Towards Efficient Conversions by use of Partial Evaluation

Morten Welinder
DIKU, Department of Computer Science
University of Copenhagen
DK–2100 Copenhagen Ø, Denmark
Email: `terra@diku.dk`

August 1994

Abstract

Naïvely written conversions in HOL tend to be slow. There are several reasons for this: 1) The theorems by which the rewriting is performed are present as syntax that needs to be inspected by some matcher as opposed to ML-code that decides whether some rule is applicable. 2) Rewriting is attempted on terms that it is known *a priory* that they will not succeed on, e.g., because the terms might just have been derived by a known rewriting rule. 3) Rewriting is attempted several times on the same terms.

In this article it is demonstrated that the technique known as *partial evaluation* can be used to solve problem 1. It will be argued that stronger partial evaluators than currently available might help solving problem 2. A solution to problem 3 seems to require a deeper understanding of the rewritings in question.

Some experiments with rewriting systems describing primitive recursive functions have been made. The results suggest that partial evaluation can lead to conversions almost three times as fast as naïvely written equivalents but currently not as fast as hand-written conversions. Solving problem 2 above would lead to significant further speed-up.

1 Introduction

A conversion in HOL is a function that takes as its argument a term and produces a theorem, presumably stating that the source term equals some other term. Conversions span from the trivial `REFL` simply expressing that any term equals itself to functions that repeatedly traverses a term doing rewritings here and there.

In many applications of HOL it seems natural to use conversions to simplify terms. Unfortunately, conversions tend to be slow because it is difficult to specify exactly where one would like them applied and because they are constructed by very general tools. This slowness is the reason why conversions are not always used where one would like to.

This article describes an *automatic* way of deriving relatively efficient conversions from inefficient ones. This is done by using *partial evaluation,* also known as program specialization.

More efficient conversions can be created by hand-writing the code needed, but this is a time-consuming process and requires some expertise. The methods here, on the other hand, are fully automatic once the system has been set up.

2 Conversions from Equality Theorems

A significant subset of all conversions is the *rewriting conversions.* Such a conversion is based on a list of 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, then substituting the corresponding right-hand side. Variations (for this type of conversions as well as others) include repeating the process and subjecting subterms to the same treatment. HOL provides primitives like `REDEPTH_CONV` for deriving these variations.

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

$$\text{(ADD1)} \qquad \vdash \forall n : 0 + n = n$$

$$\text{(ADD2)} \qquad \vdash \forall m \forall n : \mathrm{SUC}\, m + n = \mathrm{SUC}(m + n)$$

Addition is a primitive recursive function, so any left-to-right rewriting of any term will terminate eventually. For example, by rules (ADD2), (ADD2), and (ADD1) we have

$$\begin{aligned} \mathrm{SUC}(\mathrm{SUC}\,0) + \mathrm{SUC}\,0 &= \mathrm{SUC}(\mathrm{SUC}\,0 + \mathrm{SUC}\,0) \\ &= \mathrm{SUC}(\mathrm{SUC}(0 + \mathrm{SUC}\,0)) \\ &= \mathrm{SUC}(\mathrm{SUC}(\mathrm{SUC}\,0)) \end{aligned}$$

and neither of the rules can be applied to the resulting term. The conversions that we consider in this article will perform exhaustive rewriting and thus in this case produce the theorem

$$\text{(THM1)} \qquad \vdash \mathrm{SUC}(\mathrm{SUC}\,0) + \mathrm{SUC}\,0 = \mathrm{SUC}(\mathrm{SUC}(\mathrm{SUC}\,0))$$

when applied to $\mathrm{SUC}(\mathrm{SUC}\,0) + \mathrm{SUC}\,0$. In HOL,[1] primitives exist that allow us to rewrite with respect to a list of theorems. The following SML-function will do so.

[1] Here and in the following, the term HOL is used meaning the SML-version HOL-90.

```
(* Naive rewriter *)
fun rewriter thms =
  GEN_REWRITE_CONV REDEPTH_CONV (ref Net.empty_net) thms;

val rewrite_plus = rewriter [ADD1,ADD2];
```

This may or may not be the smartest way to do the rewriting, but that is beside the point here. The conversions presented later will unless noted rewrite the same way, and the arguments about optimization apply regardless of strategy.

Internally `GEN_REWRITE_CONV` works with term nets, a data structure for fast tree matching. The theorems are compiled into these nets at the time `rewrite_plus` is constructed. For efficiency reasons, `GEN_REWRITE_CONV` additionally (no pun intended) allows a precompiled term net to be used, but this feature is not usable in our case.

We use this naïve rewriter for reference because it is very easy for the user to construct.

3 Partial Evaluation

Suppose now, that we expect to rewrite a lot of different terms with respect to the same set of theorems. In this case it might often be beneficial to work hard in order to get an efficient conversion. Partial evaluation [JGS93,BW93] is an automatic procedure for doing this.

In order to describe the ideas of partial evaluation we need some notation. For a program p written in some language L we let $[\![p]\!]$ be the meaning of p as an input-output function as specified by the semantics of L.

For a two-argument program like `rewriter` above we will now assume that the first argument (the list of theorems) is known while the second (the term to be rewritten) is not. This is the situation we have when we want to apply the same conversion to a lot of not-yet-known terms. The known argument we will call s and describe as static; the second we will call d and describe as dynamic.

A partial evaluator, traditionally called *mix*, is now a program that satisfies the following condition. For notational simplicity we will assume that both *mix* and p are written in the language L.

$$[\![p]\!]\ s\ d = [\![[\![mix]\!]\ p\ s]\!]\ d \tag{1}$$

for all static and dynamic values. In words, *mix* symbolically performs the partial application[2] of p to its first argument s.

Any program, *mix*, satisfying the above equation is called a partial evaluator and is a realization of Kleene's S-m-n-theorem as known in recursive function theory, but it is easy to see that in most languages a trivial partial evaluator can be constructed. Since we are aiming at getting faster programs we shall only be concerned with non-trivial partial evaluators, i.e., those that actively performs program transformation in order to improve the program.

[2]This informal description becomes difficult to uphold in the more general case when considering *partially static structures*, for example a tree of known structure but unknown elements.

The program transformations done by partial evaluators typically include 1) constant folding, 2) function specialization which is the duplication of a function definition when two calls to the function are encountered and these calls have different static parameters, and 3) unfolding of function definitions. Some partial evaluators do more, some do less. It should be noted that 'constant folding' includes choosing the right branch of a conditional when the condition is sufficiently known, and that points 2) and 3) make it inter-procedural.

We expect the *residual program* $p_s \equiv [\![mix]\!]\ p\ s$ to be faster than the *general program p* because some of the operations may already have been performed. Whether this expectation is fulfilled of course depends on the partial evaluator used.

Just as it is the case with all other optimizations techniques there is a trade-off between the time spent optimizing and the time saved. The assumption here shall be that the residual program is run so often that any optimization is worth the while no matter what it costs.[3] A more careful study of the trade-offs is beyond the scope of this article.

4 Improving the Matching

At present, partial evaluators give the best results when applied to cleanly written programs. The meaning of "clean" is not very well defined and depends on the partial evaluator used. No partial evaluator for a functional language has been able to handle assignments and exceptions with good results, so unfortunately `GEN_REWRITE_CONV` used to define the naïve rewriter in the previous section cannot be subjected to partial evaluation with good result. For this reason, a "clean" rewriter was programmed and appears in Appendix A. We will call this one the *general rewriter* using partial evaluation terminology even though it is not quite as general as the naïve rewriter becuase, for simplicity, it does not handle polymorphism.

This rewriter is used a little differently than the previous one. To make the addition conversion, one would say

```
val rewrite_plus_once:conv =
  rewrite [ADD1,ADD2]
          [fn [a,b] => a = --`0`--,
           fn [a,b] => is_comb a andalso rator a = --`SUC`--]
          [fn [a,b] => [b],
           fn [a,b] => [rand a,b]];
val rewrite_plus = REDEPTH_CONV rewrite_plus_once;
```

The first of the two extra lists needed, is a list of predicates deciding whether the corresponding theorem can be used. The second list is a list of functions saying what values the ∀-bound variables in the theorems need to be specialized with respect to. It is important to realize that both of these lists could have been calculated from the first with some extra

[3]The partial evaluation reported in this article took only a small fraction of a second. This is typical for partial evaluation in practice for source programs of the given size. However, since a non-trivial partial evaluator interprets parts of the source program there is usually no theoretical upper limit on the time it could take.

code. This has been avoided to simplify matters. Note incidently, that given these extra parameters, the code is as simple as possible and just tests the theorems one by one.

For technical reasons experiments were carried out using the Similix partial evaluator [BD90,Bon93], a partial evaluator for Scheme [CR91]. The rewriter was thus translated (by hand) into Scheme, partially evaluated, and the residual program then translated back (again by hand). In principle, a partial evaluator for SML (e.g., ML-MIX [BW93]) could just as well have been used, but they are still prototypes. The Scheme versions of the programs will not be shown here.

Specializing the general rewriter with respect to the addition theorems yields the residual program shown in Appendix B. Inspection shows that the residual rewriter is made partly from the general rewriter and partly from the static data. The lists that held the theorems and the utility functions have all gone (but the *contents* of the lists is still present in some way, of course).

In order to test the efficiency of the residual function we use the following function

```
local
  fun mk_add x y =
    mk_comb {Rand = y, Rator = mk_comb {Rand = x, Rator = --`$+`--}};
in
  fun fib 0 = --`SUC 0`--
    | fib 1 = --`SUC 0`--
    | fib n = mk_add (fib (n-2)) (fib (n-1));
end;
```

to generate `fib 10`, a quite large term involving lots of applications of `+`. (The tenth Fibonacci number is 89 by this definition.) The rewriting of this example term was done with four different rewriters giving the following results.

Program	Source	Time	Thms
Naïve rewriter	Section 2	1.39 s	934
General rewriter	Appendix A	1.32 s	1169
Residual rewriter	Appendix B	0.44 s	1169
Hand-written rewriter	Appendix C	0.38 s	1169
Hand-written rewriter unfolded	Appendix D	0.34 s	1169

For reference, these benchmarks were performed on a HP 9000 model 735 with enough memory to avoid swapping and no other users. The times shown are averages over 100 runs. The theorem counts shown are the total number of theorems proved per run. This means that nine theorems proven in the construction of the naïve `rewrite_plus` have not been counted — this is reasonable because that is done once and for all.

Since the general rewriter was written rather simple-mindedly it is not very surprising that the residual rewriter is faster. That it is as much as three times faster, however, is encouraging — especially considering that the time measured includes the time spent by `REDEPTH_CONV` for traversing the term and proving combination theorems.

An analysis shows that it is not only the elimination of the lists that helps, but also that the compiler can generate much better code for calling known functions than for parameter-passed functions.[4]

It is also interesting to note that the residual rewriter is a bit more than three times as fast as the naïve rewriter. This is the speed-up that a user can get *without any special knowledge about conversions or partial evaluation.* (Of course, since a Scheme-based system was used for experiments there would still be the translation from Scheme back into Standard ML, but that is next to trivial.)

The hand-written version serves as a reference showing the best-possible case constrained by the use of `REDEPTH_CONV`. It should be noted that the hand-written version of `rewrite_plus_once` is *not* functionally equivalent to any of the others, but that the difference (which is in exception raising) is eliminated by `REDEPTH_CONV`. The residual rewriter is only 13% slower than the hand-written one — we consider this to be satisfactory.

The hand-written rewriter was then together with the definition of `REDEPTH_CONV` (from the HOL-90 source code) subjected to further partial evaluation (*in casu* just unfolding). The result of this appears in Appendix D. Only 10% improvement was achieved, so the fact that `REDEPTH_CONV` is parameterized over the conversion to use does not add significantly to its execution time.

5 Avoiding Hopeless Matching

Even though the fast matching discussed in the previous section certainly does improve the performance of the conversions there is still a large overhead to attack.

Consider a term $\mathrm{SUC}\, x + y$ which is rewritten into $\mathrm{SUC}(x + y)$. The latter term is an application whose operator is the constant SUC.

> *There is no hope that rewriting can ever succeed on this operator, no matter what x and y are. This conclusion follows from the addition theorems alone.*

Why is it then, that the partial evaluator does not discover this property? It turns out that there are several reasons, partly concerned with HOL and the HOL community, and partly concerned with the state of partial evaluation.

First of all, to discover properties about the output from theorem handling functions, the partial evaluator would have to have access to the source of the HOL-axioms. Here, 'theorem handling functions' means the primitive axioms, or more specifically (c.f. Appendix D) `SPEC`, `TRANS`, and `concl`.

Giving the partial evaluator access to the source code of these primitive might lead to residual programs containing explicit calls to `mk_thm`,[5] thus reducing safety in case there was a bug in the partial evaluator. It has been pointed out to me, that the average member of the HOL-community would look at such residual programs with severe distrust,

[4] In HOL terms are handled in a module thus hiding implementation details. From a program development point of view this is a good thing, but it means that, e.g., `rator` is treated as an unknown entity when compiling the residual rewriter. It is conjectured that a significant further speed-up could be achieved by opening the term structure further.

[5] Recall that `mk_thm` is the ill-advertised primitives that 'type casts' sequents into theorems, a rather unsafe approach to theorem proving.

even though he/she trusts other optimizing tools, e.g., the compiler. For the sake of the argument, consider access to the source code of the axioms granted.

We will now consider the source code of the residual rewriter in Appendix D to examine what might have been improved. The double call to SPEC

```
SPEC tm1 (SPEC (rand tm3) ADD2)
```

would give us a result of the form

$$(\textsc{Thm2}) \qquad \vdash \mathrm{SUC}\langle \mathit{rand\ tm3}\rangle + \langle \mathit{tm1}\rangle = \mathrm{SUC}(\langle \mathit{rand\ tm3}\rangle + \langle \mathit{tm1}\rangle)$$

This is an entity where parts of it are known and others are not. Such an entity is called *partially static.* In this case, there is enough static information available to determine that the following call to REDEPTH_QCONV has the right-hand side $\mathrm{SUC}(\langle \mathit{rand\ tm3}\rangle + \langle \mathit{tm1}\rangle)$.

A sufficiently smart partial evaluator would use this information to conclude that this would lead to (among other things) a recursive call to REDEPTH_QCONV with the operator, SUC, the result of which would be the UNCHANGED exception. This would completely eliminate fruitless searches in the operator branches.

Unfortunately, such smart partial evaluators do not exist to the best of my knowledge. Practical partial evaluators today are *off-line* meaning that they in advance clasifies every operation in the object program as depending solely on static data or not. This greatly improves termination properties for the partial evaluator as well as its speed. On the other hand, in situations like this, it hinders using the value of ADD2 to determine the structure of the argument to the next recursive call.

6 Related Work

Partial evaluation has been discussed extensively in the literature, but only recently with some degree of overview in the text book [JGS93].

HOL is a highly modular program where the implementation of terms, types, theorems, and so on are hidden from other parts of HOL. This is good when considering program development (and security in the case of theorems), but since modules are intended to be compiled separately a significant overhead is introduced. By working with the intermediate code produced by SML/NJ (some untyped λ-calculus look-alike) a group at The University of Aarhus and Carnegie-Mellon University has produced promising results of inter-modular partial evaluation [MHD94].

7 Conclusion

This article has shown that it is possible to use partial evaluation to optimize conversions. The optimized conversions turned out to be almost three times as fast as the un-optimized ones but not quite as fast as hand-written conversions.

It was furthermore shown, that in a future where more mature partial evaluators exist, it may be possible to obtain even faster conversions than now.

In principle, partial evaluation is an automatic process, so optimized versions of conversions could be generated without understanding the principles of partial evaluation. The only time where such knowledge is needed is in writing the rewriter subjected to partial evaluation. This is a one-time act.

Further research in the area might include trying to have hopeless search branches cut off.

References

[BD90] Anders Bondorf and Olivier Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. Technical Report 90/4, DIKU, 1990. Revised version in [Bon91].

[Bon91] Anders Bondorf. Automatic autoprojection of higher order recursive equations. *Science of Computer Programming*, 17:3–34, March 1991.

[Bon93] Anders Bondorf. Similix 5.0 manual. Distributed with the Similix system, 1993.

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

[CR91] William Clinger and Jonathan Rees. Revised^4 report on the algorithmic language Scheme, November 1991.

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

[MHD94] Karoline Malmkjær, Nevin Heintze, and Oliver Danvy. ML partial evaluation using set-based analysis. In *Record of the 1994 ACM SIGPLAN Workshop on ML and its Applications, Orlando, Florida*, pages 112–119. INRIA, June 1994. Rapport de Recherche no. 2265.

A General Rewriter

The following is a general rewriter, that rewrites a term with respect to a list of theorems defining a primitive recursive function. The rewriting is only attempted once and only at the top level of the term.

```
val bad = HOL_ERR {message = "No matching theorem",
                   origin_function = "rewrite",
                   origin_structure = "top level"};

(* Given ARITY, FUNCTION, and TERM.  Is TERM a full application of
   FUNCTION having ARITY.  *)
fun is_app 0 func tm = (tm = func)
  | is_app n func tm = is_comb tm andalso is_app (n-1) func (rator tm);

(* Provided the above is satisfied, extract the arguments.  *)
fun get_args 0 tm acc = acc
  | get_args n tm acc = get_args (n-1) (rator tm) (rand tm::acc);

(* Find and specialize theorem by trying each one in turn. *)
fun spec_thm args (thm::thms) (disc::discs) (extr::extrs) =
  if disc args then
    (SPECL (extr args) thm)
  else
    spec_thm args thms discs extrs
  | spec_thm _ _ _ _ = raise bad;

fun rewrite_prop func arity thms discs extrs tm =
  if is_app arity func tm then
    spec_thm (get_args arity tm []) thms discs extrs
  else
    raise bad;

fun rewrite thms discs extrs tm =
  let
    val lhs1 = lhs (#2 (strip_forall (concl (hd thms))))
    fun get_func_arity tm n =
      if is_comb tm then
        get_func_arity (rator tm) (n+1)
      else
        (tm,n)
    val (func,arity) = get_func_arity lhs1 0
  in
    rewrite_prop func arity thms discs extrs tm
  end;
```

B Residual Rewriter

The general rewriter in Appendix A partially evaluated with respect to the two addition theorems. The reason that operator and operand are calculated more than once is that so did the general rewriter. The partial evaluator has not been shown the source code for neither `rand` nor `rator` and therefore has not been able to reorder, throw away, or duplicate calls to these functions.

```
val plusterm = --‘$+‘--;
val zeroterm = --‘0‘--;
val sucterm = --‘SUC‘--;

val bad = HOL_ERR {message = "No matching theorem",
                   origin_function = "rewrite",
                   origin_structure = "top level"};

fun rewrite_plus_once tm0 =
  if is_comb tm0 andalso
    let
      val tm1 = rator tm0
    in
      is_comb tm1 andalso plusterm = rator tm1
    end then
    let
      val tm3 = rand tm0
      val tm4 = rator tm0
      val tm5 = rand tm4
      val _ = rator tm4
    in
      if zeroterm = tm5 then
        SPEC tm3 ADD1
      else
        if is_comb tm5 andalso sucterm = rator tm5 then
          SPEC tm3 (SPEC (rand tm5) ADD2)
        else
          raise bad
    end
  else
    raise bad;

  val rewrite_plus = REDEPTH_CONV rewrite_plus_once;
```

C Hand-written Rewriter using REDEPTH_CONV

This hand-written rewriter is the fastest that I could come up with given that it should use REDEPTH_CONV for traversing the source term. It uses the fact that REDEPTH_CONV will trap any HOL_ERR, in particular any raised by dest_comb.

```
val plusterm = --`$+`--;
val zeroterm = --`0`--;
val sucterm = --`SUC`--;

val bad = HOL_ERR {message = "No matching theorem",
                   origin_function = "rewrite_prop",
                   origin_structure = "top level"};

fun rewrite_plus_once tm0 =
  let
    val {Rand = tm1, Rator = tm2} = dest_comb tm0
    val {Rand = tm3, Rator = tm4} = dest_comb tm2
  in
    if tm4 = plusterm then
      if zeroterm = tm3 then
        SPEC tm1 ADD1
      else
        if sucterm = rator tm3 then
          SPEC tm1 (SPEC (rand tm3) ADD2)
        else
          raise bad
    else
      raise bad
  end;

val rewrite_plus = REDEPTH_CONV rewrite_plus_once;
```

D Hand-written Rewriter w/Unfolded REDEPTH_CONV

The appendix shows the hand-written rewriter where the definiton of REDEPTH_CONV has been unfolded. For reference, this definition is in .../src/1/conv.sml in the HOL-90 distribution. Note that the REDEPTH_QCONV appearing here has been specialized to conversion with respect to the two addition theorems.

```
exception UNCHANGED;

val plusterm = --`$+`--;
val zeroterm = --`0`--;
val sucterm = --`SUC`--;
val bad = HOL_ERR {message = "No matching theorem",
                   origin_function = "rewrite_prop",
                   origin_structure = "top level"};


fun REDEPTH_QCONV tm =
  let
    val th1 =
      if is_comb tm then
        let
          val {Rator,Rand} = dest_comb tm
        in
          let
            val th = REDEPTH_QCONV Rator
          in
            MK_COMB (th, REDEPTH_QCONV Rand)
            handle UNCHANGED => AP_THM th Rand
          end
          handle UNCHANGED => AP_TERM Rator (REDEPTH_QCONV Rand)
        end
      else
        if is_abs tm then
          let
            val {Bvar,Body} = dest_abs tm
            val bodyth = REDEPTH_QCONV Body
          in
            MK_ABS (GEN Bvar bodyth)
          end
        else
          raise UNCHANGED;
    val tm = rhs (concl th1)
  in
  (* continued *)
```

```
    (let
       val {Rand = tm1, Rator = tm2} = dest_comb tm
       val {Rand = tm3, Rator = tm4} = dest_comb tm2
       val th2 =
         if tm4 = plusterm then
           if zeroterm = tm3 then
             SPEC tm1 ADD1
           else
             if sucterm = rator tm3 then
               SPEC tm1 (SPEC (rand tm3) ADD2)
             else
               raise bad
         else
           raise bad
     in
       TRANS th1 (TRANS th2 (REDEPTH_QCONV (rhs (concl th2))))
       handle UNCHANGED => TRANS th1 th2
     end
     handle UNCHANGED => TRANS th1 (REDEPTH_QCONV tm))
     handle _ => th1
  end
  handle UNCHANGED =>
    ((let
        val {Rand = tm1, Rator = tm2} = dest_comb tm
        val {Rand = tm3, Rator = tm4} = dest_comb tm2
        val th2 =
          if tm4 = plusterm then
            if zeroterm = tm3 then
              SPEC tm1 ADD1
            else
              if sucterm = rator tm3 then
                SPEC tm1 (SPEC (rand tm3) ADD2)
              else
                raise bad
          else
            raise bad
      in
        TRANS th2 (REDEPTH_QCONV (rhs (concl th2)))
        handle UNCHANGED => th2
      end
      handle UNCHANGED => REDEPTH_QCONV tm)
      handle _ => raise UNCHANGED);

fun rewrite_plus tm = REDEPTH_QCONV tm
  handle UNCHANGED => REFL tm;
```