Very Efficient Conversions

Morten Welinder*

Carnegie Mellon University
School of Computer Science
5000 Forbes Avenue, Pittsburgh, PA–15213, USA
Email: Morten.Welinder@cs.cmu.edu

Abstract. Using program transformation techniques from the field of partial evaluation an automatic tool for generating very efficient conversions from equality-stating theorems has been implemented.
In the situation where a HOL user would normally employ the built-in function `GEN_REWRITE_CONV`, a function that directly produces a conversion of the desired functionality, this article demonstrates how producing the conversion in the form of a program text instead of as a closure can lead to significant speed-ups.
The HOL system uses a set of 31 simplifying equations on a very large number of intermediate terms derived, e.g., during backwards proofs. For this set the conversion generated by the two-step method is about twice as fast as the method currently used. When installing the new conversion, tests show that the *overall* running times of HOL proofs are reduced by about 10%. Apart from the speed-up this is completely invisible to the user. With cooperation from the user further speed-up is possible.

1 Introduction

A *conversion* in HOL [GM93] is a function that takes as its argument a term, called the *object term,* and produces a theorem, presumably stating that the object term equals some other term. Conversions span from the trivial `REFL` simply expressing that any term equals itself to functions that repeatedly traverse a term doing rewriting on subterms.

In many applications of HOL, for instance during backwards proofs, it seems natural to use conversions to simplify terms because these simplifications can be very tedious to do by hand. By using simplifying conversions the user can focus on the problematic parts of the proofs.

Given one conversion, say beta reduction, the HOL system provides the user with a number of conversion augmenting functions. For example there is `REPEATC` which from beta conversion will produce a conversion that will do repeated beta conversion at a term's top level, and there is `REDEPTH_CONV` that will produce a conversion which will perform exhaustive beta conversion in a term and its

* On leave from Department of Computer Science, University of Copenhagen, Denmark. Partially supported by The Danish Research Academy.

subterms. Since these augmenting functions, thoroughly described by Paulson in [Pau83], are both elegantly and quite efficiently implemented in the current HOL system this article will focus on conversions that do whatever they do once at the top of a term. The implementation relies on ideas by Roger Fleming and Richard Boulton, see [Bou94, Sli95].

Several versions of HOL exist and the methods presented here should work with any of them, and even with some other theorem provers. The presentation and the implementation, however, are based on a single version namely HOL-90.7. Since the efficiency of HOL-functions varies from implementation to implementation, arguments about the relative efficiency of methods should be understood in the HOL-90.7 setting. They are, however, believed to hold for all HOL implementations.

1.1 Overview

We will start with describing the particular class of conversions that we will deal with in this paper. This is done in Section 2 and is followed by a short description of the method that HOL currently uses for generating such conversions.

In Section 3 we will then describe how a variant of the HOL system's current one-step method can be turned into a two-step method. Pros and cons of doing so will be discussed.

Then, in Section 3.3, we will describe how the HOL system can benefit from conversions generated by the two-step method. We will also show how a cooperating user can gain even better results.

Finally we will draw our conclusions in Section 4 and sketch directions for future work.

2 Conversions from Equalities

An important subset of all conversions is the set of *rewriting conversions.* Such a conversion is based on one or more equality-stating theorems, possibly universally quantified, and works by matching the object term against the left-hand sides of the theorems until a match is found. When a match is found, the free (or universally bound) variables of the theorem are instantiated according to the match yielding the desired result. If none of the given theorems can be matched an exception is raised. As we are looking for instantiations (i.e., substitutions of terms for free variables) of the theorems' left-hand sides, we will call these left-hand sides *patterns.*

It is important to realize that the result of applying a rewriting conversion to a term is *not* just the simplified term, but a theorem evincing the validity of the rewriting.

Consider, for example, the following two theorems describing addition on Peano numbers:

$$\vdash \forall n.0 + n = n \tag{1}$$

$$\vdash \forall m.\forall n.\,\mathtt{SUC}\,m + n = \mathtt{SUC}(m + n) \tag{2}$$

(As theorems often come in groups we will from now on sometimes present them as conjunctions. This should be considered syntactic sugar and a pre-processor will eliminate it. We will still count each equality as a theorem.) The rewriting conversion based of the two addition theorems will exhibit the following behaviour:

$$0 + \texttt{SUC}(\texttt{SUC}(0 + 0)) \Rightarrow \vdash 0 + \texttt{SUC}(\texttt{SUC}(0 + 0)) = \texttt{SUC}(\texttt{SUC}(0 + 0)) \tag{3}$$

$$\texttt{SUC}(\texttt{SUC}\,0) + \texttt{SUC}\,0 \Rightarrow \vdash \texttt{SUC}(\texttt{SUC}\,0) + \texttt{SUC}\,0 = \texttt{SUC}(\texttt{SUC}\,0 + \texttt{SUC}\,0) \tag{4}$$

$$\texttt{SUC}(\texttt{SUC}(0 + 0)) \Rightarrow \texttt{HOL\_ERR}\ \langle\text{whatever}\rangle \tag{5}$$

The number of theorems used to create conversions in this manner can be quite large. For example, the HOL system employs a basic set of 31 theorems for general simplification. These are listed below to give the reader an idea of what kind of theorems to expect.

$$\forall x.(x = x) = T \tag{6}$$

$$\forall t.((T = t) = t) \wedge ((t = T) = t) \wedge ((F = t) = \neg t) \wedge ((t = F) = \neg t) \tag{7}$$

$$(\forall t.\neg\neg t = t) \wedge (\neg T = F) \wedge (\neg F = T) \tag{8}$$

$$\forall t.(T \wedge t = t) \wedge (t \wedge T = t) \wedge (F \wedge t = F) \wedge (t \wedge F = F) \wedge (t \wedge t = t) \tag{9}$$

$$\forall t.(T \vee t = T) \wedge (t \vee T = T) \wedge (F \vee t = t) \wedge (t \vee F = t) \wedge (t \vee t = t) \tag{10}$$

$$\forall t.(T \Rightarrow t = t) \wedge (t \Rightarrow T = T) \wedge (F \Rightarrow t = T) \wedge (t \Rightarrow t = T) \wedge (t \Rightarrow F = \neg t) \tag{11}$$

$$\forall t_1.\forall t_2.((T \rightarrow t_1 \mid t_2) = t_1) \wedge ((F \rightarrow t_1 \mid t_2) = t_2) \tag{12}$$

$$\forall t.(\forall x.t) = t \tag{13}$$

$$\forall t.(\exists x.t) = t \tag{14}$$

$$\forall t_2.\forall t_1.(\lambda x.t_1)t_2 = t_1 \tag{15}$$

$$\forall x.(\texttt{FST}\,x, \texttt{SND}\,x) = x \tag{16}$$

$$\forall x.\forall y.\,\texttt{FST}(x, y) = x \tag{17}$$

$$\forall x.\forall y.\,\texttt{SND}(x, y) = y \tag{18}$$

Note that more than one rule may apply: the term $T = F$ may be rewritten into either F or $\neg T$ by the first and fourth conjunct of (7). The "correct" behaviour for a rewriting conversion is unspecified in this situation and proofs shouldn't depend on which choice is made. As we will see later, however, they occasionally do.

The list suggests that theorems with lambda-abstractions are rare; in fact only (15) contains one. This is supported by the fact that the most common additions to the above list are definitions of functions like (1) and (2) above. Actually, the last three theorems above represent such an addition that is done during boot-strapping. We will assume that abstractions are indeed rare and consider ourselves lucky because matching abstractions with their freedom in naming could be expensive. Thus relatively little effort will go into optimizing matching of abstractions.

2.1 Direct Conversion Generation

The naïve way to turn a list of theorems into a rewriting conversion would be to simply try each theorem in turn:

```
fun naive_make_conv thl = FIRST_CONV (map REWR_CONV thl)
```

Recall that `REWR_CONV` is a primitive that will turn one equality theorem into a conversion, and that `FIRST_CONV` composes a list of conversions by trying them in turn until one succeeds.

Obviously conversions generated by the naïve method get slow as the number of theorems grows. Handling 31 theorems would be painfully slow. Therefore HOL uses a reduction method based on what we will call the *face* of a term:

Definition 1. By the face of a term we mean: (i) for a constant, the pair of symbol `CONST` and the constant's name; (ii) for a variable, the symbol `VAR`; (iii) for a combination, the symbol `COMB`; (iv) for an abstraction, the symbol `ABS`.

The face of a given term expresses what we can learn about the term from just looking at its top node. Computing a term's face is inexpensive. When a face is used for pattern matching we can use the HOL primitive `dest_term` for computing the face. It returns a little more than the face, but as we will see that is an advantage.

Observe now, that a term can only match a given pattern (the left-hand side of a theorem's conclusion) when they have the same face or when the face of the pattern is `VAR`; if the faces are different and if the pattern is not a variable then no instantiation of the pattern will equal the object term. This important property is what the term net module in HOL uses to make efficient conversions. Term nets are a variation of the discrimination net technique used in artificial intelligence, see [CRMM87, Chapters 8 and 11]. The following is a simplified description of how the matching in the term net module goes on.

In order to produce an efficient conversion from a list of theorems we start out by dividing the theorems into groups with respect to the patterns' faces. As variables match everything we will then add the theorems with `VAR`-face to the other groups. The conversion we are looking for will then examine the face of the object term and pick the right group. In case of the `COMB` and `ABS` groups matching can then recursively take place on subterms, but with the reduced list of theorems. At the leaves of this matching tree we are left with a (hopefully) small set of possible theorems with which we use the naïve method above.

Note that just because a theorem is present at some leaf there is no guarantee that it actually does match; the above method is only a quick-and-dirty method for reducing a problem size. There are several things that might go wrong with the actual matching: (1) If a pattern is not linear (i.e., some free variable occurs more than once) the assumption that a variable at any point matches anything does not hold. (2) The types may fail to match. (3) If the pattern contains an abstraction there is no keeping track of bound variables.

All the grouping of the theorems might at a first glance look very expensive, but since it is completely independent of the object term it can be done once

and for all. This leaves a quite efficient conversion that quickly selects a small number of possible theorems — typically none or just one — then tries those in turn.

3 Staging the Matching Process

Given the observations that (1) a lot of the work in the matching process only depends on the list of theorems, and (2) the list of theorems is known well in advance, and (3) the conversion will be used a lot of times, it seems natural to ask whether we can produce an optimized version for one particular list of theorems. More specifically, can we from the program (text, not closure),

$$\underline{\texttt{makeconv}} : \underline{\text{thmlist} \to \text{term} \to \text{thm}},$$

(We use the underlining to stress that we are talking about the program text of a function that when considered an SML-program has the non-underlined type.) and the value for its first parameter,

$$\text{thl} : \text{thmlist},$$

(the actual value or a suitable encoding of it) produce a new program (text again),

$$\underline{\texttt{conv}} : \underline{\text{term} \to \text{thm}},$$

where the theorems are "frozen" in the new program? The answer is "Yes," automatic program transformators of this kind do exist and are called partial evaluators [JGS93, BW93, BW94]. Unfortunately, for various technical reasons to be elaborated on in a separate paper, the resulting programs from using currently implemented partial evaluators are not satisfactorily efficient. See also [Wel94].

Having failed the automatic approach we turn to a related technique: hand-writing a program that given a list of theorems will produce the wanted conversion as a program text:

$$\texttt{convgen} : \text{thmlist} \to \underline{\text{term} \to \text{thm}}.$$

Note that the types of `makeconv` and `convgen` differ only in the underlining. Using terminology from the field of partial evaluation, `convgen` is called a generating extension of `makeconv`. We say that evaluation using `convgen` is *two-stage* because we have to invoke the SML-interpreter/compiler twice: once on `convgen` itself and once on the generated conversion.

Writing `convgen` directly is not nearly as elegant as the automatic approach, if for no other reason then because we could produce something functionally equivalent to `convgen` automatically from `makeconv` by evaluating the expression (`mix` `mix` `makeconv`) where `mix` is the partial evaluator. (That was indeed the application of `mix` to its own program text.) This variant would unfortunately produce conversions suffering from the same lack of efficiency as observed above.

3.1 Staging Versus Non-Staging

Since the two-stage method is somewhat more complicated than the one-step there must be some benefits to make it worth while. And in fact there are:

Speed. The two-stage conversions are about twice as fast, see the timings below. Several factors contribute to the increase in speed: (1) The actual compiled code in the one-stage conversions is something that uses the object term to work through a term net, i.e., it sort-of interprets the term net; in contrast the same thing is accomplished by control flow in the two-stage version. (2) When the matching is over, the one-stage conversion always uses `REWR_CONV` which redoes the matching while the two-stage version usually uses `SPEC` which is [implemented by] a relatively cheap beta conversion. (3) Evaluation involves considerably more closures in the one-stage version.

Memory. Conversions produced by the two-stage method ought to produce less garbage on the heap. "Ought" here means that I have been unable to verify that experimentally, possibly because I have been unable to get sufficiently stable readings of garbage collection times.

There are unfortunately also some drawbacks:

Time to construct. Since the compiler has to be re-invoked for the two-stage method, construction takes longer. Obviously this means that the two-stage method should only be used when the conversions are expected to be used often.

Extensibility. Conversions produced by the one-step method (actually the term nets behind) are easily and efficiently extended with extra theorems. This is what takes place when, e.g., `REWRITE_TAC` is used. This is not directly possible with the two-step approach.

Code size. The code size of a two-step conversion is approximately five lines per theorem plus a minor overhead. I fail to see how this could become a problem in real-life use, even if hundreds of theorems were used.

Note, that all these advantages and disadvantages are special cases of the general problematics between using interpreters and compilers.

3.2 Generating Conversions

This section describes how to construct a program that generates the conversions we are looking for, i.e., how to construct a conversion generator. The method described here has actually been implemented in the form of The MW Conversion Generator, currently version 0,21. It is available from the author.

The job is, given a list of patterns, to write a function that matches a term against the patterns. Since we will be dealing with term combinations which have more than one (namely two) subterm each which must all match, we generalize the situation to matching a list of terms against a list of pattern lists. The terms here cannot be real terms (i.e., of type `Term`) since the terms are not available

at the time we are generating the conversion; instead they are names of SML variables, e.g. `"tm122"`, of variables that in the resulting conversion will hold the terms in question.

At a given point in the matching we therefore have a list of options each consisting of a list of termvar-pattern pairs, the theorem we are matching against, and some extra information to be discussed later. For ease of presentation only the termvar-pattern pairs are considered in the following and the list of options therefore has SML–type `(string * term) list list`. The code to be generated for a given list of options can now be calculated as in the following where `typewriter` style is used for the generated code and plain roman style is used for calculations performed when generating.

```
code([ ]) = raise bad
code([ ] :: _) = (* Match found, see below. *)
code(opts) =
  let
    (var, tm) = hd (hd opts)
    (var1, var2) = (var ^ "1", var ^ "2")
    rest f = map tl (filter (hasface f) opts)
  in
    (case dest_term var of
     CONST c1 => code (rest (CONST c1))
    | ...
    | CONST cn => code (rest (CONST cn))
    | ABS {Body=var1,...} => code ((var1, body tm) :: rest ABS)
    | COMB {Rator=var1,Rand=var2} =>
        code ((var1, rator tm) :: (var2, rand tm) :: rest COMB)
    | _ => raise bad
    ) handle _ => code (rest VAR)
  end
```

The appearance of an expression like "var" in "`dest_term` var" is to be understood as that the result of evaluating that expression while generating the conversion — the result will be a string — is to be inserted in the generated program at the place where the expression occurs.

In the branches of the generated case expression only those options with matching face need to be considered. If there are no matching options, the corresponding branch can be eliminated. The exception handling is for variables which at this point are assumed to match anything that the other patterns did not match. If there are no patterns with variable face, then the exception handling construct can be eliminated.

Since the patterns are finite the above function will produce a finite tree of nested case expressions. At the leaves we are left with a typically small number of theorems and we can simply return `(REWR_CONV thm1) ORELSEC ... ORELSEC (REWR_CONV thmn)` applied to the object term. This can be seen as an unfolded version of the naïve method.

There are a few optimizations apart from the obvious above that are needed in order to get really good performance. First of all, using `REWR_CONV` means that the matching is redone, so we will want to avoid that. This definition tells of when that is possible:

Definition 2. A theorem is said to be `SPEC`-safe when (1) it has no free variables, i.e., all variables have been universally quantified over; (2) the pattern is linear, i.e., all variables occur exactly once; (3) the pattern contains no abstractions; and (4) the variables are all monomorphic.[2]

When a theorem is `SPEC`-safe it means that all the assumptions we have made, i.e., that variables match everything and that the names of λ-bound variables need not be checked, are indeed valid. We can therefore *guarantee* that the object term matches the pattern. Furthermore, since we have already done the matching of variables against terms, we can use `(SPEC` $\mathtt{tm}_a$ `(...(SPEC` $\mathtt{tm}_z$ `thm)...)` to get the resulting theorem. For this to work, the conversion generator must keep track of what variables are matched against.

The second optimization worth making is that when all theorems handled by the case expression are `SPEC`-safe then it is often possible to replace the underscore branch by the code for the variable case. This eliminates the expensive exception handling.

One final optimization worth making is to install a variant of `dest_term` in the term module. The current version is quite expensive for abstractions because of the deBruijn representation used internally. A version that doesn't do variable renaming is just as good for us since that does not change the faces of subterms. The term net module uses exactly the same trick.

As an example, consider again the theorems (1) and (2) for addition on Peano numbers. The complete conversion produced for these two theorems is shown in appendix A. The essential matching part is this piece of code

```
val conv:conv = fn tm =>
 (case dest_term tm of
   COMB {Rator=tm1,Rand=tm2} =>
    (case dest_term tm1 of
     COMB {Rator=tm11,Rand=tm12} =>
      (case dest_term tm11 of
        CONST {Name="+",...} =>
         (case dest_term tm12 of
          CONST {Name="0",...} => SPEC tm2 thm_1
         | COMB {Rator=tm121,Rand=tm122} =>
           (case dest_term tm121 of
             CONST {Name="SUC",...} => SPEC tm122 (SPEC tm2 thm_2_1)
            | _ => raise bad)
         | _ => raise bad)
       | _ => raise bad)
    | _ => raise bad)
 | _ => raise bad)
```

[2] This can be relaxed to either monomorphic or just a type variable with the restriction that type variables must then be different. This change would require that `INST_TYPE` be used to instantiate the types of the theorem. This is easy, but has not been implemented.

This is a very efficient combined matching and destruction of the object term. Furthermore, when matching succeeds, the beta-conversion-like `SPEC` is used to produce the wanted equality theorem. Except for the use of `dest_term` instead of the older term interface `is_...` this code is what a user would write by hand.

3.3 Integration with Hol

To test the conversions in practice we wish to replace the standard 31-theorem rewriting conversion by one produced by the two-step method. It turns out that it is quite easy to integrate such a conversion with HOL: we have to produce two conversions, one without and one with the pair theorems (16), (17), and (18). The former is only needed during boot-strapping until pairs are defined. We need to add 40 lines and change another 20 lines in the `Rewrite` functor and signature, change 5 lines in the boot-strapping scripts, and optionally add 10 lines in the `Term` functor and signature (to install a faster `dest_term` function). It seems reasonable to characterize this as minor changes.

There is one tricky point to the integration. As mentioned above the one-step conversions are extensible and the two-step conversions are not. We can solve this by noting that conversions can be composed sequentially with `ORELSEC` and that we can use the old machinery to handle the extra *ad hoc theorems.* In the presence of n ad hoc theorems, the following method was used for tests. $n = 0$: just the two-stage conversion; $n = 1, \ldots, 4$: two-stage followed by naïve conversion; $n = 5, \ldots, 10$: two-stage followed by term net conversion; $n > 10$: revert completely to term net conversion. Exactly when to use which method is the result of hunches and a little testing, not deep theoretical insight or elaborate benchmarking.

Testing shows that some of the library proofs depend on the conversions to choose a certain one of several possible theorems to rewrite by. Consider the object term $(x_1 + x_2) * (x_3 + x_4)$ with the theorems $\vdash x * (y + z) = x * y + x * z$ and $\vdash (x + y) * z = x * z + y * z$ for integer or real numbers. The term net based conversions will always rewrite the term with respect to the latter theorem regardless of whether it is mentioned before or after the former theorem in the list of theorems. The proof of, e.g., theorem `POW_PLUS1` from the real number library depends on this. I consider such a proof just as broken as one that depends on the order of assumptions. Therefore it is not a problem that the two-step method chooses a different theorem. Fixing the proof is trivial.

3.4 Some Timing Results

For reference, the timings below have been run on a Sun 4/75 (also know as a Sparc-Station 2) with 64MB Ram running little but the SunOS kernel and HOL-90. In particular this means that no swapping took place during the tests. The timings were done with SML–NJ version 0.93's built-in timers.

There seems to be no standard reference proof used for timings, so for the purpose of this article a part of the HOL library will be used for timing. More

specifically, the code in `library/real/theories/src/real.sml`, a part of HOL-90.7's real number library, was used. This is approximately a two minute proof.

To test the stand-alone speed of a generated conversion the 31 theorems from the basic HOL conversion were used. From the test proof, the first 10000 terms to which the basic conversion is applied were used. Of these only 284 can actually be rewritten. These rewritable terms were duplicated to form another 10000 terms.

Terms	One-stage	Two-stage	Reduction
10000 first	5.6	2.5	55%
10000 rewritable	23.4	12.9	45%

Since obviously the proof process does other things apart from rewriting, the entire proof (i.e., `real.sml`) was also timed. This gives an estimate of the overall reduction in the running times for proofs.

Proof	One-stage	Two-stage	Reduction
`real.sml`	134.8	120.0	11%

We consider a speed-up of over ten percept to be satisfactorily since the only user intervention needed was fixing the proof of `POW_PLUS1` as discussed previously.

3.5 Cooperation

When the internal rewriting done by the HOL system is changed to use a two-stage conversion, the ad hoc theorems used for rewriting, i.e., those passed to for example `REWRITE_TAC` become relatively expensive because they have to be handled by the old method.

A cooperating user will therefore have to reduce the number of ad hoc theorems used for rewriting in order to get maximal benefit. Since rewritings are a very convenient tool during proof this is somewhat awkward. However, there is one class of rewriting that comes up repeatedly, namely rewriting with respect to some function's definition. The theorems defining addition on Peano numbers, (1) and (2), serve again as an example. In a large theory it would be beneficial to create a two-step conversion for such a definition and to use `CONV_TAC` instead of using `REWRITE_TAC` with the defining equations as ad hoc theorems.

This method is not feasible for existing theories since it would require major changes in the proofs. It might, however, be very worth while for new theories.

4 Conclusions and Future Work

It has been shown in this article that evaluation of a function in some situations can be staged with significant speed-up as result. This observation has led to the construction of a fully automatic tool that generates conversions from equality-theorems. Since such conversions are used by the HOL core a general speed-up of HOL of about ten percent was possible.

The principle of staging can be used in other situations, for example the library `generator` seems promising with its interpretive style and extended running times. (This library helps with the embedding of programming languages' syntax and semantics in HOL, see [RK94].) It would be interesting to see this hypothesis proven in practice, if true.

The present work could be improved by a closer study of the garbage collection aspects involved. This is pending, awaiting the availability of a reliable method of measurement.

The work described in this paper has stayed away from combining the top-level rewriting with depth conversions like `TOP_DEPTH_CONV`. Doing so would mean giving up some of the modularity, but the potential gain is to avoid many fruitless attempted rewritings since a term resulting from one rewriting have a well-known structure.

Acknowledgements

The author wishes to thank Thomas F. Melham for suggesting the conversion generation as a target for partial evaluation techniques and for help with an earlier version of the conversion generator. Further thanks goes to Jesper Jørgensen for valuable discussions of some of the partial evaluation aspects involved.

References

[Bou94] Richard J. Boulton. *Efficiency in a Fully-Expansive Theorem Prover*. PhD thesis, University of Cambridge Computer Laboratory, New Museums Site, Pembroke Street, Cambridge CB2 3QG, United Kingdom, May 1994.

[BW93] Lars Birkedal and Morten Welinder. Partial evaluation of Standard ML. Technical Report 93/22, DIKU, Department of Computer Science, University of Copenhagen, October 1993.

[BW94] Lars Birkedal and Morten Welinder. Handwriting program generator generators. In Manuel Hermenegildo and Jaan Penjam, editors, *Programming Language Implementation and Logic Programming. 6th International Symposium, PLILP '94, Madrid, Spain*, volume 844 of *Lecture Notes in Computer Science*, pages 198–214. Springer Verlag, September 1994.

[CRMM87] Eugene Charniak, Christopher K. Riebeck, Drew V. McDermott, and James R. Meehan. *Artificial intelligence programming*. Lawrence Erlbaum Associates, 2nd edition, 1987.

[GM93] Michael J. C. Gordon and Thomas F. Melham. *Introduction to HOL*. Cambridge University Press, 1993.

[JGS93] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.

[Pau83] Lawrence Paulson. A higher-order implementation of rewriting. *Science of Computer Programming*, 3:119–149, 1983.

[RK94] Ralf Reetz and Th. Kropf. Simplifying deep embedding: A formalised code generator. In Thomas F. Melham and Juanito Camilleri, editors, *Proc. of the 7th International Workshop on Higher Order Logic Theorem Proving*

and its Applications, volume 859 of *Lecture Notes in Computer Science*, pages 378–390. Springer Verlag, September 1994.

[Sli95] Konrad Slind. Hol-90.7, 1995. An implementation of the Higher-Order Logic Theorem Prover in Srandard ML.

[Wel94] Morten Welinder. Towards efficient conversions by use of partial evaluation. In Thomas F. Melham and Juanito Camilleri, editors, *Supplementary Proceedings of the 7th International Workshop on Higher Order Logic Theorem Proving and its Applications*, September 1994.

A Peano Addition Conversion

```
(* ------------------------------------------------------------------- *)
(* The following code is machine generated.  For this reason it is not *)
(* wise to edit it by hand.  The main function is a conversion that    *)
(* rewrites left-to-right with respect to one of the theorems below.   *)
(*                                                                     *)
(* The code was generated by MW Conversion Generator version 0,21.     *)

(* Make sure the conversion is compiled properly: *)
val save_System_Control_interp = !System.Control.interp;
val _ = System.Control.interp := false;

local
 open Term Thm Conv Drule Dsyntax

 (* This or another HOL_ERR exception will be raised if none of
    the theorems matches: *)
 val bad = HOL_ERR {message = "No matching theorem",
                    origin_function = "conv",
                    origin_structure = "top level"};

 (* Utility functions: *)
 (* We check that the theorems are the right ones: *)
 fun check thm tm =
   if concl thm=tm then
     thm
   else
     raise HOL_ERR {message = "Environment changed",
                    origin_function = "",
                    origin_structure = "top level"};
 val thm_1 = check (ADD1) (--‘!n. 0 + n = n‘--)
 val thm_2 = check (ADD2) (--‘!m n. SUC m + n = SUC (m + n)‘--)
in
 val conv:conv = fn tm =>
  (case dest_term’ tm of
     COMB {Rator=tm1,Rand=tm2} =>
     (case dest_term’ tm1 of
        COMB {Rator=tm11,Rand=tm12} =>
        (case dest_term’ tm11 of
           CONST {Name="+",...} =>
           (case dest_term’ tm12 of
              CONST {Name="0",...} => SPEC tm2 thm_1
            | COMB {Rator=tm121,Rand=tm122} =>
              (case dest_term’ tm121 of
                 CONST {Name="SUC",...} => SPEC tm2 (SPEC tm122 thm_2)
               | _ => raise bad)
            | _ => raise bad)
         | _ => raise bad)
      | _ => raise bad)
   | _ => raise bad)
end;
val _ = System.Control.interp := save_System_Control_interp;

(* Machine generated code ends here.                                   *)
(* ------------------------------------------------------------------- *)
```