Journal of Automated Reasoning manuscript No.
(will be inserted by the editor)

Rewriting Conversions Implemented with Continuations

Michael Norrish

the date of receipt and acceptance should be inserted later

Abstract We give a continuation-based implementation of rewriting for systems in the LCF tradition. These systems must construct explicit proofs of equations when rewriting, and currently do so in a way that can be very space-inefficient. An explicit representation of continuations improves performance on large terms, and on long-running computations.

1 Introduction

Rewriting (or *simplification*) is a standard work-horse of all interactive theorem-proving systems. Rewriting is used both to expand the definitions of constants, and to apply previously proven equational results.

Despite its heavy use, rewriting in the various HOL systems (HOL4 [6, 8], HOL Light [7] and ProofPower [9]), and in Isabelle [12], is nonetheless implemented in "fully expansive" fashion, in keeping with these systems' LCF ancestry.[1] This means that if rewriting has transformed initial term t_1 to t_2, then the system will have explicitly proven the theorem $\vdash t_1 = t_2$ to justify this step. If the transformation is, for example, DNF normalisation by the rewrite rules

$$\begin{aligned} p \wedge (q \vee r) &= (p \wedge q) \vee (p \wedge r) \\ (p \vee q) \wedge r &= (p \wedge r) \vee (q \wedge r) \end{aligned}$$

then there may well have been exponentially many sub-equations to knit together to create the final equational theorem.

The task of assembling all of these sub-equations, the task of "replacing equals with equals" in the correct places and in the correct order, is the preserve of the rewriting system.

Michael Norrish
Canberra Research Lab., NICTA,
PO Box 8001, Canberra,
ACT 2601, AUSTRALIA
E-mail: Michael.Norrish@nicta.com.au

[1] The Coq system is also an "LCF-style" system, but the nature of its underlying logic means that achieving equational reasoning of the sort described in this paper is still an active research topic. See for example, Chrząszcz and Walukiewicz-Chrząszcz [5].

In the LCF-style systems examined here, this assembly of sub-equations ultimately depends on the rules of inference presented in Figure 1. The second rule implements transitivity; the last two are congruence rules, allowing the substitution of equals for equals in sub-terms. The rest of this introduction focuses on transitivity. Sections 2.4 and 2.5 discuss rewriting sub-terms using congruence rules.

$$\frac{}{\vdash x = x} \qquad \frac{\Gamma_1 \vdash x = y \quad \Gamma_2 \vdash y = z}{\Gamma_1 \cup \Gamma_2 \vdash x = z}$$

$$\frac{\Gamma_1 \vdash f = g \quad \Gamma_2 \vdash x = y}{\Gamma_1 \cup \Gamma_2 \vdash f\ x = g\ y} \qquad \frac{\Gamma \vdash t_1 = t_2}{\Gamma \vdash (\lambda x.t_1) = (\lambda x.t_2)}\ x \notin \Gamma$$

Fig. 1 Primitive Rules of Inference for Rewriting. The HOL4 names for these rules are `REFL`, `TRANS`, `MK_COMB` and `ABS`. In themselves, these rules can only derive instances of reflexivity, but can be used in conjunction with user-provided equations to derive more interesting results.

The HOL systems all use a rewriting technology ultimately derived from the influential paper by Paulson, *A Higher-Order Implementation of Rewriting* [11]. In this paper, the concept of a *conversion* is introduced:

> A term conversion is any function that maps a term t to a theorem ⊢t=u. This converts the term t to another term u, and proves the two equivalent. Since ML allows us to take theorems apart, we can extract the new term u from the theorem ⊢t=u.

In all of the HOL systems, the ML data type `term` implements well-typed terms of the simply-typed λ-calculus. In each HOL system, theorems (type `thm`) are created by functions defined in the system's kernel, or by derived functions that will ultimately depend on the kernel's facilities. A conversion is thus an ML function that manipulates the term and theorem types exported by the LCF-style kernel to produce equational theorems.

An atomic conversion might be one that demonstrates a β-equivalence: the function `BETA_CONV` takes, for example, the term $(\lambda x.x\,y)\,t$ and returns the theorem

$$\vdash (\lambda x.x\,y)\,t = t\,y$$

Or a conversion may match and instantiate a given rewrite rule. In HOL4 and HOL Light, `REWR_CONV` is of type `thm->(term->thm)`.[2] The first argument is the rewrite rule, and the second argument is the term to be transformed. For example, if `ADD_SUC` is the theorem

$$\vdash \mathsf{SUC}\,x + y = \mathsf{SUC}\,(x + y)$$

then `REWR_CONV ADD_SUC` is a conversion that will take the input term $\mathsf{SUC}\,3 + z$, and return the theorem

$$\vdash \mathsf{SUC}\,3 + z = \mathsf{SUC}\,(3 + z)$$

instantiating the variables x and y from `ADD_SUC` appropriately. (The issue of what happens when `REWR_CONV` is given a term that *doesn't* match the provided rewrite rule is addressed in Sections 2.2 and 3.1.)

[2] The analogue in ProofPower is `simple_eq_match_conv`.

The phrase "higher-order" in Paulson's title comes from the fact that conversions, themselves functions, can be combined using higher-order functions. The most important of these is THENC (HOL4 and HOL Light retain this name; ProofPower calls it THEN_C). A call to c_1 THENC c_2 (THENC is used as an infix) creates the sequential composition of the two conversions c_1 and c_2. An SML definition of THENC is given in Figure 2. As in the quotation from Paulson's original paper, the definition of THENC uses existing functions to take theorems apart. The function concl returns a theorem's conclusion, and the function rhs returns the right-hand-side of an equation. Finally, the rule of inference TRANS implements the rule of transitivity (the second rule from Figure 1). If both theorems are equations, and the left-hand side (l.h.s.) of the second is the same term as the right-hand side (r.h.s.) of the first, then a new theorem equating the l.h.s. of the first with the r.h.s. of the second is returned.

```
fun THENC(c1, c2) t = let
  val th1 = c1 t
  val th2 = c2 (rhs (concl th1))
in
  TRANS th1 th2
end
```

Fig. 2 An SML definition of THENC. The type of THENC is conv*conv -> conv, where conv is an abbreviation for term->thm. Using SML's facilities for infixes, one can write THENC(c1,c2) as c1 THENC c2.

It is now possible to examine the way in which the composite conversion

```
BETA_CONV THENC REWR_CONV ADD_SUC
```

will act on the input term $(\lambda x.\ x + 3)\,(\mathsf{SUC}\,10)$. First, the intermediate SML value th1 is created by applying BETA_CONV to the input term.

$$\texttt{val}\ \textit{th1}\ = \ \vdash (\lambda x.\ x + 3)\,(\mathsf{SUC}\,10) = \mathsf{SUC}\,10 + 3$$

The r.h.s. of the conclusion of this theorem is the term $\mathsf{SUC}\,10 + 3$, and this term is passed to the second conversion. The result will be

$$\texttt{val}\ \textit{th2}\ = \ \vdash \mathsf{SUC}\,10 + 3 = \mathsf{SUC}\,(10 + 3)$$

When TRANS is applied to *th1* and *th2*, the final result is returned:

$$\vdash (\lambda x.\ x + 3)\,(\mathsf{SUC}\,10) = \mathsf{SUC}\,(10 + 3)$$

1.1 The Problem with THENC, and a Simple CPS Solution

This technology (with a number of enhancements discussed in later sections) underpins rewriting in the HOL systems and Isabelle. Rewriting in these systems is undeniably important and the success of the systems is a credit to Paulson's original design. Nonetheless there is a problem, observable on large rewriting tasks, in all of the implementations.

When assembling a chain of rewrites, all of the systems put them together with the equivalent of THENC. The problem is that in a call (*c1* THENC *c2*), THENC does not relinquish control to the second conversion (*c2*). Instead, it waits for *c2* to return so that TRANS can be applied. In a chain of a million rewrites, the language implementation will create a million stack-frames, each waiting for its two sub-calls to finish so that the two results can be combined with TRANS. This can be very space-inefficient. Perhaps the easiest demonstration of this inefficiency is to set up a looping rewrite in any of the systems considered here, and to set it running. As the implementation diverges, the amount of memory consumed by the system increases.[3] This use of additional space can only be a waste of time. Nor would this consumption of space be necessary if the sequence of rewrites was making progress towards some goal rather than looping.

When confronted with code that is wasting space on the stack, a natural response is to try converting it to continuation-passing style. In the case of THENC however, this will not save any memory: the pending calls to TRANS will simply be stored on the heap as closures, and space saved on the stack will be wasted on the heap.

The solution is to represent continuations explicitly (rather than performing the usual conversion to continuations as closures), and to exploit the arising opportunities to merge pending calls to TRANS. Code to do just this is presented in Figure 3.

```
datatype cont =
  Conv of ((cont -> term -> thm) * cont)
| Trans of (thm * cont)
| Done

type cps_conv = cont -> term -> thm

fun apply_cont k th =
    case k of
      Done            => th
    | Conv (c,k')     => c (Trans (th,  k')) (rhs (concl th))
    | Trans (th', k') => apply_cont k' (TRANS th' th)

fun lift (cnv:term->thm) k t = apply_cont k (cnv t)
fun drop c = c Done

fun Cv (c, Trans(th, k)) = Trans(th,Conv(c,k))
  | Cv (c, k)            = Conv(c,k)

fun kTHENC (c1, c2) k t = c1 (Cv (c2,k)) t
```

Fig. 3 SML code implementing conversions in a continuation passing style (thus "cps-conversions"). The functions lift and drop allow cps-conversions and normal conversions to be inter-converted. The function Cv stops a Trans constructor from being buried under a Conv. Finally, kTHENC implements sequencing.

The data type cont is the type of continuations. Then, a "cps-conversion" takes not just a term to which it is applied, but a cont as well, representing what is to happen to the theorem that results. As continuations are not closures (functions), but rather the concrete type cont, the action of applying a continuation to a theorem must be implemented by a separate function (apply_cont).

The three different sorts of continuation in type cont can be summarised as follows

[3] In interactive sessions with Poly/ML (used in ProofPower, Isabelle, and also a development version of HOL4), just interrupting such a divergent execution takes a long time: the system is relatively slow to unwind its enormous chain of stack-frames. Moscow ML, used in the standard HOL4 implementation, raises an Out_of_memory quickly, signalling that the relatively limited space it has allocated for the stack has been exhausted. OCaml, used in HOL Light, grows its stack slowly, and terminates immediately when interrupted.

`Conv(c,k)` Continue with cps-conversion `c`, and after that, do `k`. The `Conv` constructor is thus used to stack up a to-do list of future work.

`Trans(th,k)` Continue with continuation `k`, but remember that we got this far *via* the equation in `th`.

`Done` Stop.

As is clear from the definition of `apply_cont`, the `Trans` constructor represents a pending call to the TRANS inference rule. The aim of the conversion to continuation-passing style is to ensure that these calls cannot proliferate.

1.1.1 A Worked Example

Let the term `t0` be $(\lambda x.\ (\lambda y.\ x+y)\,3)\,(\mathsf{SUC}\,10)$ and let `c1` and `c2` be the lifted conversions `lift BETA_CONV` and `lift (REWR_CONV ADD_SUC)` respectively. Further let `t1`, `t2` and `t3` be the successive terms created by applying those conversions to `t0`. That is,

$$\begin{aligned} \mathtt{t1} &= (\lambda y.\ \mathsf{SUC}\,10 + y)\,3 \\ \mathtt{t2} &= \mathsf{SUC}\,10 + 3 \\ \mathtt{t3} &= \mathsf{SUC}(10 + 3) \end{aligned}$$

Right-associating the calls to `kTHENC` in order to demonstrate the effect of the `Cv` function, consider the application

```
(c1 kTHENC (c1 kTHENC c2)) Done t0
```

The following computation will unfold:

```
  (c1 kTHENC (c1 kTHENC c2)) Done t
= c1 (Cv(c1 kTHENC c2, Done)) t0
= c1 (Conv(c1 kTHENC c2, Done)) t0
= apply_cont (Conv(c1 kTHENC c2, Done)) (BETA_CONV t0)
= apply_cont (Conv(c1 kTHENC c2, Done)) (⊢ t0 = t1)
= (c1 kTHENC c2) (Trans(⊢ t0 = t1, Done)) t1
= c1 (Cv(c2, Trans(⊢ t0 = t1, Done))) t1
= c1 (Trans(⊢ t0 = t1, Conv(c2,Done))) t1
= apply_cont (Trans(⊢ t0 = t1,Conv(c2,Done))) (BETA_CONV t1)
= apply_cont (Trans(⊢ t0 = t1,Conv(c2,Done))) (⊢ t1 = t2)
= apply_cont (Conv(c2,Done)) (TRANS (⊢ t0 = t1)(⊢ t1 = t2))
= apply_cont (Conv(c2,Done)) (⊢ t0 = t2)
= c2 (Trans(⊢ t0 = t2,Done)) t2
= apply_cont (Trans(⊢ t0 = t2,Done)) (REWR_CONV ADD_SUC t2)
= apply_cont (Trans(⊢ t0 = t2,Done)) (⊢ t2 = t3)
= apply_cont Done (TRANS (⊢ t0 = t2) (⊢ t2 = t3)
= apply_cont Done (⊢ t0 = t3)
= ⊢ t0 = t3
```

The only places where work is not peformed at the head position is when calling the lifted conversions, when calling TRANS, and when calling `Cv`. The latter is a constant-time pattern match. The other calls are where the work of the rewriting is done, and unavoidable. The advantage of this approach is that the two calls to TRANS are made as soon as possible.

It is interesting to note that the calls to TRANS made by

```
(BETA_CONV THENC (BETA_CONV THENC REWR_CONV ADD_SUC)) t0
```

would be

```
TRANS (⊢ t1 = t2) (⊢ t2 = t3)
TRANS (⊢ t0 = t1) (⊢ t1 = t3)
```

The same sequence as in the cps-conversion computation could be produced by calling the left-associated

```
((BETA_CONV THENC BETA_CONV) THENC REWR_CONV ADD_SUC) t0
```

However, reassociating the cps-conversion makes no difference; it will always make the same calls to `TRANS`.

1.1.2 Logic and Algebra

Definition 1 A `cont` value is *well-formed* if it contains no occurrence of the `Trans` constructor, or if it contains just one, which must be at the outermost position.

By inspection of the defined operations, an important invariance result follows:

Theorem 1 *The operations of Figure 3 preserve* well-formed *continuations.*

Definition 2 A call of cps-conversion c with arguments k and t is *legitimate* if k is `Trans`-free, or if k is of the form `Trans`$((\vdash\ t_0\ =\ t), k_0)$ for some term t_0 and some (`Trans`-free) continuation k_0. In other words, a call `c k t` is legitimate if `k` is well-formed, and additionally requires that if `k` has a top-most `Trans`, then the term `t` must be the r.h.s. of the `Trans` theorem.

If c_1 `kTHENC` c_2 is called legitimately, then the call it makes to c_1 will also be legitimate. Further, the call to a cps-conversion in `apply_cont` is always legitimate.

This “`THENC`-fragment” is as yet a rather impoverished language, but we can implement a left and right-identity for `kTHENC`. This is `kALL_CONV`:

```
fun kALL_CONV k t = apply_cont k (REFL t)
```

where `REFL` is the SML function that takes a term t and returns the theorem $\vdash t = t$. It thus implements the first rule of inference from Figure 1.

Definition 3 Say that a cps-conversion c is *well-behaved* if

- when legitimately applied to a continuation k, and to a term t, execution of $c\ k\ t$ either aborts, or eventually reaches `apply_cont` $k\ \ th$, where th is a theorem of the form $\vdash\ t = u$ for some term u, and where u is the same for all possible k. Further, if $c\ k\ t$ aborts, then so too does $c\ k'\ t$ for all k'.

Note that `lift c` is automatically well-behaved, and that c_1 `kTHENC` c_2 is well-behaved if c_1 and c_2 are.

Theorem 2 *The conversion `kALL_CONV` is an identity (extensionally) for `kTHENC`.*

Left identity for a well-behaved c, and a `Trans`-free k follows from:

$$\begin{aligned}
& (\texttt{kALL\_CONV}\ \texttt{kTHENC}\ c)\ k\ t && \\
= {} & \texttt{kALL\_CONV}\ (\texttt{Cv}(c,k))\ t && \text{(def'n of } \texttt{kTHENC}\text{)} \\
= {} & \texttt{apply\_cont}\ (\texttt{Cv}(c,k))\ (\vdash t = t) && \text{(def'n of } \texttt{kALL\_CONV}\text{)} \\
= {} & c\ (\texttt{Trans}((\vdash t = t), k))\ t && \text{(def'n of } \texttt{apply\_cont}\text{)} \\
= {} & \texttt{apply\_cont}\ (\texttt{Trans}((\vdash t = t), k))\ (\vdash t = u) && \text{(well-behaved)} \\
= {} & \texttt{apply\_cont}\ k\ (\texttt{TRANS}\ (\vdash t = t)\ (\vdash t = u)) && \text{(def'n of } \texttt{apply\_cont}\text{)} \\
= {} & \texttt{apply\_cont}\ k\ (\vdash t = u) && (\texttt{TRANS})
\end{aligned}$$

This is the same result that would arise if the initial call was just $c\ k\ t$. □

If k has an outermost `Trans`, the proof requires the definition of `apply_cont` to be expanded in the various calls `apply_cont` k.

The fact that `kALL_CONV` is also a right-identity for `kTHENC` follows from the fact that

$$\texttt{TRANS}\ (\vdash t = u)\ (\vdash u = u) \quad = \quad (\vdash t = u)$$

and similar equational reasoning.

1.1.3 Experimental Performance

With a CPS version of `THENC` implemented, it is possible to test the performance of the cps-conversions in comparison with the original implementation. We construct an admittedly artificial rewriting test by defining a function f:

$$\begin{aligned} f\ 0 &= 0 \\ f\ (\textit{SUC}\ n) &= f\ n \end{aligned}$$

This function has the property that a term such as $f(\texttt{SUC}(\texttt{SUC} \cdots 0))$ (with n applications of `SUC`) can be repeatedly rewritten without ever needing to descend into sub-terms (which important feature is described below in Sections 2.4 and 4).

The experimental task is to rewrite a term $f\ (\texttt{SUC}^n 0)$ to $f\ 0$, where varying n allows us to generate larger problems as necessary. Assume that `f0` is the theorem $\vdash f(0) = 0$ and that `fSUC` is $\vdash f(\texttt{SUC}\ n) = f(n)$. Further define "$n$-fold conversion" functions, `NCONV` and `kNCONV`:

```
fun NCONV n c t =
  if n = 0 then REFL t
  else (c THENC NCONV (n - 1) c) t

fun kNCONV n c k t =
  if n = 0 then apply_cont k (REFL t)
  else (c kTHENC kNCONV (n - 1) c) k t
```

Then the conversion used to test the standard implementation is

```
NCONV n (REWR_CONV fSUC)
```

and the corresponding cps-conversion is

```
drop (kNCONV n (lift (REWR_CONV fSUC))
```

Experiments were run comparing the performance of these code-snippets on terms of size up to $n = 2^{19}$. Figure 4 is a graphical presentation of the results, with graphs for both the standard HOL4 implementation in Moscow ML, and a second for the as-yet unreleased Poly/ML implementation. These, and all subsequent, experiments were run on a 2.33 GHz Intel Core Duo in a Macbook Pro running MacOS 10.5. The use of the Poly/ML implementation as well as the Moscow ML implementation provides some confidence that results are not the result of a quirk in the SML implementation.

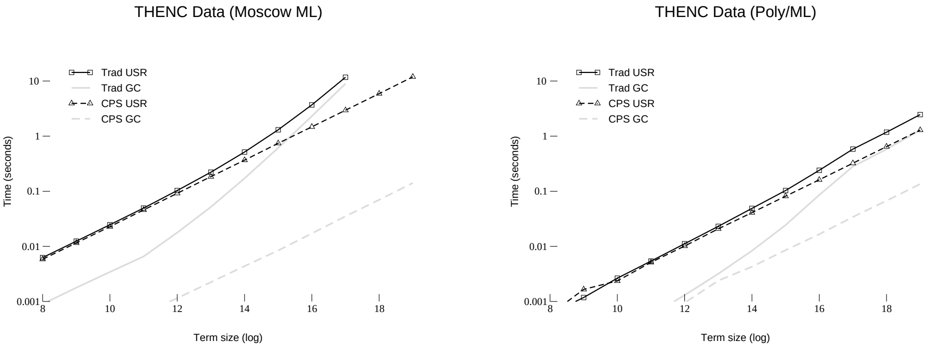


Fig. 4 Timing statistics for the THENC experiment. The data-points are means collected over 100 runs. The Moscow ML implementation raises an Out_of_memory exception on terms of size $\geq 2^{18}$, meaning that its runtime stack has been exhausted. At size $n = 2^{19}$, the Poly/ML CPS implementation has $\mu = 1.205$, $\sigma = 0.010$, while the traditional implementation has $\mu = 3.138$, $\sigma = 0.586$. In the Moscow ML CPS implementation at the same point, $\mu = 11.649$ and $\sigma = 0.149$.

The graphs make it clear that the cps-conversions are both quicker, and that they stress the implementations' memory management much less. Running with Moscow ML, the performance of the traditional implementation is worse than linear whereas the CPS implementation retains linear performance. Under Poly/ML, the effect is harder to see at the term sizes tested, but here it also appears as if garbage collection is causing the performance of the traditional implementation to become worse than linear. In any case, at the maximum term size, the CPS implementation is more than two and a half times faster.

The rest of this paper will first describe Paulson's original system [11], give a detailed description of how it has been updated and implemented in HOL, and touch on two further problems with the system (Section 2). Then, the paper will demonstrate how the simple continuation-based system already presented can be extended to implement these important features, and to do so efficiently.

2 Paulson's System and its Implementation in HOL

The important functions from Paulson [11] are

THENC Sequencing of two conversions c_1 and c_2, such that if c_1 transforms t_0 to t_1, and c_2 transforms t_1 to t_2, then (c_1 THENC c_2) transforms t_0 to t_2 (producing the theorem $\vdash t_0 = t_2$ along the way).

`ORELSEC` Alternation of two conversions, which may raise exceptions to indicate failure. The expression $(c_1$ `ORELSEC` $c_2)$ is a conversion which will first try c_1 on input t. If that application succeeds, then that is the result. If $c_1\ t$ raises an exception, then $c_2\ t$ is tried.

`REPEATC` Repetition of a conversion. `REPEATC` c applies c to its input, and if that succeeds, applies c to the result, and so on. When c eventually fails, returns the accumulated effect of the previous successful calls.

`COMB_CONV` Applies a conversion to the sub-terms of an application term. If the argument t is the application $f\ x$, then `COMB_CONV` $c\ t$ will result in c being called on f and x independently, and the results combined, deriving a theorem of the form $\vdash f\ x = f'\ x'$

`ABS_CONV` Applies a conversion to the body of an abstraction term. If the argument t is the abstraction $(\lambda x.M)$, then `ABS_CONV` $c\ t$ will apply to c to M, returning a theorem of the form $\vdash (\lambda x.M) = (\lambda x.M')$

`TOP_DEPTH_CONV` Applies a conversion repeatedly at all positions within a term, sweeping across the term in a top-down fashion. This is just one of a family of possible rewriting strategies for term traversal. Its implementation relies on all of the other functions described so far.

These functions are written in ML, and rely on basic infrastructure for manipulating the terms and theorems of the kernel. A summary of these functions from HOL4 is given in Figure 5.

```
val total     : ('a -> 'b) -> 'a -> 'b option
val HOL_ERR   : ⟨exndata⟩ -> exn
val UNCHANGED : exn

val aconv     : term -> term -> bool

(* access sub-terms *)
val dest_comb : term -> term * term
val dest_abs  : term -> term

(* split equation terms *)
val lhs       : term -> term
val rhs       : term -> term
val dest_eq   : term -> term * term

val concl     : thm -> term

(* theorem primitives, as per Figure 1 *)
val REFL      : term -> thm
val TRANS     : thm -> thm -> thm
val MK_COMB   : thm -> thm -> thm
val ABS       : thm -> thm
```

Fig. 5 Functions from the HOL4 API. The utility function `total` is used to catch exceptions, turning a successful application of $f\ x$ into `SOME` *result*; if $f\ x$ raises an exception, then the result of `total` $f\ x$ is `NONE`. The `HOL_ERR` value is the standard way for HOL functions to report failures. The `UNCHANGED` exception is used to indicate a successful rewriting step that doesn't change its input; see Section 2.6. A call to `aconv` $t_1\ t_2$ returns true iff the two terms are alpha-convertible. The functions `dest_comb` and `dest_abs` pull apart application and abstraction terms respectively, returning immediate sub-terms. Both raise `HOL_ERR` exceptions if the terms are not of the right shape.

2.1 Sequencing and its Identity

Repeating from Figure 2, the HOL4 definition of THENC is

```
fun THENC(c1, c2) t = let
  val th1 = c1 t
  val th2 = c2 (rhs (concl th1))
in
  TRANS th1 th2
end
```

The identity for THENC is ALL_CONV which when applied to a term t returns the theorem $\vdash t = t$. In other words, ALL_CONV is simply the same as the primitive rule REFL.

Section 1.1 made it clear how the pending call to TRANS, waiting for *c1* and *c2* to return, is not necessarily space-efficient.

2.2 Failure

A conversion can indicate that it has failed on a particular input by raising an exception. In HOL4, this exception will be a HOL_ERR. The implementation of ORELSEC is simple.

```
fun ORELSEC(c1,c2) t = c1 t handle HOL_ERR _ => c2 t
```

Like THENC, ORELSEC is typically used as an infix. The HOL4 implementation only catches HOL_ERR exceptions (which take as a parameter additional information about the nature of the error). Other exceptions, such as the built-in Interrupt, or those users might care to invent, are allowed to propagate out.

Following Paulson, there is an identity conversion for ORELSEC, NO_CONV:

```
fun NO_CONV t = raise HOL_ERR ⟨data⟩
```

2.3 Repetition with REPEATC, a Poor Man's Loop

REPEATC, which applies a conversion repeatedly until it fails, can be defined

```
fun REPEATC c t =
  ((c THENC REPEATC c) ORELSEC ALL_CONV) t
```

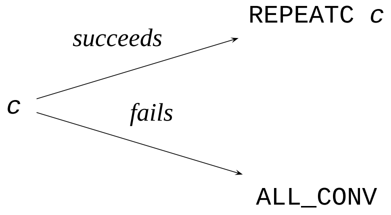
(The eta-expansion to include the argument *t* is necessary to avoid an immediate divergence.)

There is a problem with REPEATC. It lies with its poor structuring of control-flow, which in turn leads to the proliferation of unnecessary exception-handlers, and thus the unnecessary consumption of memory. After expanding the outermost call to ORELSEC, the execution state is

```
(c THENC REPEATC c) t handle HOL_ERR _ => ALL_CONV t
```

The exception handler has scope over not just *c* but the call to REPEATC. This is unnecessary because the call to REPEATC cannot raise an exception. Worse, each successive call to REPEATC will create a fresh handler.

Conceptually, the exception handler is only present to catch exceptions arising in *c*. The flow control should be



To handle this possibility, an explicit conditional conversion can be written. In the traditional style, this is rather awkward (the cps-conversion `kIFC` of Section 3.2 is much more elegant):

```
fun IFC(g,s,f) t =
  case total g t of
    SOME th => TRANS th (s (rhs (concl th)))
  | NONE => f t
```

The implementation doesn't refer to `THENC`, but of course it has `THENC`'s standard problem: the call to `TRANS` is suspended until *s* returns.

Nonetheless, a new implementation of `REPEATC` can be written, here called `iREPEATC` to distinguish it from Paulson's original:

```
fun iREPEATC c t = IFC(c,iREPEATC c,ALL_CONV) t
```

2.4 Rewriting Sub-terms

Rewriting systems must be able to apply their rewrites at any position within the term that is being rewritten. This ability is permitted by the two congruence rules presented in Figure 1. In the HOL4 implementation, these rules are implemented by two primitive rules of inference: `ABS`, and `MK_COMB`. The first implements the congruence for abstractions: given an equational theorem $\Gamma \vdash t_1 = t_2$ and a variable v, a call to `ABS` v $(\Gamma \vdash t_1 = t_2)$ returns the theorem $\Gamma \vdash (\lambda v.t_1) = (\lambda v.t_2)$, with an exception occurring if $v \in \Gamma$. The rule `MK_COMB` takes a pair of theorems $\Gamma_1 \vdash f = g$ and $\Gamma_2 \vdash x = y$ and returns the theorem $\Gamma_1 \cup \Gamma_2 \vdash f(x) = g(y)$, as long as f (and g) are of function type with domain equal to the type of x (and y).

The functions `COMB_CONV` and `ABS_CONV` are used to rewrite with these rules. An SML implementation of `COMB_CONV` is

```
fun COMB_CONV c t = let
  val (rator,rand) = dest_comb t
in
  MK_COMB(c rator, c rand)
end
```

where the function `dest_comb` pulls apart an application (or "combination" term) returning the pair of the function and argument terms. If the term is not an application (is an abstraction, variable or constant), or if the call to `MK_COMB` fails because the equations provided are not compatible, an exception is raised.

The implementation of `ABS_CONV` is similar.

2.5 Term Traversal Strategies

The standard rewriting strategy in HOL4 is:

> *Repeatedly apply rewrites at the top level, as long as any apply. Then descend recursively into sub-terms. If anything changes from this traversal, try a rewrite at the top-level again. If this succeeds, repeat.*

The essence of this strategy is captured in the function TOP_DEPTH_CONV. This is another higher-order conversion, which takes as an argument the conversion to perform rewrites at the top position of a term. It is then the responsibility of TOP_DEPTH_CONV to organise the term-traversal. This function is similar to, but not quite the same as the TOP_DEPTH_CONV in Paulson [11]. The difference is that Paulson's version doesn't check if the recursion changed the term or not. The HOL4 implementation is

```
fun TOP_DEPTH_CONV c t =
  (REPEATC c THENC
   TRY_CONV
      (CHANGED_CONV (SUB_CONV (TOP_DEPTH_CONV c)) THENC
       TRY_CONV (c THENC TOP_DEPTH_CONV c))) t
```

There are three new functions here. Two are simple:

```
fun TRY_CONV c = c ORELSEC ALL_CONV
fun SUB_CONV c = TRY_CONV (COMB_CONV c ORELSEC ABS_CONV c)
```

A call to CHANGED_CONV c t applies c to t and checks to see if the result is an instance of reflexivity. If so, it fails. The implementation is straightforward:

```
fun CHANGED_CONV c t = let
   val th = c t
   val (l,r) = dest_eq (concl th)
in
   if aconv l r then raise HOL_ERR ⟨No change error⟩
   else th
end
```

2.6 Avoiding Excessive Term-Rebuilding

There is one extremely important feature of the modern LCF-style rewriting systems that is not in Paulson [11]: the use of special return values to record that a term has not been changed by a rewriting step.

The standard term traversal strategies repeatedly examine all of a term looking for possible matches for rewrite rules. When rewriting an application term $f\ x$, the traversal will eventually create theorems of the form $\vdash f = g$ and $\vdash x = y$ that will be fed into the rule MK_COMB. However, it can often happen that a term doesn't change. Then both theorems will be instances of reflexivity, and so too will the output from MK_COMB.

If a term is traversed, but doesn't contain any matches for the supplied rewrite rules, the action of all the calls to MK_COMB and ABS will be to create fresh copies of the original's internal structure (though not its leaves). This follows from the kernel's implementation of these rules of inference. For example, the kernel's MK_COMB will be something like

```
fun MK_COMB (fth,xth) = let
  val (f,g) = dest_eq (concl fth)
  val (x,y) = dest_eq (concl xth)
in
  mk_thm(union_hyp (hyp fth) (hyp xth),
         mk_eq(mk_comb(f,x), mk_comb(g,y)))
end
```

where the calls to `dest_eq` check that the input theorems really are equalities, and where the calls to `mk_comb` check that the types of the functions link up with those of the argument appropriately before creating an application term.

When rewriting, the original term being traversed was of the form `mk_comb(f,x)`, and this code will create this term again to serve as the output theorem's l.h.s. If *g* is syntactically equal to *f*, and *x* is syntactically equal to *y*, then the second call to `mk_comb` will end up creating a *second* copy of the original.

One might imagine extending the kernel to provide a version of `MK_COMB` that took the original application term as an argument, and that used a constant-time pointer-equality test to check both that the input theorems did actually equate the sub-terms of the original, and also to avoid creating a fresh r.h.s. when the sub-terms hadn't changed. (If the necessarily approximate pointer-equality test failed on either check, the new rule would fall back to calling `mk_comb`.)

Rather than tailor the kernel to this one (admittedly important) application, the LCF rewriting implementations all use a different trick: they change their conversions so that they can signal that the input term has not changed. In HOL4, HOL Light and ProofPower, this is done by having conversions raise an exception. In Isabelle's simplifier, the process of rewriting uses the option type to do the same thing. This idea is extremely important for efficiency. Boulton [4], who is reponsible for its HOL implementation within the Paulson framework of conversions, credits it to Huet.

In HOL4, the exception used is `UNCHANGED`. The function `THENC` (first in Figure 6) changes, with two handlers set up to cope with the possibility of either conversion raising `UNCHANGED`. The bare call to `c2 t` naturally admits the possibility that the call to `THENC` will also raise `UNCHANGED`, when both conversions do the same. The definitions of `ORELSEC`, `NO_CONV`, `REPEATC`, `ABS_CONV`, `SUB_CONV`, `TRY_CONV`, and also of the term-traversal operator `TOP_DEPTH_CONV` are unchanged.

The new implementation of `COMB_CONV` (third in Figure 6) has a similar feel to that of `THENC`, but it becomes fiddlier to get the scopes right (the nested **`let-in-end`** is necessary so that the outermost handler can still "see" the value of sub-terms `f` and `x`).

Note that a successful traversal will still end up creating a copy of the original term on the l.h.s. of the resulting theorem. (More accurately: any internal structure above sub-terms that change on the r.h.s. will be created afresh on the l.h.s. as well.)

Finally, `CHANGED_CONV` needs to now check that its call to `c t` hasn't raised the `UNCHANGED` exception. If it does, this needs to be converted to a `HOL_ERR`.

3 The CPS-Conversion System

When moving beyond the introduction's simple system (which only implements sequencing with `kTHENC`), there are two critical new features that influence the design. The first is the possibility of failure.

```
fun THENC(c1,c2) t = let
  val th1 = c1 t
in
  TRANS th1 (c2 (rhs (concl th1))) handle UNCHANGED => th1
end handle UNCHANGED => c2 t

fun ALL_CONV t = raise UNCHANGED

fun COMB_CONV c t = let
  val (f,x) = dest_comb t
in
  let
    val fth = c f
    val xth = c x handle UNCHANGED => REFL x
  in
    MK_COMB(fth, xth)
  end handle UNCHANGED => MK_COMB(REFL f, c x)
end

fun CHANGED_CONV c t = let
   val th = c t handle UNCHANGED => raise HOL_ERR ⟨No change error⟩
   val (l,r) = dest_eq (concl th)
in
   if aconv l r then raise HOL_ERR ⟨No change error⟩
   else th
end
```

Fig. 6 HOL4 implementations of THENC, ALL_CONV, COMB_CONV and CHANGED_CONV when the UNCHANGED exception is being used to signal that a conversion has not affected its input term.

To create an analogue of raising an exception in the world of cps-conversions, the basic type changes. A cps-conversion will now include two continuations: one for a successful computation, and one for failure. This is a well-known treatment for exceptions in a CPS style. There is also a new form of continuation, ReturnTo, specifying a term to return to before continuing with a failure continuation. This constructor will be explained further in Section 3.1, where we will see how this new constructor parallels the existing Trans form.

The second critical feature from Section 2 that informs the design of the full CPS system is the use of the UNCHANGED exception to indicate that a conversion has succeeded, but that it has not changed the input. In the world of cps-conversions, success continuations need to be able to transparently handle the possibility that they can be called in a state where a previous conversion has not returned a theorem, but has instead signalled an unchanged condition.

Our approach is to create a new sum type, convresult:

```
datatype convresult = TM of term | THM of thm
```

The TM constructor indicates that a conversion has signalled “unchanged” on the term argument to TM. The THM constructor is used for the normal return of an theorem equating the input term to some result.

The type of a continuation (cont) changes to include the new failure continuation ReturnTo, and to have a closure as the argument to Conv. We also note that the ReturnTo and Done forms will only occur in failure continuations, and that the Trans constructor will only occur in success continuations.

```
datatype cont = Conv of (convresult -> convresult)
              | Trans of (thm * cont)
              | ReturnTo of (term * cont)
              | Done
```

Ignoring ReturnTo for the moment, there are two natural questions to ask about this design. Why is there a closure as an argument to Conv, and why does the closure have the type that it does?

To answer the second question first: a continuation is an implementation of a function that consumes the result type of the underlying computation, and then performs some further transformation. (Of course, in the traditional CPS translation of functions, the continuations *are* functions.) Because the aim of this design is to stay within the space of cps-conversions (allowing them to be combined using Paulson's algebraic operators), the "further transformations" will naturally produce the result type again.

The use of the function space under the Conv constructor allows flexibility. There are too many sorts of continuation possible when one comes to implement operators like COMB_CONV and CHANGED_CONV. Trying to encode these concretely leads to the creation of a very *ad hoc* and confusing data type. Using a closure under Conv still allows the TRANS-merging optimisation of the introduction because the Trans constructor remains available for just that purpose.

The type of a cps-conversion is then

```
type cps_conv = cont -> cont -> term -> convresult
```

Precisely because the cont type is not simply a closure, the action of a cont on a convresult must be explicitly defined. Figure 7 gives the action of a success continuation in two separate functions, one for theorems (the THM constructor), and one for terms (the TM constructor). Next, we must treat failure continuations.

```
fun apply_cont k th =
  case k of
    Conv f => f (THM th)
  | Trans(th0,k) => apply_cont k (TRANS th0 th)

fun apply_unchanged k t =
  case k of
    Conv f => f (TM t)
  | Trans(th,k0) => apply_cont k0 th
```

Fig. 7 Functions apply_cont and apply_unchanged, detailing how to apply a success continuation to the two sorts of conversion result, theorems and terms respectively.

3.1 Failure Continuations

A failure continuation represents what should be done if a conversion fails. If a conversion does fail, there will not be a theorem to apply the continuation to. Instead, the term to which the conversion that failed was applied will be available. As a result, one might imagine that the apply_fail function for applying a continuation should have type

```
cont -> term ->convresult
```

In order to support the provision of extra data about a conversion's failure, we actually use the type

```
cont -> exn -> term -> convresult
```

In this way, the ultimate result of applying a failure continuation can still be the raising of an exception.

Now, consider the conversion

```
(c1 THENC c2) ORELSEC c3
```

Here, the conversion `c3` will get called if either `c1` or `c2` fails. Imagine that in fact, `c1` successfully transforms some input term `t0` to `t1`. If `c2` then fails, `c3` should be applied to `t0`, not `t1`. When this conversion is translated to its CPS analogue, this analysis suggests that although the failure continuation when `c2` is called should get handed term `t1`, the continuation should itself know to call `c3` on `t0`. This is done by having the failure continuation use the `ReturnTo` constructor to encode the requirement to return to some previous term.

The definition of `apply_fail` is given in Figure 8.

```
fun apply_fail k e t =
  case k of
    Done => raise e
  | ReturnTo(t',k') => apply_fail k' e t'
  | Conv f => f (TM t)
```

Fig. 8 The definition of `apply_fail`, the function that computes the effect of applying a failure continuation to a term. The additional exception argument allows the world of cps-conversions to raise an exception if it runs out of other things to do (has a `Done` continuation). The term argument will have been the argument to a conversion that failed.

Just as the simple system of the introduction avoided having multiple `Trans` constructors in a continuation, it makes sense to avoid having chains of `ReturnTo` constructors. In a continuation like

```
ReturnTo(t1,ReturnTo(t2,k))
```

the outermost `ReturnTo` is redundant; the continuation might as well call for a direct return to `t2`. This minor optimisation is performed by the `RT` function:

```
fun RT (t1, k as ReturnTo _) = k
  | RT (t,  c)               = ReturnTo(t, c)
```

3.2 Sequencing and Alternation: `THENC`, `ORELSEC` and `IFC`

The next component of the full CPS-system is the `continue` function. This implements the action of applying a conversion to the result of a conversion, thereby allowing conversions to be composed, or chained together. In the vanilla Paulson-system, the result of a conversion is a theorem. One can see the Paulson-analogue of `continue` by splitting `THENC` thus:

```
fun pcontinue c th  = TRANS th (c (rhs (concl th)))
fun (c1 THENC c2) t = pcontinue c2 (c1 t)
```

In the CPS-system, the `continue` function takes a cps-conversion, success and failure continuations and a value of type `convresult`. If the `convresult` is a term, meaning that the previous conversion left its argument unchanged, then the cps-conversion argument should just be applied to that term, along with the continuations. This is also the behaviour that is required when a conversion is applied to a bare term on which some other conversion has failed.

If the `convresult` is a theorem, then the cps-conversion needs to be called on the r.h.s. of that theorem's conclusion. In addition, both the failure and success continuations need to be adjusted. The success continuation needs to store the theorem with the `Trans` constructor so that a call to `TRANS` can be made later. The failure continuation needs to record the l.h.s. of the theorem, so that if the cps-conversion fails, the failure continuation can return to the original term. The full definition of `continue` is given in Figure 9.

```
fun continue c k f (TM t) = c k f t
  | continue c k f (THM th) = let
      val (l,r) = dest_eq (concl th)
    in
      c (Trans(th,k)) (RT(l,f)) r
    end

fun SCont c k f =
  case k of
    Trans(th,k') => Trans(th,Conv(continue c k' f))
  | _ => Conv(continue c k f)
fun FCont c k f = Conv(continue c k f)

fun kTHENC   (c1,c2) k f t = c1 (SCont c2 k f) f              t
fun kORELSEC (c1,c2) k f t = c1 k              (FCont c2 k f) t
fun kIFC   (g,c1,c2) k f t = g  (SCont c1 k f) (FCont c2 k f) t
```

Fig. 9 Fundamental sequencing and alternation operators (`kTHENC`, `kORELSEC` and `kIFC`). The `continue` function applies a cps-conversion to the two usual continuations and a `convresult`. Also shown are the `SCont` and `FCont` functions, which construct success and failure `cont` values embodying `continue`. The `SCont` function keeps any `Trans` constructor uppermost.

The expression `continue c k f` is a function taking one `convresult` to another, so it can be given as an argument to the `Conv` constructor to create a continuation (`cont`) value.

With `continue` defined, it is then possible to define the sequencing and alternation operators `kTHENC`, `kORELSEC` and `kIFC`, as in Figure 9. The first two call the first conversion (*`c1`*), and push *`c2`* onto the success or failure continuations respectively. The `kIFC` operator calls its argument *`g`*, and pushes the other two conversion arguments onto the success and failure continuations.

Note how this design overloads `continue` so that it is used to apply a conversion both to the result of a successful conversion, and to a term when a conversion fails. When working with a failure continuation, only the first branch of `continue`'s definition will be called. This is why the `FCont` auxiliary does not need to worry about keeping `Trans` constructors uppermost, as is done by `SCont`.

3.3 Connection with Traditional Conversions

The final elements of the CPS-system are `lift` and `drop`. A naïve attempt to write `lift` might be

```
fun lift (cnv : term -> thm) k f t =
  apply_cont k (cnv t)
  handle e as HOL_ERR _ => apply_fail f e t
       | UNCHANGED => apply_unchanged k t
```

This is incorrect because the scope of the exception handler ranges over not just `cnv t`, but also `apply_cont`. This is the problem diagnosed and discussed in Benton and Kennedy [2], and unfortunately, there is little option here but to work around with a custom sum type.

Begin with a new data type, and a revised form of the `total` function:

```
datatype 'a result = OK of 'a | UNCH | ERR of exn
fun total f x =
  OK (f x) handle e as HOL_ERR _ => ERR e
                | UNCHANGED => UNCH
```

Then, `lift` is

```
fun lift c k f t =
  case total c t of
    OK th => apply_cont k th
  | UNCH  => apply_unchanged k t
  | ERR e => apply_fail f e t
```

The `drop` function for turning a cps-conversion into a normal conversion is simpler:

```
fun drop c t =
  case c (Conv(fn x => x)) Done t of
    TM   _ => raise UNCHANGED
  | THM th => th
```

3.4 Initial Derived Forms

Having set up the CPS-system carefully, it is possible to define derived operators in ways that directly mimic definitions made in Paulson's system. The identities for `kTHENC` and `kORELSEC` are `kALL_CONV` and `kNO_CONV` respectively:

```
fun kALL_CONV k f t = apply_unchanged k t
fun kNO_CONV  k f t =
  apply_fail f (HOL_ERR ⟨kNO_CONV data⟩) t
```

Repetition should be implemented with `kIFC`, giving

```
fun kREPEATC c k f t = kIFC(c,kREPEATC c,kALL_CONV) k f t
```

The function `REPEATC` has type `conv -> conv`, and the corresponding `kREPEATC` has type `cps_conv -> cps_conv`.

A definition for `kREPEATC` analogous to Paulson's original definition would be

```
fun kREPEATC c k f t =
  ((c kTHENC kREPEATC c) kORELSEC kALL_CONV) k f t
```

but experiments reveal that this is less efficient.

When the operators for moving into sub-terms have been defined, the definition of `TOP_DEPTH_CONV` in the CPS-system can also directly mimic the original definition.

3.5 Algebra

Theorem 3 *The cps-conversion* `kNO_CONV` *is extensionally a left-identity for* `kORELSEC`*.*

$$\begin{array}{lll} & (\texttt{kNO\_CONV kORELSEC}\ c)\ k\ f\ t & \\ = & \texttt{kNO\_CONV}\ k\ (\texttt{Conv}(c,k,f))\ t & \text{(def'n of } \texttt{kORELSEC}\text{)} \\ = & \texttt{apply\_fail}\ (\texttt{Conv}(c,k,f))\ \langle \textit{error-exn} \rangle\ t & \text{(def'n of } \texttt{kNO\_CONV}\text{)} \\ = & c\ k\ f\ t & \text{(def'n of } \texttt{apply\_fail}\text{)} \end{array}$$

As with the analogues `ORELSEC` and `NO_CONV` in the HOL4 implementation, `kNO_CONV` is not quite a right identity for `kORELSEC`. In c `kORELSEC kNO_CONV`, if c raises an exception, then the result of the expression will be `kNO_CONV`'s exception, not `c`'s.

3.6 Experimental Performance: Flavours of `REPEATC`

To test conversion repetition, the function f from the `THENC` tests of the introduction can be re-used

$$\begin{aligned} f\ 0 &= 0 \\ f\ (\textit{SUC}\ n) &= f\ n \end{aligned}$$

The experimental task is to rewrite a term $f\ (\texttt{SUC}^n\ 0)$ to zero. The theorem $\vdash f(0) = 0$ is called `f0` and `fSUC` is $\vdash f(\texttt{SUC}\ n) = f(n)$. The experiments compare the behaviour of three different conversions, the traditional original:

```
REPEATC (REWR_CONV f0 ORELSEC REWR_CONV fSUC)
```

the same conversion with `REPEATC` replaced by `iREPEATC` from Section 2.3 (*i.e.*, `REPEATC` implemented with `IFC`), and the cps-conversion:

```
drop (kREPEATC (lift (REWR_CONV f0) kORELSEC
                lift (REWR_CONV fSUC)))
```

Each conversion attempts to rewrite with `f0` first. This attempt fails on all but the last term in the sequence taking $f(\texttt{SUC}^n\ 0)$ to 0. This tests the handling of exceptions in `ORELSEC` and its analogues. Conversely, the successful branch inside the repetition is taken on every term *except* the zero.

The experimental results (for Moscow ML) are presented in Figure 10. These demonstrate that the `IFC` optimisation is of considerable benefit when applied to traditional conversions, and that the cps-conversion performs better still. Not shown are results for the naïve cps-conversion written without `kIFC`. At maximum size, this version of the conversion took 13.3s, compared to 11.5s for the `kIFC` version.

4 Accessing Sub-terms with CPS-Conversions

Figure 11 gives all of the code necessary to process the sub-terms of an application term. The `done_right` function specifies how to proceed once both sub-terms have been processed (either, both or neither may have been unchanged by their respective conversions, so there is four-way case split). The `do_right` continuation applies a cps-conversion to a term

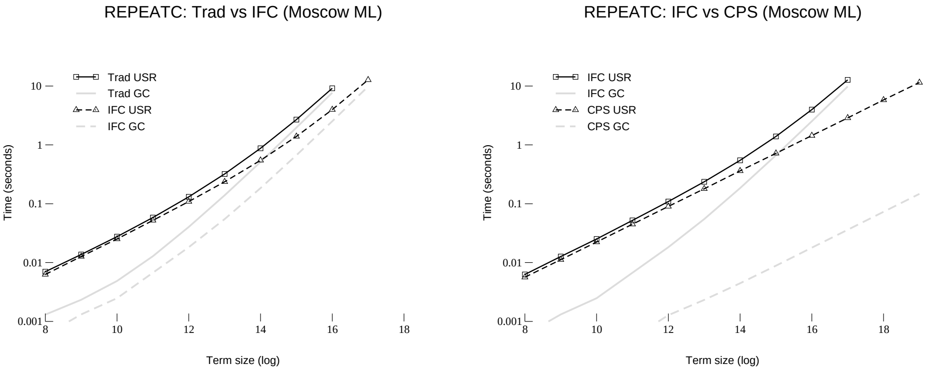


Fig. 10 Timing statistics for the REPEATC experiment, with user and garbage collection times. The data-points are means collected over 100 runs. At term size $= 2^{16}$, the mean elapsed times for Trad, IFC and CPS are 9.2s, 4.0s and 1.4s respectively. The Trad implementation raises an Out_of_memory exception on terms of size $\geq 2^{17}$; the IFC implementation does the same on terms of size $\geq 2^{18}$.

```
fun done_comb parent k left right =
  case (left,right) of
    (TM  _, TM  _) => apply_unchanged k parent
  | (TM  l, THM r) => apply_cont k (MK_COMB(REFL l, r))
  | (THM l, TM  r) => apply_cont k (MK_COMB(l, REFL r))
  | (THM l, THM r) => apply_cont k (MK_COMB(l, r))

fun do_right c parent k f x left =
  c (Conv(done_comb parent k left)) f x

fun kCOMB_CONV c k f t = let
  val fl = RT(t,f)
in
  case total dest_comb t of
    OK(t1,t2) => c (Conv(do_right c t k fl t2)) fl t1
  | ERR e => apply_fail f e t
end
```

Fig. 11 Accessing the sub-terms of an application term, leading to the implementation of kCOMB_CONV.

that is assumed to be the right-hand sub-term of an application. Finally, kCOMB_CONV is straightforward.

Using these primitives, it is also possible to implement cps-conversion analogues of the standard functions RATOR_CONV and RAND_CONV, which apply a conversion just to a particular sub-term. The new definition of kABS_CONV is omitted in the interests of space, but is included in the (available, see Section 6.2) source-code.

The system has become rather reminiscent of abstract machines for implementing particular evaluation strategies for the λ-calculus. Such a machine is presented in Barras [1], where it is used to perform very efficient applicative order “computation”. The difference between that work and this is that Barras is implementing a very specific evaluation strategy, while this work is meant to allow any strategy to be expressed. Computation in Barras’s sense also requires the system to include a notion of what constitutes a “value”.

In order to implement the analogue of TOP_DEPTH_CONV from Section 2.5, the CPS-system needs analogues of SUB_CONV, TRY_CONV and CHANGED_CONV. The first two can be translated directly (ORELSEC is replaced by kORELSEC *etc*). The analogue of

`CHANGED_CONV` is presented in Figure 12. This code handles the `UNCHANGED` exception when the argument `c` is called, and additionally looks at the r.h.s. of the resulting theorem. In the cps-conversion (second in Figure 12), if `c` leaves `t` unchanged, then the `check` auxiliary will be handed that term to compare with itself. As `aconv` uses a pointer-equality comparison internally as a fast-path check, the behaviour will still be constant-time, as desired.

```
fun kCHANGED_CONV c k f t = let
  val fail_exn = HOL_ERR ⟨No change error⟩
  fun kont (TM _) = apply_fail f fail_exn t
    | kont (THM th) =
        if aconv t (rhs (concl th)) then apply_fail f fail_exn t
        else apply_cont k th
in
  c (Conv kont) f t
end
```

Fig. 12 CPS-conversion implementation of `CHANGED_CONV`, a higher-order conversion that fails if its argument does not change its input. The `aconv` (read “alpha-convertible”) function tests if two terms are alpha-equivalent.

One can now produce an implementation of `kTOP_DEPTH_CONV` that directly parallels the traditional code. Though experiments reveal the slowdown to be smaller, this suffers from the same problem as `REPEATC` (identified in Section 2.3): exception handlers are given scope over functions that cannot raise exceptions, and these accumulate unnecessarily. It is better to implement `kTOP_DEPTH_CONV` as

```
fun kTOP_DEPTH_CONV c k f t =
  (kREPEATC c kTHENC
   kIFC(kCHANGED_CONV (kSUB_CONV (kTOP_DEPTH_CONV c)),
        kIFC(c, kTOP_DEPTH_CONV c, kALL_CONV),
        kALL_CONV)) k f t
```

4.1 Experimental Performance: Rewriting Sub-terms

Experiment #1: Ackermann’s Function The first test of the cps-conversions on a problem where (repeated) term-traversal is required is to rewrite with the following definition of Ackermann’s function:

```
Ack       0       n = SUC n
Ack (SUC m)       0 = Ack m (SUC 0)
Ack (SUC m) (SUC n) = Ack m (Ack (SUC m) n)
```

This definition is effectively three rewrite theorems. The action of trying each rewrite in turn can be done efficiently through the use of a standard HOL auxiliary `REWRITES_CONV`, and a specialised data structure called a term-net.

If the term-net for these three rewrites is `Ack_rwts`, then the traditional conversion is

```
TOP_DEPTH_CONV (REWRITES_CONV Ack_rwts)
```

and the cps-conversion is

```
drop (kTOP_DEPTH_CONV (lift (REWRITES_CONV Ack_rwts)))
```

In this way, the behaviour of the cps-conversion at the level of atomic rewriting steps directly mimicks the original conversion. The only difference lies in the efficiency, or otherwise, of the assembly of the resulting theorems.

The experiment compared the performance of these conversions when applied to terms of the form

$$\texttt{Ack (SUC(SUC(SUC 0))) (SUC}^{n}\texttt{ 0)}$$

for $0 \leq n \leq 8$. This computation is an extremely inefficient way of calculating $2^{(n+3)} - 3$ (in unary notation no less). The results for this experiment are presented in Figure 13. When having to recurse over the structure of increasingly large terms, remembering one's position along the way, the cps-conversions' advantage is less marked than in the earlier experiments. The cps-conversions cannot escape having to build up what is effectively a stack of positions. Moreover, the traversal only ever chains together at most three rewrites at the same term position (moving from a term matching the second clause of the definition to one matching the third, and then finally, one matching the first), vitiating the cps-conversions' strength in chaining together calls to `TRANS`.

Nonetheless, in the Moscow ML implementation, the cps-conversions are significantly quicker on the largest problem. In the Poly/ML implementation, the cps-conversions are slightly slower. It is not clear why this should be so. It is not the use of closures: a version of the cps-conversion system with concrete constructors for the data type `cont` performed much the same as the version with just `Conv` and `Trans` as constructors for `cont`. See Section 6.1 for more discussion of Poly/ML.

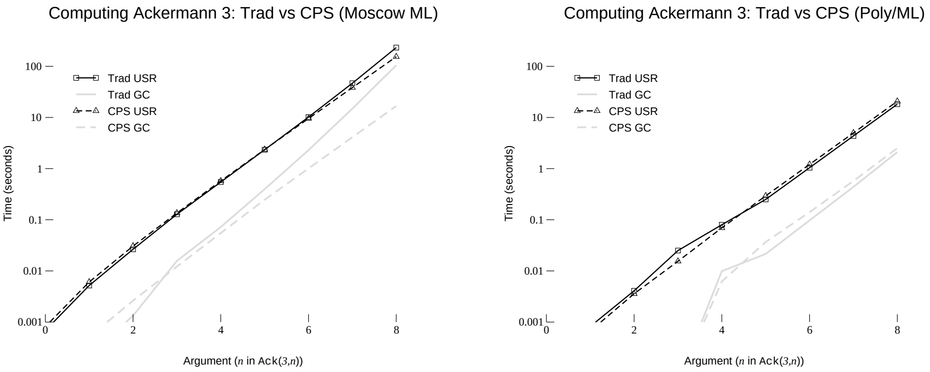


Fig. 13 Timing statistics for the Ackermann experiment. The data-points are means collected over 5 runs at each argument size. The mean times taken in the Moscow ML implementation at $n = 8$ were 232s for the traditional implementation, and 154s for the cps-conversion. For the Poly/ML implementation, the corresponding times were 18.1s and 20.7s, *i.e.*, the cps-conversion was slower.

The technology in the Barras's previously mentioned "call-by-value" rewriting engine (called `EVAL` in HOL4) provides an interesting comparison here. The `EVAL` function is generally quicker than rewriting. However, if one attempts to call `EVAL` on the last problem from the experiment (calculating $2^{11} - 3$), it causes Poly/ML to crash with what it claims is a "stack limit reached" error. When the same test is made in the Moscow ML implementation, it succeeds after 830s, which is over three times slower than the use of traditional conversions.

Experiment #2: Naïve NNF and Prenexing The rewriting done above to calculate Ackermann's function is somewhat atypical in that the term of interest grows larger as the rewriting proceeds. In interactive use, it is more typical to have the input term stay roughly the same size, or even grow smaller.

In order to test the CPS-system on a rewriting problem that is more typical, we tried a formula normalisation problem. At the atomic level, we combine the action of the following rewrites:

$$\begin{array}{ccccccccc}
\neg(p \land q) & = & \neg p \lor \neg q & \neg(p \lor q) & = & \neg p \land \neg q & \neg\neg p & = & p \\
(\exists x.P) & = & P^{\dagger} & (\forall x.P) & = & P^{\dagger} & & (\dagger :\ x \notin P) & \\
\neg(\exists x.P) & = & \forall x.\neg P & \neg(\forall x.P) & = & \exists x.\neg P & & & \\
(\exists x.P) \land Q & = & \exists x.P \land Q & P \land (\exists x.Q) & = & \exists x.P \land Q & & & \\
(\forall x.P) \land (\forall x.Q) & = & \forall x.P \land Q & (\forall x.P) \land Q & = & \forall x.P \land Q & P \land (\forall x.Q) & = & \forall x.P \land Q \\
(\exists x.P) \lor (\exists x.Q) & = & \exists x.P \lor Q & (\exists x.P) \lor Q & = & \exists x.P \lor Q & P \lor (\exists x.Q) & = & \exists x.P \lor Q \\
(\forall x.P) \lor Q & = & \forall x.P \lor Q & P \lor (\forall x.Q) & = & \forall x.P \lor Q & & &
\end{array}$$

The rewrites are attempted in the order given. There is no attempt made to avoid unnecessary quantifier-alternation, and by combining quantifier movement to the head of the formula with pushing negations inwards, repeated traversals of the input term are guaranteed.

The above rewrites were turned into a list of conversions `norm_convs`. The traditional conversion tested was

```
TOP_DEPTH_CONV (FIRST_CONV norm_convs)
```

where `FIRST_CONV` is the standard auxiliary defined as

```
fun FIRST_CONV [] tm = NO_CONV tm
  | FIRST_CONV (c::rst) tm =
      c tm handle (HOL_ERR _) => FIRST_CONV rst tm
```

The CPS-system version of `FIRST_CONV` is

```
fun kFIRST_CONV clist k f t =
  case clist of
    [] => kNO_CONV k f t
  | c::cs => (lift c kORELSEC kFIRST_CONV cs) k f t
```

This auxiliary was defined just for the purpose of the experiment, and takes as an argument a list of traditional conversions rather than a list of cps-conversions. This allows the code run as the CPS part of the experiment to be

```
drop (kTOP_DEPTH_CONV (kFIRST_CONV norm_convs))
```

For each term size, a random binary tree was generated (with the algorithm due to Martin and Orr [10]), and nodes randomly annotated with negations and quantifiers. The largest term had size 3319. The results are presented graphically in Figure 14. Again, the CPS-system is a clear improvement when running in Moscow ML. Under Poly/ML, the overall improvement is barely discernible, though the superiority in garbage collection times is even visible in the log-scaled graph.

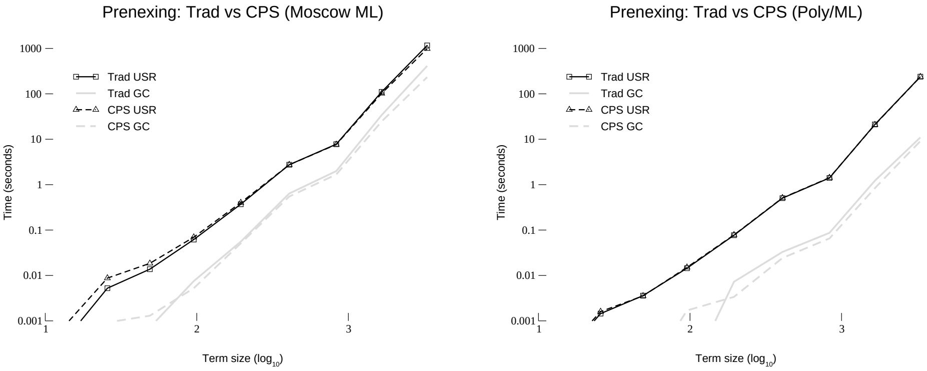


Fig. 14 Timing statistics for the prenexing experiment. The data-points are means collected over 10 runs on the same term of the given size. The mean times taken in the Moscow ML implementation on the largest formula were 1165s for the traditional implementation, and 986s for the cps-conversion. For the Poly/ML implementation, the corresponding times were 238s and 237s, *i.e.*, the cps-conversion was only very slightly faster.

5 Advanced Features

The last features to be implemented in the CPS-system are conditional rewriting, and user-supplied congruences. These features are described in Paulson [11], but the implementations are only sketched, and were clearly the subject of intensive work at the time.

Both features require the rewriter to be called recursively. In the case of conditional rewriting, this recursive call happens at what might be thought of as the “leaf-level” of the rewriting, when calling the analogue of `REWR_CONV`.

Each feature requires rather more code to implement than do the simple conversions in the earlier parts of this paper. For this reason, the implementations are not given completely. Again, full source code is available from the URL given in Section 6.2.

5.1 Conditional Rewriting

Conditional rewriting is the ability to perform a rewrite which only holds if some pre-condition is discharged. For example, conditional rewrites are common when working with integer division and modulus, where one uses theorems like

$$\begin{gathered}
\vdash r < n \quad \Rightarrow \quad (qn + r)\ \mathsf{MOD}\ n = r \\
\vdash 0 < n \quad \Rightarrow \quad (xn + r)\ \mathsf{DIV}\ n = x + r\ \mathsf{DIV}\ n \\
\vdash k < n \quad \Rightarrow \quad k\ \mathsf{MOD}\ n = k \\
\vdash 0 < n \wedge 1 < d \quad \Rightarrow \quad (n\ \mathsf{DIV}\ d < n) = \top
\end{gathered}$$

If a user specifies that a rewrite should be done with a theorem of this form, the rewriting system needs to be able to show that the precondition is true before it replaces the l.h.s. with the r.h.s. from the rewrite theorem. Assuming that the appropriate tool for discharging these preconditions is the rewriting engine itself, this then requires a recursive call.

There is a risk of exponential blow-up here: a pre-condition usually involves sub-terms of the original, so that if simplification of the pre-condition doesn’t succeed, and if the simplifier then goes onto look at that sub-term again as part of a top-down traversal, it will be simplified twice. In a bottom-up traversal, sub-terms may again be traversed twice, but the second traversal (in the pre-condition solver) will at least be of terms that have already been normalised.

In order to sit well with the CPS style of the cps-conversions, these recursive calls can be recast as calls to continuations. Because of the generality of the `Conv` constructor (from Section 3), this is quite straightforward.

The only nicety in the following is a limit on the number of times the rewriter will let itself be called to decide a conditional rewrite's pre-condition. With pathological rewrites (of the sort that users throw at rewriting systems as a matter of course), it is all too easy for a system to go into an infinite loop chaining backwards on unprovable pre-conditions. In practice, it appears that this limit can be quite low. Here, the limit is chosen to be three.

```
fun kCondRewr th rewriter n k fl t =
  if n > 3 then apply_fail fl stackdeep t
  else if is_imp (concl th) then
    case total (PART_MATCH (lhs o #2 o dest_imp) th) t of
      ERR e => apply_fail fl e t
    | OK rwt => let
        val (cond, _) = dest_imp (concl rwt)
        fun kont (TM _) = apply_fail fl cond_unproven t
          | kont (THM th) =
              case total EQT_ELIM th of
                ERR _ => apply_fail fl cond_unproven t
              | OK th’ => apply_cont k (MP rwt th’)
      in
        rewriter (n + 1) (Conv kont) (RT(t,fl)) cond
      end
  else
    case total (PART_MATCH lhs th) t of
      ERR e => apply_fail fl e t
    | OK rwt => apply_cont k rwt
```

Fig. 15 The conditional rewriting primitive as a cps-conversion. The `PART_MATCH` function attempts to match part of a theorem (here its l.h.s.) to a given term, and returns the appropriately instantiated theorem. The `EQT_ELIM` function attempts to turn a theorem of the form $\vdash x = \top$ into $\vdash x$. The function `MP` implements *modus ponens*. The code also assumes predefined exception values `stackdeep` and `cond_unproven` to signal that pre-condition solving has recursed too far, and that a pre-condition couldn't be proven, respectively.

The code for `kCondRewr` is given in Figure 15. The first check is to see whether or not the limit on pre-condition attempts has been reached. Then the code tests to see whether or not the provided rewrite is conditional or not. If not, the code falls through to code at the bottom of the definition that attempts to match and instantiate an unconditional rewrite. Finally, in the last case, if there is a match with the conditional rewrite, then the provided *`rewriter`* function is called on the rewrite's pre-condition (*`cond`*). The continuation `cont` checks to see if *`cond`* has been reduced to true, and if so, uses *modus ponens* to discharge the pre-condition, creating a result theorem that does indeed equate the original argument to some new r.h.s.

Putting the atomic `kCondRewr` into a rewriting system using `kTOP_DEPTH_CONV` or some other traversal strategy becomes slightly complicated when it comes to tying the various recursive knots. However, using the existing cps-conversion combinators, the code mimics that used in the existing HOL4 simplifier, and the fact that a CPS style is being used is concealed from the programmer.

5.2 Rewriting with User-supplied Congruences

A congruence is a theorem capturing how equalities on sub-terms can be used to construct an equality on a larger term. For example, the inference rule behind `MK_COMB` can be captured by the theorem

$$\vdash (f = g) \Rightarrow ((x = y) \Rightarrow (f\,x = g\,y))$$

In general, a congruence will be

$$\vdash \mathit{subeqn}_1 \Rightarrow (\cdots \Rightarrow (\mathit{subeqn}_n \Rightarrow \mathit{eqn}))$$

where the equation in eqn will match terms of a particular form and the various subeqn_i specify that certain sub-terms should be rewritten. For example, the theorem

$$\vdash g = g' \Rightarrow (\textsf{if } g \textsf{ then } t \textsf{ else } e) = (\textsf{if } g' \textsf{ then } t \textsf{ else } e)$$

is a congruence that causes only the guard of an if-then-else form to be rewritten.

One important refinement of this idea is to support the inclusion of additional context when a sub-term is examined. The idea is to allow the various subeqn_i above to have the form $P \Rightarrow (t_0 = t)$. The P represents additional context that may be assumed in the course of simplifying t_0. For example, one can use the following as a congruence for simplifying conjunctions:

$$\vdash (P = P') \Rightarrow ((P' \Rightarrow (Q = Q')) \Rightarrow (P \wedge Q = P' \wedge Q'))$$

This uses the idea that it is valid to assume the truth of P when simplifying Q. More specifically, this rule encodes the rewriting strategy of first simplifying P to P', and then assuming P' while simplifying Q to Q'.

A naïve cps-conversion version of a simplifier that uses congruence rules in this vein can be written in 50 lines of code. The handling of congruence rules is done by a function taking a theorem representing the congruence rule as an argument. If the rule has been reduced to a bare equality, this is the result. Otherwise, the implication is split, and the rewriter called recursively on the implication's first argument. The continuation passed to the recursive call is to use *modus ponens* on the rule and the result, generating a smaller congruence rule, and then repeating.

If the implication's first argument is itself an implication, then this implication's first argument can be assumed going into the recursive call. In the continuation, the assumption is discharged from the result theorem, producing something of the right form for the call to *modus ponens*.

Note that using a congruence rule makes it harder to detect when a sub-term traversal has left its argument unchanged. Currently, both the HOL4 and Isabelle simplifiers make explicit calls to an equality check after a congruence has been applied so that they can then signal unchangedness appropriately. Though this may seem like a source of inefficiency, the way in which congruence rules are instantiated should result in the call to `aconv` quickly hitting sub-terms that are pointer-equivalent.

6 Conclusion

In the conclusion of [11], Paulson writes

> It is hard to improve the efficiency in the LCF framework. Any simplifier must produce a theorem to justify its result. It must coexist with other theorem-proving tools, and with ML.

This paper demonstrates that while requiring non-trivial effort, some efficiency gains in LCF-style rewriting are indeed possible. These gains are achieved by considering the structure of the call graph brought into being as rewriting proceeds, and more generally, by worrying about rewriting's memory allocations.

Switching to CPS-conversions allows exactly the same sequence of calls to "atomic" rewriting steps (*e.g.*, calls to functions like REWR_CONV and BETA_CONV), while combining the results of these calls more efficiently. If existing code performed its rewriting steps as a result of the application of a particular term-traversal strategy, the new CPS-system provides a directly analogous API so that the same strategy can be used, and the same rewriting steps produced.

The improvements in performance are incontrovertible, but hard to observe when rewriting small terms, or when rewriting terminates quickly. Of course, these are the scenarios usually encountered when using the LCF-style systems interactively. However, the beauty of these programmable systems is that they can be the basis for the construction of non-interactive tools. In these situations, long-running rewriting efforts can and do occur. This was certainly the author's experience when implementing the tool behind the work described in [3].

Boulton [4] also found it hard to make HOL's rewriting go faster. In addition to investigating the UNCHANGED optimisation (which *is* a clear advance compared to not using it), he looked at deferring all theorem-proving work within a rewriting application until the end of the process. This then enables optimisations such as reordering rewrites so that all changes in one sub-term can be done together, minimising the number of term-traversals done with MK_COMB.

Using Boulton's general framework it is certainly possible to avoid building an explicit chain of calls to TRANS in a call-stack. Instead, one of the optimisation he considered constructs a large representation of all these calls in memory, and then examines them for optimisation opportunities later. This "lazy" approach to the handling of TRANS (and other inference rules) is thus quite the opposite tactic to this paper's aim of eagerly evaluating TRANS as soon as possible.

6.1 Future Work

There are a number of areas that still remain to be investigated. One is the behaviour of Poly/ML, which can be quite odd on large terms. In some situations, though it is running 5–10 times faster than Moscow ML, it is consuming 2–3 times as much memory, and garbage collections can cause a rewriting application that normally takes roughly a second to take 10 times as long. Independently, we observed Poly/ML crash on a large problem (evaluating Ack 3 8 with Barras's EVAL) that Moscow ML has no problem with. I hope that when these issues, whether they be in our port of HOL4 or in the Poly/ML implementation, are resolved, future experiments will provide yet more vindication of the cps-conversion approach.

The memory behaviour of rewriting discussed in Section 2.6 is also intriguing. If `t` is the term $\mathtt{SUC}^{(n+1)}\ 0$, then rewriting with the theorem $\vdash \mathtt{SUC}\ 0 = 1$ results in a theorem

$$\vdash \mathtt{SUC}^{(n+1)}\ 0 = \mathtt{SUC}^{n}\ 1$$

In general, one would hope that applying a conversion to a term `t` should result in a theorem $\vdash$ `t` = `t'` where the `t` in the theorem should be the same structure in memory as the `t` that was passed to the conversion. But in fact, in the example given, the `t` in the theorem is a "deep" copy of the input, sharing no structure with the original term `t`. This profligacy with memory surely represents a run-time cost. Approaches to this problem might include the alternative version of `MK_COMB` discussed in Section 2.6, or (more radically) hash-consing in the kernel.

For the moment, the advantages offered by cps-conversions do not seem great enough to warrant reworking the standard HOL4 simplifier to use this approach. In particular, there is a great deal of code in use which relies on conversions being of type `term -> thm`. Changing extant code to instead use the more complex continuation-based type is unappealing in general. Of course, in specialised settings, where the developer knows that rewriting will be running for long periods, cps-conversions may indeed be appropriate. This paper demonstrates that cps-conversions support a familiar set of combinators, and that they inter-convert with normal Paulson-style conversions straightforwardly.

The same analysis of memory consumption and call-patterns that led to the development of cps-conversions also led to the recognition that `REPEATC` is not implemented ideally at the moment. The change to use `IFC` instead of `ORELSEC` is one that will appear in a future release of HOL4.

6.2 Availability

The source code for the various forms of cps-conversion, the source code for the experiments performed, and the data generated in those experiments are all available in a compressed tar-file from

`http://users.rsise.anu.edu.au/~michaeln/data/cpsconvs.tgz`.

Acknowledgements NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program. Joe Hurd and Rob Arthan provided useful early discussion and comments on this work. The anonymous referees suggested a much-needed reorganisation of the material's presentation.

References

1. Bruno Barras. Programming and computing in HOL. In Mark Aagard and John Harrison, editors, *Theorem Proving in Higher Order Logics, 13th International Conference, TPHOLs 2000*, volume 1869 of *Lecture Notes in Computer Science*, pages 17–37. Springer, August 2000.
2. Nick Benton and Andrew Kennedy. Exceptional syntax. *Journal of Functional Programming*, 11(4):395–410, July 2001.
3. Steve Bishop, Matthew Fairbairn, Michael Norrish, Peter Sewell, Michael Smith, and Keith Wansbrough. Engineering with logic: HOL specification and symbolic-evaluation testing for TCP implementations. In *POPL'06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 55–66, New York, NY, USA, 2006. ACM Press.
4. R. J. Boulton. Transparent optimisation of rewriting combinators. *Journal of Functional Programming*, 9(2):113–146, March 1999.

5. Jacek Chrząszcz and Daria Walukiewicz-Chrząszcz. Towards rewriting in Coq. In Huber Comon-Lundh, Claude Kirchner, and Hélène Kirchner, editors, *Rewriting, Computation and Proof: Essays Dedicated to Jean-Pierre Jouannaud on the Occasion of His 60th Birthday*, volume 4600 of *Lecture Notes in Computer Science*, pages 113–131. Springer, 2007.
6. M. J. C. Gordon and T. Melham, editors. *Introduction to HOL: a theorem proving environment for higher order logic*. Cambridge University Press, 1993.
7. John Harrison. HOL Light: a tutorial introduction. In Mandayam Srivas and Albert Camilleri, editors, *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design (FMCAD'96)*, volume 1166 of *Lecture Notes in Computer Science*, pages 265–269. Springer-Verlag, 1996.
8. HOL website. `http://hol.sourceforge.net`.
9. R. B. Jones. ICL ProofPower. *BCS-FACS FACTS*, Series III, 1(1):10–13, Winter 1992.
10. Harold W. Martin and Bonnie J. Orr. A random binary tree generator. In *CSC '89: Proceedings of the 17th Annual Computer Science Conference*, pages 33–38, New York, NY, USA, 1989. ACM.
11. Lawrence Paulson. A higher-order implementation of rewriting. *Science of Computer Programming*, 3(2):119–149, August 1983.
12. Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1994.