ATPs is untyped, this translation preserves the types of the two operands: the equality literal $X = Y$ is translated to

```
ti(X,T_a) = ti(Y,T_a).
```

In detail, the translation works as follows:

- A schematic variable is translated to `ti(`V`,`T`)`, where V is a logical variable and T is a translation of its type.

- A free variable or constant is translated to `ti(`c`,`T`)`, where c is a constant and T is a translation of its type.

- A function application is translated to `ti(@(`t`,`u`),`T`)`, where t and u are translations of the two subterms and T is the translation of the resulting type.

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. For example, unifying `ti(X, T_a)` with `ti(0, nat)` instantiates `T_a` to `nat`, preventing the unification of `ti(Y, nat)` with `ti(b, two)`. 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.

As the example illustrates, 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. These large terms are likely to yield a poor success rate, compared with more compact translations.

The size of the type information grows quadratically. To see this, consider how many times the type τ occurs in the application of a function f that takes n arguments:

arity	term	occs. of τ
0	$f : \tau$	1
1	$(f : \tau \to \tau)(x : \tau) : \tau$	4
2	$(((f : \tau \to \tau \to \tau)(x : \tau)) : \tau \to \tau)(x : \tau) : \tau$	8

As we increase the arity of f from n to $n+1$, we replace $f : \tau^n \to \tau$ by

$$(f : \tau^{n+1} \to \tau)(x : \tau) : \tau^n \to \tau.$$

Therefore, the number $h(n)$ of occurrences of τ satisfies the recurrence $h(0) = 1$ and $h(n+1) = h(n) + n + 3$. A simple induction allows us to prove that $h(n) = (n^2 + 5n + 2)/2$.

To achieve a compact HOL translation, we must omit some type information, potentially admitting unsound proofs. Hurd [8] uses an untyped translation, relying on proof reconstruction to verify the proofs and reject unsound ones. If reconstruction fails, Hurd calls the ATP again, using the fully-typed translation. Combining an efficient but unsound translation with a soundness check achieves both efficiency and soundness.

We cannot use an untyped translation because our requirements differ from Hurd's. His tactic gives the ATP a list of theorems chosen by the user. In contrast, we send ATPs hundreds of theorems, many involving overloading. Omitting all types from this large collection would result in many absurd proofs, where for example, the operator $\leq$ simultaneously denoted an integer ordering and the subset relation. We have designed and experimented with two compact HOL translations: the partially-typed and constant-typed translations. These attach the most important type information (such as type instantiations of polymorphic constants) that can block some incorrect resolutions. Neither attaches type information to variables, so neither correctly handles type `two`, described in Sect. 2.2 above.

2.5. The Partially-Typed Translation

The partially-typed translation is intended to preserve enough type information to prevent most unsound proofs, while avoiding the extreme redundancy of the fully-typed translation. It 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 $X \leq Y$ as an example, we translate it to

```
@(@(le, X, T_a => T_a => bool), Y, T_a => bool)
```

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

In detail, the translation works as follows:

- A schematic variable is translated to a logical variable.
- A free variable or constant is translated to a constant.
- A function application is translated to `@(`t`,`u`,`T`)`, where t and u are translations of the two subterms and T is the translation of the function's type.

This translation includes the type of every term, whether a constant or not, that is used as a function. It still contains some redundant type information, as the example shows.

2.6. The Constant-Typed Translation

The constant-typed translation is designed to be as compact as possible. It retains the minimum type information needed to ensure correct overloading of constants. Each polymorphic constant carries type information. 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. For example, the $\leq$ operator is declared to have type `'a=>'a=>bool`; if it appears with the type `nat=>nat=>bool`, then the type argument used in its translation is `nat`. This treatment of types is similar to the one we use for problems that are already first-order.

Again considering our standard example, if X and Y are natural numbers (type `nat`), we translate the formula $X \leq Y$ to

```
@(@(le(nat), X), Y)
```

If X and Y are sets (type α `set`), we translate the formula to

```
@(@(le(set(T_a)), X), Y)
```

Equality literals use the built-in equality symbol and contain no type information. If equality appears as a constant in a Boolean-valued term, then we use the equality function `fequal` mentioned in Sect. 2.1 above. This constant is treated like any other, and is translated to `fequal(T)`, where `T` expresses the type of its operands.

In detail, the translation works as follows:

– A schematic variable is translated to a logical variable.

– A free variable is translated to a constant.

– A constant is translated to a function applied to translated types, as described above.

– A function application is translated to `@(`t`,`u`)`, where t and u are translations of the two subterms.

This translation can reduce the size of terms significantly. However, it includes little type information and can be expected to admit many unsound proofs.

2.7. First-Order Versions of the Translations

As remarked above, our translations do not aim to achieve higher-order reasoning, but merely to put higher-order problems into a form acceptable to first-order provers. For example, the function `map` can appear sometimes with one argument and sometimes with two, due to currying. We obtain a single arity for `map` by introducing a function application operator, `@`.

The application operator naturally captures the syntax of higher-order logic, but it produces large terms. Are there more compact alternatives? The simplest way of avoiding arity conflicts is to regard function occurrences with different arities as denoting different functions: say, `map1` and `map2`. However, such an approach could preclude many proofs, given the importance of currying. A lemma about `map1` could not be used in a theorem about `map2`.

We have therefore investigated a hybrid approach that attempts to minimize uses of `@` while not eliminating them. We examine the set of clauses to find the minimum arity of each function. If some function f always appears with at least n arguments, then we use `@` only for arguments in excess of this minimum. For example, if `map` always appears with at least one argument, then (ignoring types) we translate `map F L` as `@(map(F),L)`.

This approach also precludes some proofs, namely those that disassemble function applications. For example, suppose we have the following axiom:

$$\forall F\,G\,X\,Y\,[\texttt{dominates}\,F\,G \wedge Y < GX \longrightarrow Y < FX].$$

This axiom expects terms of the form `@(f,x)` and `@(g,x)`; it will not be able to take part in proofs where the application operator has been suppressed. However, note that if `dominates f g` appears in the problem, then both functions will have minimum arities of zero, forcing the use of @ for all of their arguments. We do not expect many natural problems to be affected by this optimization; recall that our concern is the overall success rate rather than any notion of completeness.

This optimization is easily applied to our translations. We simply modify them to pass the first n arguments of function f directly to that function, provided all occurrences of f in the problem have at least n arguments. A related optimization is to eliminate the predicate B for boolean-valued functions that are used exclusively as predicates and never as arguments of functions.

For the fully-typed translation, suppose that the operator $\leq$ always appears with two arguments. Then, the term $X \leq Y$ is translated to

```
ti(le(ti(X,T_a), ti(Y,T_a)), bool).
```

If it occurs as a literal, and if the operator $\leq$ only appears in literals, then we omit the predicate B and the constraint to type bool:

```
le(ti(X,T_a), ti(Y,T_a)).
```

This translation is still sound: as before, all terms and subterms carry types, right down to the variables. The difference from the unoptimized version is that there are fewer subterms.

For the partially-typed translation, omitting @ could mean omitting all type information, so we do not consider this option.

For the constant-typed translation, if X and Y have type nat then $X \leq Y$ becomes simply le(X,Y,nat). This is very close to the translation we already use for first-order problems.

2.8. Soundness Issues

We can contemplate the use of unsound translations because all proofs are reconstructed in Isabelle. We use Hurd's Metis prover, which generates proof objects specifically to assist reconstruction [8]. Metis has now been integrated with Isabelle/HOL [16]. Hurd envisaged users calling Metis with a list of hand-chosen theorems, to be supplied as axiom clauses to assist the proof. Our integration, however, allows all known theorems to be considered as lemmas. Given a conjecture, we apply our simple relevance filter [11] to reduce the number of clauses from thousands to hundreds, and then call an ATP such as Vampire. From the resulting proof, we discover which lemmas were actually used, finally generating a Metis call referring to a few existing theorems. In other words, we use Vampire as a powerful relevance filter, making the problem small enough for Metis to prove it. Some five percent of problems are too difficult for Metis even with this reduction [16]. However, with a prover such as E that outputs TSTP format [23], we can use Metis to reconstruct each proof line individually. Each clause is translated to the corresponding Isabelle/HOL assertion; it is proved by a Metis call whose arguments refer to the proof lines justifying that inference. The idea is similar to that of the Otterfier proof transformation service [27], which pushes arbitrary resolution proofs through Otter.

Our implementation of proof reconstruction [16] is an instance of a general approach to converting TSTP proofs from an unsound translation to a sound one. The constant-typed and partially-typed translations contain enough information to reconstruct full types using standard type inference techniques [18]. Failure of type inference would indicate that the proof was unsound. Success would not necessarily produce a correct TSTP proof, as the reconstructed term could contain new type variables, so a final Otterfier phase might be necessary.

We approach the soundness problem pragmatically. Unsound proofs cannot produce false theorems in Isabelle, but they can block the discovery of well-typed proofs. Therefore, rather than giving our linkup *all* existing theorems, we filter out those having certain harmful features: specifically, expressions of finiteness. In particular, the Isabelle/HOL type `unit` contains only one element. It exists for technical reasons, but has few serious uses. Theorems containing variables of this type easily lead to unsound proofs, so we forbid them. The Isabelle/HOL type `bool` contains only two elements, and similarly leads to unsound proofs. Of course, the type of truth values is hugely more important than type `unit`. However, reasoning about truth values is the job of the ATP; lemmas concerning `bool` would allow higher-order reasoning in a few cases, but at excessive cost. An example may be helpful. The following two clauses allow us to derive Boolean equality ($P = Q$) from logical equivalence ($P \iff Q$).

```
B(P) | B(Q) | P=Q
~B(P) | ~B(Q) | P=Q
```

These clauses cause numerous unsound proofs unless we use the fully-typed translation. They might be useful for proving equations between sets coded as characteristic functions, but they do not appear to be relevant to many Isabelle proofs. We suppress certain other clauses that tend to cause unsound proofs. Typical are induction rules that are highly unlikely to yield bona fide proofs by induction; note that Beeson's achievements concerning induction [2] require modifying the ATP's unification algorithm.

3. Experiments

We have to choose between a sound but prolix treatment of types and two treatments that are more compact but admit ill-typed proofs. Applying the first-order optimization of §2.7 gives another sound and another unsound translation, for a total of five. Which translation allows the most sound proofs to be found? The answer can be found empirically, based on data presented below.

3.1. Experimental Setup

For our experiments, we took 153 problems generated by Isabelle, most of which contain higher-order features. Since our HOL translation can also be used for purely FOL problems and our experiments were aimed at testing the efficacy of the translation methods, we translated all problems (both HOL and FOL) using the three translation methods described in the previous section. We eliminated λ-abstractions by translating them to combinators. We used our relevance filter [11] to reduce each problem. We ran these tests on a bank of Dual AMD Opteron processors running at 2400MHz, using Condor[3] to manage our batch jobs.

Readers may wonder, however, how a comparison between sound and unsound translations can be fair. We have taken steps to ensure that no unsound proofs are being counted as successful.

Rather than integrating Isabelle with our test harness, which would be complicated, we simulate proof reconstruction using similar ideas. When a proof is found using an unsound translation, we can identify which axioms were actually used in the proof and generate a new problem by selecting those axioms from the fully-typed translation. Thus, we automatically convert each solved problem to a sound one, while greatly reducing the number of clauses. If some ATP can prove the converted problem, then we regard the original proof as sound.

We automated this process and applied it to all of the translations under test. We used all of the ATPs under consideration, in order to see whether any of them were finding unsound proofs. In each trial, Vampire confirmed at least 94 percent of the proofs. The remainder were supplied to other ATPs, and those still failing were inspected manually. Problems containing no conjecture clauses could immediately be classified as unsound. (No problem set contained more than two such cases.) A few problems looked correct but were large, so we devoted some time to deleting needless clauses. We eventually obtained machine

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

confirmation of the soundness of all the proofs that used conjecture clauses.

Therefore, in the graphs presented below, we count a proof as sound provided it uses at least one conjecture clause. In making this choice, we are discounting three risks as unlikely:

- We have only verified the soundness of proofs found by ATPs run with a limit of 300 seconds per problem. The graphs include proofs found by ATPs run with many smaller time limits. Conceivably some of these proofs are different from the ones we verified.
- Showing that the axioms used in the original proof can be used to express a well-typed proof does not ensure that the original proof was sound. This possibillity is not merely unlikely but harmless: Metis, given the fully-typed translation, will find only the sound proof.
- The ATPs themselves could be unsound.

The manual steps described in this section were done to make our graphs as accurate as possible. They also suggest steps that a user can take when proof reconstruction fails. Our approach to proof reconstruction is to deliver an Isabelle proof script based on the automatic resolution proof [16]. Users do occasionally simplify these scripts by hand. Of course, if the work environment requires full automation, failure of proof reconstruction simply means failure.

Unsound proofs look very different from sound ones. They typically bear no relation to the problem at hand, ignoring the conjecture clauses and instead finding a contradiction from unrelated axioms. The terms in these proofs are crazy combinations of functions of various types.

3.2. Results

Each graph compares the success rates of the five translations, for some prover, as the runtime per problem increases from 20 to 300 seconds. These short runtimes are appropriate for our application of ATPs to support interactive proofs.

- “Success rate” denotes the percentage of the 153 problems proved.
- “Runtime” denotes the time spent in the ATPs alone. The problem files were generated in advance using Isabelle. Translation time is negligible; other processing within Isabelle takes a few seconds per problem.

Table I. Common to All Translations

Number of clauses	329 (40 non-Horn; 126 unit)
Number of literals	681 (171 equality)
Maximal clause size	5 (2 average)

Table II. Differences between Three Translations

	Function symbols	*Max. term depth*	*Words*
constant (FO)	39 (7 constant)	6 (2 average)	3909
constant	42 (7 constant)	9 (3 average)	3909
partially	42 (39 constant)	11 (5 average)	9188
full (FO)	40 (12 constant)	13 (4 average)	5886
full	43 (39 constant)	18 (8 average)	12317

We tested three provers: E 0.99 “Singtom” (Fig. 1), SPASS 2.2 (Fig. 2) and Vampire 8 (Fig. 3). SPASS ran with SOS enabled and splitting disabled.[4] Although SOS makes SPASS incomplete, it greatly improves SPASS's success rate for our problems by making the proof search more goal-directed. Vampire ran with its CASC option, which is highly effective.

The graphs show that the constant-typed translation does indeed yield the highest success rate, while the fully-typed translation yields the lowest. The first-order optimization is beneficial to both translations. The optimized, constant-typed translation is clearly best for all three ATPs. SPASS gives the widest spread of success rates, depending on the translation used.

To obtain a quantitative picture of the differences between the three translations, we chose one of the problems and used tptp2X [22] to summarize its syntactic features. This problem is of median size in our problem set. It has 329 clauses after relevance filtering, of which 310 are non-trivial; the remaining 19 constitute a monadic Horn theory that describes Isabelle's type class system. Table I shows the figures common to all three translations. Table II shows variations among the translations. The figures shown under “Words” were obtained by the UNIX `wc` utility, since a problem's size is better measured in words than in bytes.

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

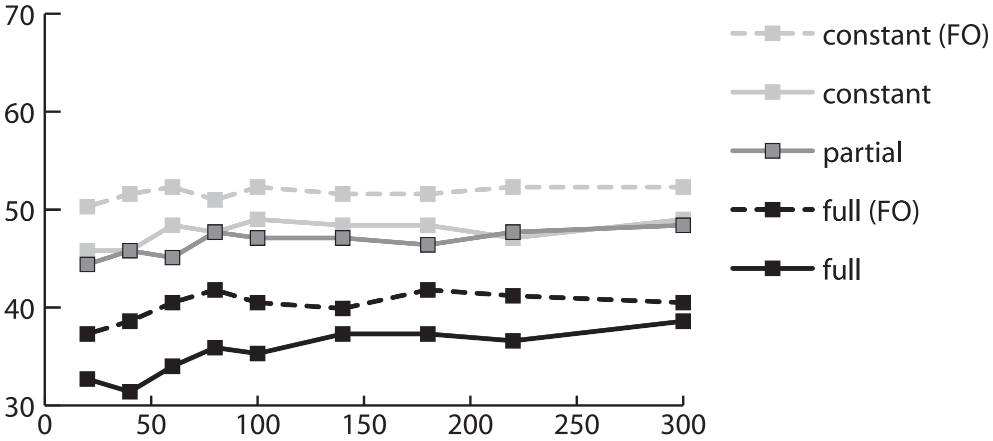

Figure 1. E, Version 0.99: percent solved against runtime

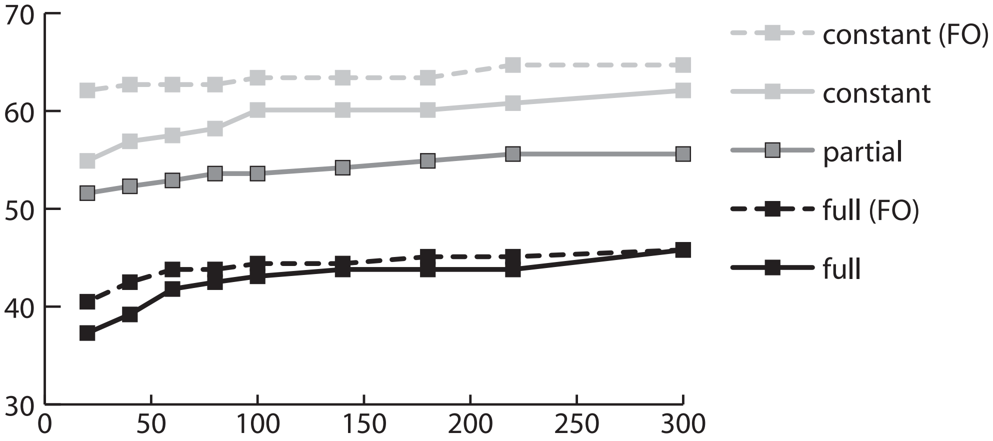

Figure 2. SPASS (SOS Enabled): percent solved against runtime

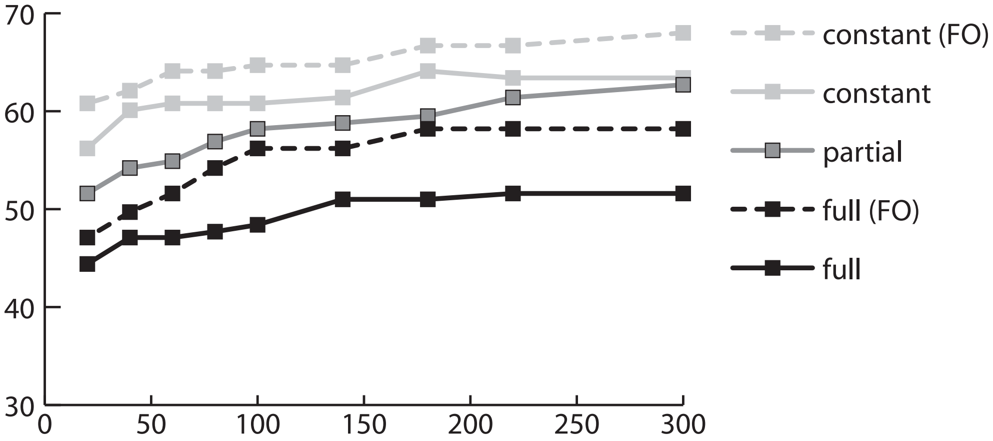

Figure 3. Vampire (CASC Mode): percent solved against runtime

A major difference is the maximal term depth, which increases monotonically as we move from the best-performing translation to the worst. Shallower terms are presumably less complex. The problem size in words also increases for the poorer-performing translations: the largest number exceeds the smallest by a factor of 3.2. For the fully-typed translation, the first-order optimization halves the number of words. This optimization also reduces the number of constants in the problem because it formalizes functions as functions rather than relying exclusively on the apply operator.

The workshop version of this paper [10] presents somewhat different results. In a few cases,[5] the partially-typed translation comes top. There are many differences between the two sets of experiments. Our problem set is larger: 153 rather than 79. Our current problem set includes the original 79 but adds many harder problems, especially to test reasoning about λ-expressions. We have refined the code that generates problems. In particular, we exclude clauses that we expect could harm the success rate, such as low-level definitions of certain primitives. We now use our relevance filter [11] rather than supplying substantially the same axioms to all problems.

When choosing a translation, we should also take account of soundness. As mentioned in §3.1 above, we tested all proofs for type correctness by using them to generate reduced, fully-typed versions. Nearly all proofs turned out to be sound, and all the bad ones had the tell-tale sign of using no conjecture clauses. Unsound proofs are not shown in the graphs. As expected, the fully-typed translation produced no unsound proofs. With the partially-typed translation, one proof was unsound. With the constant-typed translation, two proofs were unsound, but with the first-order optimization, no proofs were unsound. The additional freedom offered by the apply operator seems to lend itself mainly to unsound proofs. The optimized, constant-typed translation is obviously the best. If soundness is essential, then we suggest the optimized fully-typed one.

4. Translating λ-Abstractions: An Empirical Comparison

Most first-order provers cannot handle terms that contain λ-abstractions. The literature on functional programming discusses a variety of different methods for translating λ-abstractions. A well known approach is to use the five combinators **S**, **K**, **I**, **B** and **C**. The first two combinators suffice in theory, but they yield an output

[5] For Vampire 8.0, with a time limit below 300 seconds

that is exponential in the number of nested λ-abstractions. Even with all five combinators, the size of the output is quadratic [17]. They are inefficient: during a β-reduction, namely the evaluation of a function application, numerous occurrences of combinators must be expanded in the function body.

In an attempt to improve the success rate, we decided to experiment with two alternatives to the venerable Curry combinators: Turner's extended combinator set [24] and λ-lifting. We were surprised to discover that they yielded no convincing improvement. We feel that these experiments are of interest despite this outcome. Future research may improve upon these translations.

4.1. The Five Curry Combinators

S and **K** alone can express all λ-abstractions. The identity combinator, **I**, can be defined by **SKK**. As Turner relates [24], Curry improved upon this system by introducing two new combinators, **B** and **C**, to handle special cases of **S**. The full set can be defined as follows.

$$\begin{aligned}
\mathbf{I}\,x &= x \\
\mathbf{K}\,x\,y &= x \\
\mathbf{S}\,x\,y\,z &= x\,z(y\,z) \\
\mathbf{B}\,x\,y\,z &= x(y\,z) \\
\mathbf{C}\,x\,y\,z &= x\,z\,y
\end{aligned}$$

Note that $\mathbf{B}\,x\,y$ yields the function composition of x and y.

A λ-expression can be translated to combinators by the following rewrite rules.

$$\begin{aligned}
\lambda x.\,x &\longmapsto \mathbf{I} && \\
\lambda x.\,p &\longmapsto \mathbf{K}\,p && (x \text{ not free in } p) \\
\lambda x.\,p\,x &\longmapsto p && (x \text{ not free in } p) \\
\lambda x.\,p\,q &\longmapsto \mathbf{B}\,p\,(\lambda x.\,q) && (x \text{ not free in } p) \\
\lambda x.\,p\,q &\longmapsto \mathbf{C}\,(\lambda x.\,p)\,q && (x \text{ not free in } q) \\
\lambda x.\,p\,q &\longmapsto \mathbf{S}(\lambda x.\,p)(\lambda x.\,q) && (x \text{ free in } p \text{ and } q)
\end{aligned}$$

Unfortunately, this translation is quadratic in the nesting depth of λ-abstractions. For example, consider the translation of the expression $\lambda xyz.\,pq$, where p and q may be arbitrary terms. As each abstraction is translated, the prefix grows dramatically.

$$\begin{aligned}\lambda xyz.\,pq &\longmapsto \lambda xy.\,\mathbf{S}(\lambda z.\,p)(\lambda z.\,q)\\ &\longmapsto \lambda x.\,\mathbf{S}(\mathbf{B}\,\mathbf{S}(\lambda yz.\,p))(\lambda yz.\,q)\\ &\longmapsto \mathbf{S}(\mathbf{B}\,\mathbf{S}(\mathbf{B}(\mathbf{B}\,\mathbf{S})(\lambda xyz.\,p)))(\lambda xyz.\,q)\end{aligned}$$

Three nested λ-abstractions arise quite easily in our examples, so it is natural to seek improvements.

4.2. The Turner Combinators

Turner [24] introduced three new combinators, $\mathbf{S}'$, $\mathbf{B}'$ and $\mathbf{C}'$, in order to address the quadratic behaviour shown above. Peyton Jones [17] claims that Turner's system can be further improved if $\mathbf{B}'$ is replaced by $\mathbf{B}^*$, yielding the following definitions of the new combinators.[6]

$$\begin{aligned}\mathbf{S}'\,w\,x\,y\,z &= w(x\,z)(y\,z)\\ \mathbf{B}^*\,w\,x\,y\,z &= w(x(y\,z))\\ \mathbf{C}'\,w\,x\,y\,z &= w(x\,z)\,y\end{aligned}$$

This system is used by first translating to the Curry combinators, then simplifying the result by applying the following optimizations:

$$\begin{aligned}\mathbf{B}\,p\,(\mathbf{B}\,q\,r) &= \mathbf{B}^*\,p\,q\,r\\ \mathbf{C}(\mathbf{B}\,p\,q)r &= \mathbf{C}'\,p\,q\,r\\ \mathbf{S}(\mathbf{B}\,p\,q)r &= \mathbf{S}'\,p\,q\,r\end{aligned}$$

The optimized result is more compact and allows shorter derivations. This might be expected to yield a higher success rate.

4.3. Defining New Functions: λ-lifting

The idea of λ-lifting is that the functions actually present in the expression being translated should serve as the combinators [6]. These are sometimes called *super-combinators.* No built-in combinators are required. Instead, λ-abstractions are translated from the inside out. Each abstraction is replaced by a call to a newly defined function. This function obviously has as arguments those of the λ-abstraction, with additional arguments for all variables free in that abstraction. With λ-lifting, we can expect a compact output. (See Appendix A.5 for an example.) Moreover, β-reduction should take only one rewriting step rather than many.

During our experiments, it became clear that λ-lifting delivers poor results unless several points are noted:

[6] Turner's $\mathbf{B}'$ satisfies $\mathbf{B}'\,w\,x\,y\,z = wx(y\,z)$.

- Every occurrence of $\lambda x.fx$, where x is not free in f, should be replaced by f.

- An equation between a constant and a λ-expression should be translated directly to another equation. For example, the formula $h = \lambda x.\, f(x, g(x))$ should be translated to $h(x) = f(x, g(x))$.

- Multiple abstractions can be translated as a unit. Although the general treatment of nested abstractions will yield a correct result, it will needlessly introduce a series a function definitions.

- Existing function definitions must be re-used as often as possible.

In some of these points, we deviate from Hughes [6]. He did not combine multiple abstractions. For the additional arguments, he used not the free variables but the maximal free subexpressions. His prime motivation was to obtain fully lazy evaluation, which is of no concern to us. However, in future work we may experiment with his approach. Our current λ-lifting algorithm is straightforward.

- Traverse the formula recursively.

- If the abstraction $\lambda x_1 \ldots x_n.\, t$ is encountered, where t does not begin with a λ, then recursively perform λ-lifting on t, yielding the the λ-free term t'. Let $y_1, \ldots, y_m$ be variables that are free in $\lambda x_1 \ldots x_n.\, t'$.

- Choose a new function symbol f, define it by $f y_1 \ldots y_m\, x_1 \ldots x_n = t'$, and return $f y_1 \ldots y_m$ as the translation. However, if an instance of an existing function symbol can express $\lambda x_1 \ldots x_n.\, t'$, use that instead of defining a new function.

Re-use of existing functions is essential because equal abstractions should have identical translations. Combinators automatically have this valuable property: because they are syntax directed, they are guaranteed to give the same output for the same input. Even with different inputs, the combinator translations sometimes have a similar form, as in $\mathbf{K}X$ and $\mathbf{K}(\mathbf{SK})$. With λ-lifting, the outputs will always be different unless functions are re-used. Such differences can only be overcome in a proof by an explicit application of extensionality:

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

This theorem states that functions f and g are equal provided they deliver the same results for equal arguments. Proofs take much longer, and often fail, if they are forced to perform such a step.[7]

4.4. Experimental Results for the λ-Translations

The first question to settle is whether our three (or five) translations obey the same ranking with λ-lifting as they do with combinators. As Figs. 4 and 5 indicate, they largely do. It is surprising, however, that the two constant-typed translations are approximately equal for Vampire (Fig. 6).

To compare λ-lifting with combinators, we first consider the optimized constant-typed translation. Fig. 7 plots six graphs, showing the results for E, SPASS and Vampire with both treatments of abstractions. This graph shows some interesting effects. For E, λ-lifting delivers a clear benefit. For SPASS, λ-lifting delivers worse results, to a similar degree. For Vampire, λ-lifting is slightly inferior. This graph also shows that E delivers poor results with all of our higher-order translations; this is puzzling, since with our first-order translations it is generally superior to SPASS [11].

The comparison of λ-lifting with combinators is complicated by its interaction with our relevance filter. Relevance filtering behaves differently with λ-lifting because the abstraction functions are declared as "real" functions early in the translation to clauses. With our current problem set, we have found that relevance filtering makes several easy problems impossible by omitting essential axioms, so this could be biasing the results.

To eliminate this bias, we have also run tests with relevance filtering switched off. An unfiltered problem typically contains 8500 clauses. With such large axiom sets, the constant-typed translation admits far too many unsound proofs. In a typical test, all 153 problems were "proved", but only 53 (or 35 percent) of these proofs used any conjecture clauses. We therefore use the optimized, fully-typed translation for this test (Fig. 8). Now λ-lifting delivers a small benefit for both E and SPASS, but a deficit for Vampire. The dismal success rates demonstrate the contributions of our work on both relevance filtering [11] and higher-order translations.

Thus, λ-lifting is beneficial with two automatic provers, but is harmful with a third. This finding is surprising, given that the combinator translation is quadratic. Of course, theorem proving is very different

[7] Joe Hurd tells us that the very presence of the extensionality axiom in a problem greatly harms the success rate. We ran experiments with E, SPASS and Vampire, but found that the presence of this axiom had no effect on the success rate.

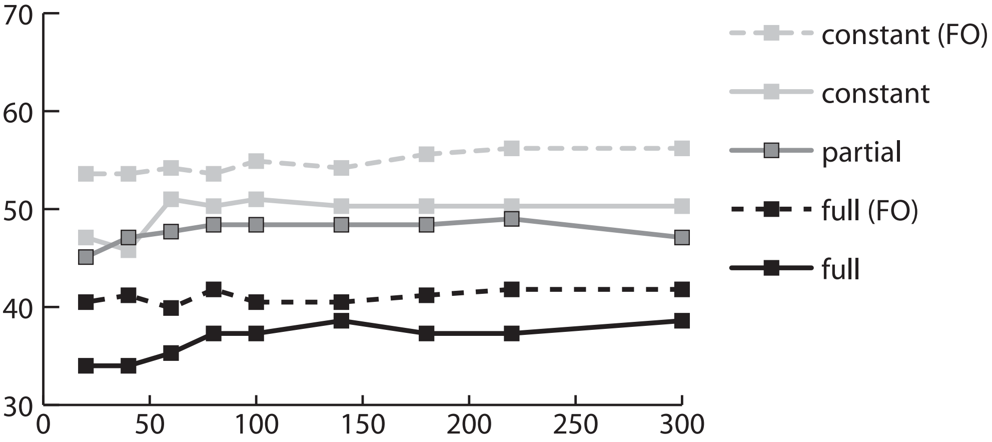

Figure 4. E, Version 0.99: percent solved against runtime

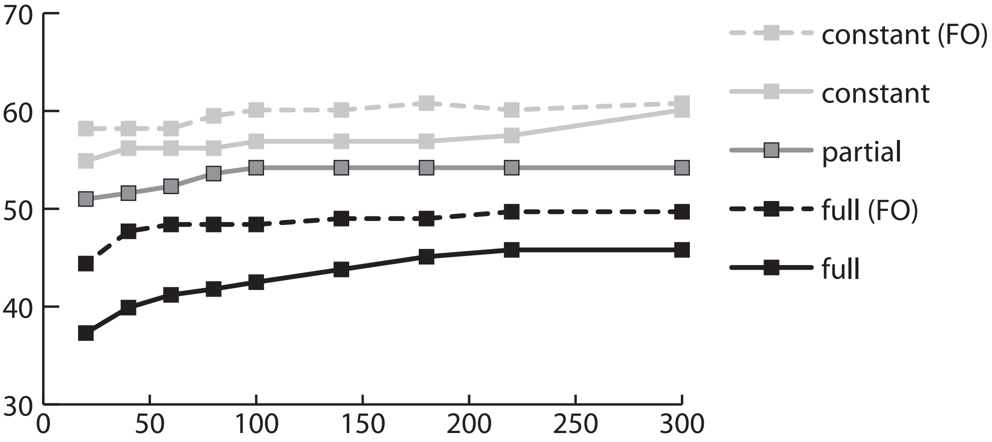

Figure 5. SPASS (SOS Enabled): percent solved against runtime

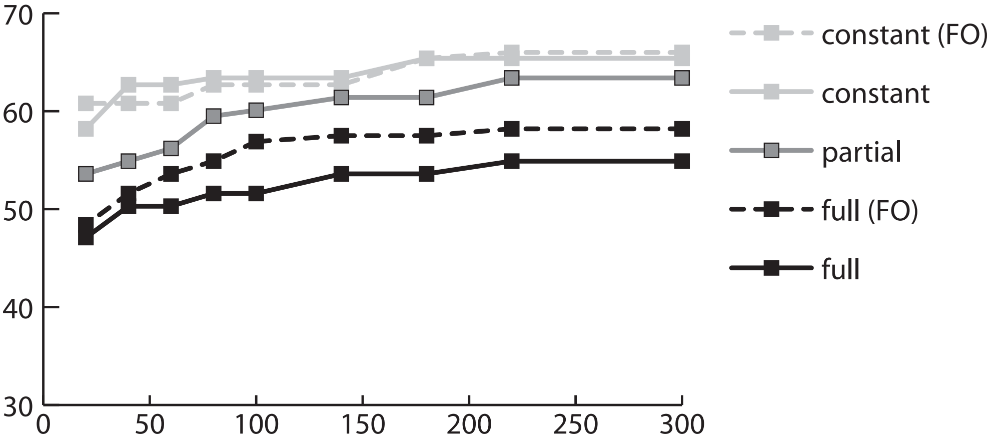

Figure 6. Vampire (CASC Mode): percent solved against runtime

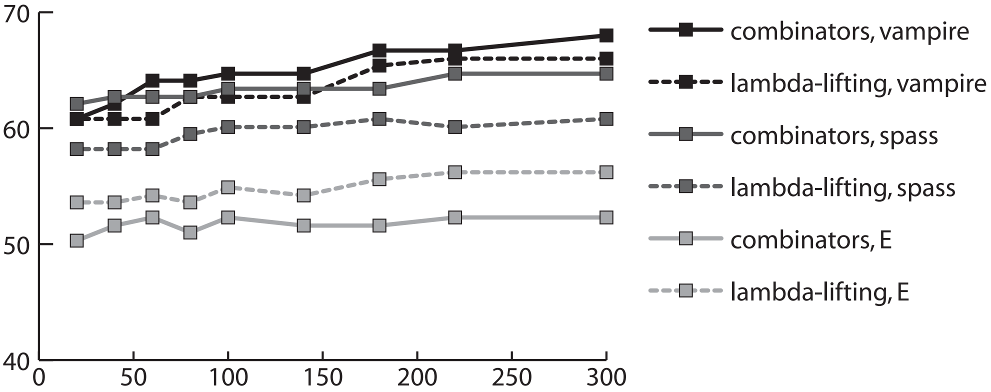

Figure 7. λ-lifting versus combinators (relevance filtering)

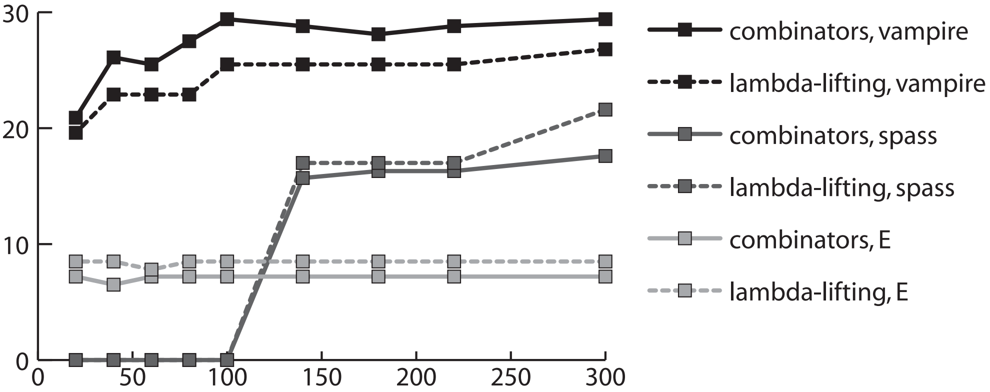

Figure 8. λ-lifting versus combinators (no relevance filtering)

from functional programming, which is concerned entirely with the reduction of huge λ-expresssions. Few of our problems have even three nested λ-abstractions, and many of them remain unsolved with both approaches. Note that λ-lifting generates many equations, compared with the five equations defining the combinators: perhaps this degrades Vampire's performance.

We were very surprised to find that Turner's combinators yielded no improvement over Curry's. They performed worse by a tiny margin. Upon closer investigation, we discovered that Turner's optimizations were not yielding dramatic reductions in the size of the output. Despite Turner's claim [24, p. 269] that "the sizes of the successive terms now [form] only a linear progression," the translation as a whole is still quadratic [9]. We observed reductions of approximately 30%, in the number of combinators produced. Set against this modest reduction is the larger number of combinator equations that must be used in proofs.

5. Conclusions

We have described three HOL to FOL translations, which differ in their treatment of types. Two of these admit a "first-order" optimization, which uses real function application whenever possible, so the number of translations is effectively five. We have carried out extensive experiments to evaluate the effectiveness of these translations. We have also obtained some statistics concerning how compact our translations are. Of the three translations—fully-typed, partially-typed and constant-typed—the constant-typed translation produces the most compact output. Our optimization, which applies to the fully-typed and constant-typed translations, reduces the term depth and the problem size. Naturally, we would expect a prover's success rate to increase with a more compact clause form. That is what we observed, with all three provers tested. The difference between best and worst was up to 20 percentage points.

Because only the fully-typed translations are sound, proofs found using the other translations must be validated in some way, such as by proof reconstruction. The proportion of unsound proofs depends on which and how many axioms are present. In our experiments, it ranged from zero (for our standard configuration) to 65 percent (for huge problems containing thousands of irrelevant axioms). Therefore, the choice of translation must be made with care.

We have implemented methods for using the sound and unsound translations in concert. A proof found using an unsound translation can be used to generate a version of the same problem using a sound translation, but containing only the axioms necessary for the proof. Thus the unsound translation is used as a means of relevance filtering, which improves the success rate of the sound but prolix translation. This method simulates, using resolution alone, the approach to proof reconstruction implemented in Isabelle.

We also compared three approaches to eliminating λ-abstractions: the five Curry combinators, the eight Turner combinators (modified), and λ-lifting. We obtained evidence in favour of λ-lifting, but it was inconclusive and we expect further gains to be made here.

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.

The test data used in our experiments is available at `http://www.cl.cam.ac.uk/~lp15/Data/ho-translations/`.

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. Christoph Benzmüller and various referees made many useful suggestions for improving this paper.

References

1. David Basin and Michaël Rusinowitch, editors. *Automated Reasoning — Second International Joint Conference, IJCAR 2004*, LNAI 3097. Springer, 2004.
2. Michael Beeson. Mathematical induction in Otter-lambda. *J. Auto. Reas.*, 36(4):311–344, 2006.
3. C. Benzmüller, V. Sorge, M. Jamnik, and M. Kerber. Can a higher-order and a first-order theorem prover cooperate? In Franz Baader and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning — 11th International Workshop, LPAR 2004*, LNAI 3452, pages 415–431. Springer, 2005.
4. Charles Bouillaguet, Viktor Kuncak, Thomas Wies, Karen Zee, and Martin Rinard. Using first-order theorem provers in the Jahob data structure verification system. In Byron Cook and Andreas Podelski, editors, *Verification, Model Checking, and Abstract Interpretation*, LNCS 4349, pages 74–88. Springer, 2007.
5. Michael J. C. Gordon and Thomas F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic.* Cambridge Univ. Press, 1993.
6. R. J. M. Hughes. Supercombinators: A new implementation method for applicative languages. In *LISP and Func. Prog.* ACM Press, 1982.
7. 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.
8. 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.
9. Richard Kennaway and Ronan Sleep. Director strings as combinators. *ACM Trans. Program. Lang. Syst.*, 10(4):602–626, 1988.
10. Jia Meng and Lawrence C. Paulson. Translating higher-order problems to first-order clauses. In Sutcliffe et al. [21], pages 70–80.
11. Jia Meng and Lawrence C. Paulson. Lightweight relevance filtering for machine-generated resolution problems. *Journal of Applied Logic*, in press.
12. Jia Meng, Claire Quigley, and Lawrence C. Paulson. Automation for interactive proof: First prototype. *Information and Computation*, 204(10):1575–1596, 2006.
13. Dale Miller. Unification under a mixed prefix. *J. Symb. Comput.*, 14(4):321–358, 1992.

14. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic.* Springer, 2002. LNCS Tutorial 2283.
15. 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.
16. Lawrence C. Paulson and Kong Woei Susanto. Source-level proof reconstruction for interactive theorem proving. In Jens Brandt, editor, *Theorem Proving in Higher Order Logics.* Springer, 2007. In press.
17. Simon L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice Hall, 1987.
18. Benjamin C. Pierce. *Types and Programming Languages.* MIT Press, 2002.
19. 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.
20. Stephan Schulz. System description: E 0.81. In Basin and Rusinowitch [1], pages 223–228.
21. Geoff Sutcliffe, Renate Schmidt, and Schulz Schulz, editors. *FLoC'06 Workshop on Empirically Successful Computerized Reasoning*, volume 192 of *CEUR Workshop Proceedings*, 2006.
22. Geoff Sutcliffe and Christian Suttner. The TPTP problem library for automated theorem proving. On the Internet at `http://www.cs.miami.edu/~tptp/`, 2004.
23. Geoff Sutcliffe, Jürgen Zimmer, and Stephan Schulz. TSTP data-exchange formats for automated theorem proving tools. In W. Zhang and V. Sorge, editors, *Distributed Constraint Problem Solving and Reasoning in Multi-Agent Systems*, number 112 in Frontiers in Artificial Intelligence and Applications, pages 201–215. IOS Press, 2004.
24. D. A. Turner. Another algorithm for bracket abstraction. *Journal of Symbolic Logic*, 44(2):267–270, June 1979.
25. David A. Turner. A new implementation technique for applicative languages. *Softw. Pract. Exper.*, 9:31–49, 1979.
26. 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.
27. J. Zimmer, A. Meier, G. Sutcliffe, and Y. Zhang. Integrated proof transformation services. In C. Benzmüller and W. Windsteiger, editors, *Workshop on Computer-Supported Mathematical Theory Development, 2nd International Joint Conference on Automated Reasoning*, Electronic Notes in Theoretical Computer Science, 2004.

Appendix

A. Examples of the Various Translations

To illustrate the effects of the translations, we present all five variants using combinators, and one variant using λ-lifting. The conjecture being proved is the base case of a structural induction on lists:

```
map (λx. (f x, g x)) [] = zip (map f xs) (map g [])
```

It relates two familiar list functions: `map`, which applies a function to every element of a list, and `zip`, which combines a pair of lists to yield a list of pairs. This base case is trivial because both sides collapse to the empty list.

The three clauses include the negation of the conjecture above and two trivial properties of our functions:

```
map f [] = []
zip xs [] = []
```

Only necessary clauses are presented. The clause set delivered to ATPs contains approximately three hundred. We have reformatted the text slightly to improve readability.

A.1. The Fully-Typed Translation

```
cnf(cls_conjecture_0,negated_conjecture,
    (ti(hAPP(ti(hAPP(ti(c_List_Omap, tc_fun(tc_fun(t_c,
     tc_prod(t_a, t_b)), tc_fun(tc_List_Olist(t_c),
     tc_List_Olist(tc_prod(t_a, t_b))))), ti(hAPP(ti(hAPP(ti(c_COMBS,
     tc_fun(tc_fun(t_c, tc_fun(t_b, tc_prod(t_a, t_b))),
     tc_fun(tc_fun(t_c, t_b), tc_fun(t_c, tc_prod(t_a, t_b))))),
     ti(hAPP(ti(hAPP(ti(c_COMBB, tc_fun(tc_fun(t_a, tc_fun(t_b,
     tc_prod(t_a, t_b))), tc_fun(tc_fun(t_c, t_a), tc_fun(t_c,
     tc_fun(t_b, tc_prod(t_a, t_b)))))), ti(c_Pair, tc_fun(t_a,
     tc_fun(t_b, tc_prod(t_a, t_b))))), tc_fun(tc_fun(t_c, t_a),
     tc_fun(t_c, tc_fun(t_b, tc_prod(t_a, t_b))))), ti(v_f,
     tc_fun(t_c, t_a))), tc_fun(t_c, tc_fun(t_b, tc_prod(t_a,
     t_b))))), tc_fun(tc_fun(t_c, t_b), tc_fun(t_c, tc_prod(t_a,
     t_b)))), ti(v_g, tc_fun(t_c, t_b))), tc_fun(t_c, tc_prod(t_a,
     t_b)))), tc_fun(tc_List_Olist(t_c), tc_List_Olist(tc_prod(t_a,
     t_b)))), ti(c_List_Olist_ONil, tc_List_Olist(t_c))),
     tc_List_Olist(tc_prod(t_a, t_b))) !=
     ti(hAPP(ti(hAPP(ti(c_List_Ozip, tc_fun(tc_List_Olist(t_a),
     tc_fun(tc_List_Olist(t_b), tc_List_Olist(tc_prod(t_a, t_b))))),
     ti(hAPP(ti(hAPP(ti(c_List_Omap, tc_fun(tc_fun(t_c, t_a),
     tc_fun(tc_List_Olist(t_c), tc_List_Olist(t_a)))), ti(v_f,
     tc_fun(t_c, t_a))), tc_fun(tc_List_Olist(t_c),
     tc_List_Olist(t_a))), ti(c_List_Olist_ONil,
     tc_List_Olist(t_c))), tc_List_Olist(t_a))),
     tc_fun(tc_List_Olist(t_b), tc_List_Olist(tc_prod(t_a, t_b)))),
     ti(hAPP(ti(hAPP(ti(c_List_Omap, tc_fun(tc_fun(t_c, t_b),
     tc_fun(tc_List_Olist(t_c), tc_List_Olist(t_b)))), ti(v_g,
     tc_fun(t_c, t_b))), tc_fun(tc_List_Olist(t_c),
     tc_List_Olist(t_b))), ti(c_List_Olist_ONil,
     tc_List_Olist(t_c))), tc_List_Olist(t_b))),
     tc_List_Olist(tc_prod(t_a, t_b))))).
cnf(cls_map_Osimps_I1_J_0,axiom,
    (ti(hAPP(ti(hAPP(ti(c_List_Omap, tc_fun(tc_fun(T_b__1,
     T_a__1), tc_fun(tc_List_Olist(T_b__1), tc_List_Olist(T_a__1)))),
     ti(V_f, tc_fun(T_b__1, T_a__1))), tc_fun(tc_List_Olist(T_b__1),
     tc_List_Olist(T_a__1))), ti(c_List_Olist_ONil,
     tc_List_Olist(T_b__1))), tc_List_Olist(T_a__1)) =
     ti(c_List_Olist_ONil, tc_List_Olist(T_a__1)))).
cnf(cls_zip_Osimps_I1_J_0,axiom,
    (ti(hAPP(ti(hAPP(ti(c_List_Ozip,
     tc_fun(tc_List_Olist(T_a__1), tc_fun(tc_List_Olist(T_b__1),
     tc_List_Olist(tc_prod(T_a__1, T_b__1))))), ti(V_xs,
     tc_List_Olist(T_a__1))), tc_fun(tc_List_Olist(T_b__1),
     tc_List_Olist(tc_prod(T_a__1, T_b__1)))), ti(c_List_Olist_ONil,
     tc_List_Olist(T_b__1))), tc_List_Olist(tc_prod(T_a__1, T_b__1))) =
     ti(c_List_Olist_ONil, tc_List_Olist(tc_prod(T_a__1, T_b__1))))).
```

A.2. The Fully-Typed Translation (Optimized)

Here we can see the tremendous reduction achieved by the first-order optimization. It is even more compact than the partially-typed version.

```
cnf(cls_conjecture_0,negated_conjecture,
    (ti(c_List_Omap(ti(c_COMBS(ti(hAPP(hAPP(c_COMBB, ti(c_Pair,
     tc_fun(t_a, tc_fun(t_b, tc_prod(t_a, t_b))))), ti(v_f,
     tc_fun(t_c, t_a))), tc_fun(t_c, tc_fun(t_b, tc_prod(t_a,
     t_b)))), ti(v_g, tc_fun(t_c, t_b))), tc_fun(t_c, tc_prod(t_a,
     t_b))), ti(c_List_Olist_ONil, tc_List_Olist(t_c))),
     tc_List_Olist(tc_prod(t_a, t_b))) !=
     ti(hAPP(hAPP(c_List_Ozip, ti(c_List_Omap(ti(v_f,
     tc_fun(t_c, t_a)), ti(c_List_Olist_ONil, tc_List_Olist(t_c))),
     tc_List_Olist(t_a))), ti(c_List_Omap(ti(v_g, tc_fun(t_c, t_b)),
     ti(c_List_Olist_ONil, tc_List_Olist(t_c))),
     tc_List_Olist(t_b))), tc_List_Olist(tc_prod(t_a, t_b))))).
cnf(cls_map_Osimps_I1_J_0,axiom,
    (ti(c_List_Omap(ti(V_f, tc_fun(T_b__1, T_a__1)),
     ti(c_List_Olist_ONil, tc_List_Olist(T_b__1))),
     tc_List_Olist(T_a__1)) =
     ti(c_List_Olist_ONil, tc_List_Olist(T_a__1)))).
cnf(cls_zip_Osimps_I1_J_0,axiom,
    (ti(hAPP(hAPP(c_List_Ozip, ti(V_xs, tc_List_Olist(T_a__1))),
     ti(c_List_Olist_ONil, tc_List_Olist(T_b__1))),
     tc_List_Olist(tc_prod(T_a__1, T_b__1))) =
     ti(c_List_Olist_ONil, tc_List_Olist(tc_prod(T_a__1, T_b__1))))).
```

A.3. The Partially-Typed Translation

```
cnf(cls_conjecture_0,negated_conjecture,
    (hAPP(hAPP(c_List_Omap, hAPP(hAPP(c_COMBS,
     hAPP(hAPP(c_COMBB, c_Pair, tc_fun(tc_fun(t_a, tc_fun(t_b,
     tc_prod(t_a, t_b))), tc_fun(tc_fun(t_c, t_a), tc_fun(t_c,
     tc_fun(t_b, tc_prod(t_a, t_b)))))), v_f, tc_fun(tc_fun(t_c,
     t_a), tc_fun(t_c, tc_fun(t_b, tc_prod(t_a, t_b))))),
     tc_fun(tc_fun(t_c, tc_fun(t_b, tc_prod(t_a, t_b))),
     tc_fun(tc_fun(t_c, t_b), tc_fun(t_c, tc_prod(t_a, t_b))))), v_g,
     tc_fun(tc_fun(t_c, t_b), tc_fun(t_c, tc_prod(t_a, t_b)))),
     tc_fun(tc_fun(t_c, tc_prod(t_a, t_b)),
     tc_fun(tc_List_Olist(t_c), tc_List_Olist(tc_prod(t_a, t_b))))),
     c_List_Olist_ONil, tc_fun(tc_List_Olist(t_c),
     tc_List_Olist(tc_prod(t_a, t_b)))) !=
     hAPP(hAPP(c_List_Ozip, hAPP(hAPP(c_List_Omap, v_f,
     tc_fun(tc_fun(t_c, t_a), tc_fun(tc_List_Olist(t_c),
     tc_List_Olist(t_a)))), c_List_Olist_ONil,
     tc_fun(tc_List_Olist(t_c), tc_List_Olist(t_a))),
     tc_fun(tc_List_Olist(t_a), tc_fun(tc_List_Olist(t_b),
     tc_List_Olist(tc_prod(t_a, t_b))))), hAPP(hAPP(c_List_Omap, v_g,
     tc_fun(tc_fun(t_c, t_b), tc_fun(tc_List_Olist(t_c),
     tc_List_Olist(t_b)))), c_List_Olist_ONil,
     tc_fun(tc_List_Olist(t_c), tc_List_Olist(t_b))),
     tc_fun(tc_List_Olist(t_b), tc_List_Olist(tc_prod(t_a, t_b))))).
cnf(cls_map_Osimps_I1_J_0,axiom,
    (hAPP(hAPP(c_List_Omap, V_f, tc_fun(tc_fun(T_b__1, T_a__1),
     tc_fun(tc_List_Olist(T_b__1), tc_List_Olist(T_a__1)))),
     c_List_Olist_ONil, tc_fun(tc_List_Olist(T_b__1),
     tc_List_Olist(T_a__1))) =
     c_List_Olist_ONil)).
cnf(cls_zip_Osimps_I1_J_0,axiom,
    (hAPP(hAPP(c_List_Ozip, V_xs, tc_fun(tc_List_Olist(T_a__1),
     tc_fun(tc_List_Olist(T_b__1), tc_List_Olist(tc_prod(T_a__1,
     T_b__1))))), c_List_Olist_ONil, tc_fun(tc_List_Olist(T_b__1),
     tc_List_Olist(tc_prod(T_a__1, T_b__1)))) =
     c_List_Olist_ONil)).
```

A.4. The Constant-Typed Translation

```
cnf(cls_conjecture_0,negated_conjecture,
    (hAPP(hAPP(c_List_Omap(t_c, tc_prod(t_a, t_b)),
     hAPP(hAPP(c_COMBS(t_c, t_b, tc_prod(t_a, t_b)),
     hAPP(hAPP(c_COMBB(t_a, tc_fun(t_b, tc_prod(t_a, t_b)), t_c),
     c_Pair(t_a, t_b)), v_f)), v_g)), c_List_Olist_ONil(t_c)) !=
     hAPP(hAPP(c_List_Ozip(t_a, t_b), hAPP(hAPP(c_List_Omap(t_c,
     t_a), v_f), c_List_Olist_ONil(t_c))), hAPP(hAPP(c_List_Omap(t_c,
     t_b), v_g), c_List_Olist_ONil(t_c))))).
cnf(cls_map_Osimps_I1_J_0,axiom,
    (hAPP(hAPP(c_List_Omap(T_b__1, T_a__1), V_f),
          c_List_Olist_ONil(T_b__1)) =
     c_List_Olist_ONil(T_a__1))).
cnf(cls_zip_Osimps_I1_J_0,axiom,
    (hAPP(hAPP(c_List_Ozip(T_a__1, T_b__1), V_xs),
          c_List_Olist_ONil(T_b__1)) =
     c_List_Olist_ONil(tc_prod(T_a__1, T_b__1)))).
```

A.5. The Constant-Typed Translation (Optimized)

This version is almost legible. Here, we compare combinators with λ-lifting.

```
cnf(cls_conjecture_0,negated_conjecture,
    (c_List_Omap(c_COMBS(hAPP(hAPP(c_COMBB(t_a, tc_fun(t_b,
     tc_prod(t_a, t_b)), t_c), c_Pair(t_a, t_b)), v_f), v_g, t_c,
     t_b, tc_prod(t_a, t_b)), c_List_Olist_ONil(t_c), t_c,
     tc_prod(t_a, t_b)) !=
     hAPP(hAPP(c_List_Ozip(t_a, t_b), c_List_Omap(v_f,
     c_List_Olist_ONil(t_c), t_c, t_a)), c_List_Omap(v_g,
     c_List_Olist_ONil(t_c), t_c, t_b)))).
cnf(cls_map_Osimps_I1_J_0,axiom,
    (c_List_Omap(V_f, c_List_Olist_ONil(T_b__1), T_b__1, T_a__1) =
     c_List_Olist_ONil(T_a__1))).
cnf(cls_zip_Osimps_I1_J_0,axiom,
    (hAPP(hAPP(c_List_Ozip(T_a__1, T_b__1), V_xs),
          c_List_Olist_ONil(T_b__1)) =
     c_List_Olist_ONil(tc_prod(T_a__1, T_b__1)))).
```

Here is the same problem, using λ-lifting. The second conjecture clause defines the function `v_llabs__subgoal__1` corresponding to the abstraction in the problem. In this proof it is not actually required, because `map f [] = []`.

```
cnf(cls_conjecture_0,negated_conjecture,
    (c_List_Omap(hAPP(hAPP(v_llabs__subgoal__1, v_f), v_g),
                 c_List_Olist_ONil(t_c), t_c, tc_prod(t_a, t_b)) !=
     c_List_Ozip(c_List_Omap(v_f, c_List_Olist_ONil(t_c), t_c, t_a),
                 c_List_Omap(v_g, c_List_Olist_ONil(t_c), t_c, t_b),
                 t_a, t_b))).
cnf(cls_conjecture_1,negated_conjecture,
    (hAPP(hAPP(hAPP(v_llabs__subgoal__1, V_f), V_g), V_x) =
     c_Pair(hAPP(V_f, V_x), hAPP(V_g, V_x), t_a, t_b))).
```

```
cnf(cls_map_Osimps_I1_J_0,axiom,
    (c_List_Omap(V_f, c_List_Olist_ONil(T_b__1), T_b__1, T_a__1) =
     c_List_Olist_ONil(T_a__1))).
cnf(cls_zip_Osimps_I1_J_0,axiom,
    (c_List_Ozip(V_xs, c_List_Olist_ONil(T_b__1), T_a__1, T_b__1) =
     c_List_Olist_ONil(tc_prod(T_a__1, T_b__1)))).
```