J. Functional Programming **9** (2): 113–146, March 1999. Printed in the United Kingdom

Transparent optimisation of rewriting combinators

RICHARD J. BOULTON*
Division of Informatics, University of Edinburgh,
80 South Bridge, Edinburgh EH1 1HN, Scotland

Abstract

The LCF system was the first mechanical theorem prover to be user-programmable via a metalanguage, ML, from which the functional programming language Standard ML has been developed. Paulson has demonstrated how a modular rewriting engine can be implemented in LCF. This provides both clarity and flexibility. This paper shows that the same modular approach (using higher-order functions) allows transparent optimisation of the rewriting engine; performance can be improved while few, if any, changes are required to code written using these functions. The techniques described have been implemented in the HOL system, a descendant of LCF, and some are now in daily use. Comparative results are given. Some of the techniques described, in particular ones to avoid processing parts of a data structure that do not need to be changed, may be of more general use in functional programming and beyond.

1 Introduction

A theorem prover is a computer program that automatically proves theorems in some logic or assists a human in doing the same. A fully expansive theorem prover is one that generates proofs entirely in terms of primitive inferences of the logic; no metatheoretic results are used and no 'short-cuts' are taken in the implementation. If a theorem prover is not fully expansive it will be referred to as being *partially expansive*.

LCF is a fully expansive interactive theorem prover developed at the University of Edinburgh in the 1970s (Gordon *et al.*, 1979). Amongst its distinguishing features is the use of a general-purpose programming language as a metalanguage in which new proof procedures can be written. Strong typing in the language is used to force all proof procedures to be fully expansive. This is critical if non-logicians are to be allowed to write their own proof procedures without jeopardising the logical soundness of the system.

User programmability has proved to be a very popular feature of LCF and its descendants. It allows new derived inference rules to be written for the logic, making

* This work was funded by the Engineering and Physical Sciences Research Council (formerly the Science and Engineering Research Council) of Great Britain and the bulk of it was done while the author was at the University of Cambridge Computer Laboratory.

interactive theorem proving less tedious. Moreover, it has allowed other formalisms and languages to be embedded in the logic via a formal semantics. Specialised proof rules can be implemented for these formalisms.

The major drawback with forcing proofs to be fully expansive is that generation of these proofs can be far more costly in computational resources than implementations that take short-cuts. However, some means of ensuring soundness is essential if users are to be allowed to write their own procedures. An alternative to full expansion is formal verification of the correctness (soundness) of the new proof procedures. This can be very time consuming for the user and requires additional skills. Though there have been investigations into this approach (Davis and Schwartz, 1979; Boyer and Moore, 1981; Allen *et al.*, 1990; Slind, 1992), it has not yet been shown to be practical. Many users would not write their own proof procedures if they had to verify them or if they could not be certain that the system will prevent them from doing anything unsound.

So, at least for the moment, there are good reasons for investigating possible optimisations to the LCF approach. The author has proposed a hybrid approach (Boulton, 1993) similar to the notion of lazy evaluation. This is applicable to general reasoning. That paper also considers an optimisation technique that is restricted to equational reasoning. In the current paper the technique is separated from the use of laziness and developed further. Optimisation of equational reasoning is significant because such reasoning is ubiquitous. Equational reasoning manifests itself most commonly as rewriting, but it is also involved in many other theorem proving operations.

In 1983, Paulson described an implementation of a modular rewriter in LCF. This was implemented using the higher-order functions of LCF's metalanguage. Components of the rewriter were composed using these higher-order functions (*combinators*). This modularity made the code easy to follow and gave users the flexibility to construct their own rewriters. Paulson's code is still in use today. The main result of this paper is that the modularity also allows transparent optimisation, i.e. optimisation without the higher-level code having to be changed.

The remainder of this section is an overview of fully expansive theorem proving, the HOL system, and equational reasoning. Section 2 describes Paulson's *conversions*. Section 3 presents a rewriting example that illustrates the inefficiencies of conversion-based rewriting. The remaining sections develop optimisations for conversions, present some results (section 8), discuss related work (section 9), and draw conclusions (section 10). The presentation assumes that the reader is familiar with functional programming and the basic features of the Standard ML programming language.

1.1 Fully expansive theorem provers

Ideally, the primitive inferences of (some formulation of) a logic are the minimal set of rules required to provide the deductive power expected of the logic. In practice, some of the rules taken as primitive may be derivable from others, i.e. the set of rules is not minimal. This may simply be for convenience or because no-one has seen

https://doi.org/10.1017/S0956796899003391 Published online by Cambridge University Press

the derivation. However, to call high-level rules 'primitive' is probably undesirable because to do so would blur the distinction between primitive and derived rules. Wong (1995) uses the term *basic inference rules* to refer collectively to the real primitive rules and the derived rules taken as primitive. As a general principle, the primitive rules should be simple and talk about only the very basic constructs of the logic.

A typical primitive rule is *modus ponens*:

$$\frac{\Gamma_1 \vdash t_1 \qquad \Gamma_2 \vdash t_1 \Rightarrow t_2}{\Gamma_1 \cup \Gamma_2 \vdash t_2}$$

This states that if a term t_1 holds under assumptions Γ_1 and that under assumptions Γ_2, t_1 implies t_2, then t_2 holds under the set-theoretic union of Γ_1 and Γ_2.

A proof produced by a fully expansive theorem prover can be viewed as a tree with axioms and pre-proved theorems as leaf nodes, and applications of primitive inference rules as internal nodes. A partially expansive theorem prover may replace certain subtrees of the proof by applications of metatheorems. Metatheorems are properties that cannot be expressed directly in the logic but are believed to be valid, e.g. "Two conjunctions are equivalent if the sets of conjuncts are equal". An instance of this metatheorem is:

$$(x \wedge y) \wedge z = y \wedge (x \wedge (z \wedge x))$$

Generating a proof entirely from primitive inferences provides security since, by use of suitable implementation techniques, the critical code of the theorem prover can be limited to the implementation of the primitive inference rules. The major drawback of the approach is that fully expansive theorem provers tend to be slow in comparison to systems that exploit metatheorems or implement derived rules of the logic as primitives. This paper describes work to improve the efficiency of a fully expansive theorem prover while avoiding major changes to the code. A primary aim was to retain full expansion to primitive inferences, so the significant decrease in computational complexity that might be achieved by use of metatheorems has not been obtained.

1.2 The HOL system

The HOL system (Gordon and Melham, 1993) is a direct descendant of LCF, used at many academic and industrial sites around the world for work in formal methods. The main difference between HOL and LCF is in the logic supported. LCF was designed to mechanise a polymorphic version of Scott's 'Logic for Computable Functions', whereas HOL supports a version of classical higher-order logic.

The higher-order logic supported by the HOL system is a version of Church's simple theory of types (Church, 1940), adapted to allow type variables in the logic (polymorphism). Every term in the logic has a unique type, though the type may be polymorphic (contain variables). Higher-order functions are allowed. These are functions that take other functions as arguments or return a function as their result. In particular, it is possible to quantify over functions.

https://doi.org/10.1017/S0956796899003391 Published online by Cambridge University Press

The metalanguage of LCF is called ML. It is a strongly typed functional programming language that uses eager evaluation, i.e. all arguments to a function are evaluated before the function application itself is evaluated. The Standard ML programming language (Milner *et al.*, 1990) is derived from it. Like LCF, HOL and ML were originally implemented in Lisp. A more recent version of the HOL system (Slind, 1991) is written in Standard ML and also uses it directly as its metalanguage. The Standard ML syntax will be used in this paper. Standard ML is an impure functional language: it has imperative features such as mutable data and a sequencing operation.

The terms of higher-order logic have types. A type in the HOL system is either a type variable denoted by a name beginning with a prime (`'`), or an application of a constructor to zero or more types. The number of arguments taken by a constructor is known as its *arity*. The basic types such as `bool` (Booleans) and `num` (Peano natural numbers) are constructors of arity zero. The basic types, cartesian-product types and function types are given special treatment by the pretty-printer, e.g.:

`()bool`	is printed as	`bool`
`(()bool, ()num)prod`	is printed as	`bool * num`
`('a, 'a)fun`	is printed as	`'a -> 'a`

The last example denotes a function from some type to itself. The variable ‘`'a`’ can be instantiated to any type. In this paper, HOL types are written in quotation marks with a leading colon, e.g. ‘‘`:bool`’’. HOL terms may also be quoted but without a colon.

A HOL term can be a constant, a variable, a function application, or a λ-abstraction:

$$t ::= c \mid v \mid t\ t' \mid \lambda v.\ t$$

Constants and variables are atomic. They consist of a name and a type. Function applications (also called *combinations*) consist of two terms, one a function and the other an argument. An abstraction consists of a variable (the bound variable) and another term (the body). It is assumed that the reader is familiar with these concepts from the λ-calculus. So, a term is a tree structure in which each of the internal nodes is either a combination or an abstraction. Each leaf node is either a constant or a variable. The structure of these trees plays a vital rôle in the optimisation of equational reasoning.

It is assumed that the reader has some basic familiarity with set theory and logic. The symbols $\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$, $\forall$, and $\exists$ are used for negation (‘not’), conjunction (‘and’), disjunction (‘or’), implication, if-and-only-if, universal quantification (‘for all’), and existential quantification (‘there exists’), respectively. In HOL, the notation ‘‘$\forall x.\ b$’’ is actually an abbreviation for the application of a function ‘$\forall$’ to a λ-abstraction: ‘‘$\forall\ (\lambda x.\ b)$’’, and similarly for ‘$\exists$’. Some function constants are displayed as infixes, e.g., ‘‘$x \wedge y$’’ is syntactic sugaring for the term ‘‘$(\wedge\ x)\ y$’’.

The logical types, terms (`term`) and theorems (`thm`) of HOL are implemented as data types in ML. Rules of inference are functions over theorems. Terms are parsed and printed as quotations, e.g. ‘‘$n+1$’’. Theorems in the HOL logic are *sequents*. A sequent is a pair with a set of formulas (the hypotheses) as the first component and

a single formula (the conclusion) as the second component. In HOL, a formula is simply a Boolean-valued term. A theorem asserts that the conclusion of the sequent is a consequence of the hypotheses.

Terms are built up by applying constructor functions to the subterms. ML records are used for the arguments. For example, an application is constructed by applying the function `mk_comb` to a record of two components, one for the operator and one for the operand. The components are labelled '`Rator`' and '`Rand`' respectively. Thus, a call to construct an application term has the following form:

```
mk_comb {Rator = ...,Rand = ...}
```

λ-abstractions are constructed in a similar way. There are also corresponding destructor functions. For these, ML pattern matching is used to extract the subterms from the record, e.g.:

```
val {Rator,Rand} = dest_comb ...
```

Here the '`Rator`' and '`Rand`' act both as record labels and as the names of variables to be bound to the subterms.

Theorems are an abstract type. The representation type consists of a list of terms for the hypotheses paired with a term for the conclusion. The identifiers exported from the abstract type are the axioms and the primitive inference rules. It is this abstract type together with the strong typing of ML that ensures that only valid conjectures can become theorems. Theorems are printed as sequents: The hypotheses are printed separated by commas, followed by a turnstile, and then the conclusion, e.g., $m < n,\ n < p \vdash m < p$.

Proofs in the HOL system are typically conducted using the subgoal package. This allows a conjecture (a *goal*) to be broken up into simpler subgoals using functions called *tactics*. The system keeps track of the subgoals, and the tactics contain inference rules that justify the original goal as a theorem given theorems for the subgoals. Tactics may be combined using higher-order functions called *tacticals*.

1.3 Equational reasoning

1.3.1 Equivalence relations

A *binary relation* R on sets X and Y is a subset of the cartesian product $X \times Y$. For $x \in X$ and $y \in Y$, the statement xRy is taken to be true if the pair (x, y) is an element of R, and is false otherwise. Relations on three or more sets can be defined in a similar way, but the discussion here will be restricted to binary relations. The above definition of a binary relation is the set-theoretic view. In HOL it is usual to consider a binary relation to be a function of two arguments which returns a Boolean value. Instead of sets, the relation is defined for HOL types.

An *equivalence relation* R is a binary relation for which both arguments are taken from the same set (or type) and that satisfies the following properties:

Reflexivity	$\forall x.\ xRx$
Symmetry	$\forall x\ y.\ xRy \Rightarrow yRx$

https://doi.org/10.1017/S0956796899003391 Published online by Cambridge University Press

$$\text{Transitivity} \qquad \forall x\ y\ z.\ xRy \wedge yRz \Rightarrow xRz$$

The higher-order logic implemented by the HOL system has a built-in equivalence relation, denoted by '='. Actually, it is a function that represents the relation by mapping pairs of values to true if they are in the relation or to false otherwise. It is polymorphic and takes its arguments separately rather than as a pair. Thus, its most general type is ``:'a -> ('a -> bool)``.

1.3.2 Congruences

One normally expects an 'equality' to be more than just an equivalence relation. If two expressions are 'equal', one expects to be able to substitute one for the other when it appears as a subterm of a term. The equality in HOL has just such a property, expressed by the following *congruence* rules:

$$\frac{\Gamma \vdash x = y}{\Gamma \vdash f\ x = f\ y}\ (\texttt{AP\_TERM}) \qquad \frac{\Gamma \vdash f = g}{\Gamma \vdash f\ x = g\ x}\ (\texttt{AP\_THM})$$

$$\frac{\Gamma \vdash t_1 = t_2}{\Gamma \vdash (\lambda x.\ t_1) = (\lambda x.\ t_2)}\ x \text{ is not free in } \Gamma\ (\texttt{ABS})$$

1.3.3 Rewriting

Informally, an *equation* is a term consisting of the application of an equality to two terms of the same type. In the context of the HOL system it is useful to consider an equation to be a theorem of the form:

$$\vdash l\ E\ r$$

where E is a congruence relation. Such a theorem can be used as a *rewrite rule*:

$$l \longrightarrow r$$

by matching the left-hand side of the equation against a term and instantiating the right-hand side appropriately. For example, the theorem:

$$\vdash x + y = y + x$$

can be matched against the term ``$(a * b) + (c + d)$`` so that x is instantiated with the term ``$a * b$`` and y is instantiated with ``$c + d$``. The free variables of the theorem are implicitly universally quantified, so the theorem is equivalent to:

$$\vdash \forall x\ y.\ x + y = y + x$$

Specialising the bound variables as determined by the matching, the following theorem is obtained:

$$\vdash (a * b) + (c + d) = (c + d) + (a * b)$$

This process is known as *rewriting*. When congruence rules are available the rewriting can be done *at depth*. For example, specialising the free variables of the

https://doi.org/10.1017/S0956796899003391 Published online by Cambridge University Press

rewrite rule with ‘‘c‘‘ and ‘‘d‘‘ yields the theorem:

$$\vdash\ c + d = d + c$$

Using the congruence rule for the operand of a function application (`AP_TERM`) it is possible to obtain:

$$\vdash\ (a * b) + (c + d) = (a * b) + (d + c)$$

Thus, in effect, the term ‘‘$(a * b) + (c + d)$‘‘ has been rewritten by applying the rewrite rule to the second argument of the top-level sum.

For an in-depth presentation of rewriting, see the book by Baader and Nipkow (1998).

2 Conversions

Equational reasoning in HOL is implemented using ML functions called *conversions* as introduced by Paulson (1983). Paulson's conversions have the ML type:

```
term -> thm
```

which is abbreviated as `conv`. Conversions also have the property that for an argument t, they return a theorem of the form:

$$\vdash\ t = t'$$

That is, the theorem is an equation, and the left-hand side is the argument term. Paulson describes some basic conversions and some functions for combining these to form new conversions, thus allowing sophisticated rewriting strategies to be developed.

Figure 1 gives some of the code for the HOL implementation of conversions. The functions given can be used to compose basic conversions into more sophisticated ones. Examples of basic conversions are application of a rewrite rule to a term, and beta-reduction. The identity conversion, `ALL_CONV`, leaves a term unchanged; it is defined to be the reflexivity rule, `REFL`. The conversion `NO_CONV` always fails. The infix functions `THENC` and `ORELSEC` are used for sequencing and alternating conversions respectively. `THENC` takes two conversions and produces a conversion that applies them in succession. The results are combined using the transitivity rule, `TRANS`. `ORELSEC` applies the first conversion, but if this fails the second is applied.

The functions `RATOR_CONV`, `RAND_CONV` and `ABS_CONV` are the ‘congruence’ conversions, that is they implement the congruence rules `AP_THM`, `AP_TERM` and `ABS` (section 1.3.2) as conversions. The function `RATOR_CONV` applies a conversion to the operator of a function application (combination), and `RAND_CONV` applies a conversion to the operand. Similarly, `ABS_CONV` applies a conversion to the body of an abstraction, but fails if the bound variable is free in the hypotheses of the theorem returned by the conversion. This failure is rare because it is unusual for conversions to return theorems with hypotheses.

As well as `RATOR_CONV` and `RAND_CONV` it is useful to have a function `COMB_CONV`, which applies a conversion to both the operator and operand of a function application. This can be defined in terms of `RATOR_CONV`, `RAND_CONV` and `THENC`, but it

https://doi.org/10.1017/S0956796899003391 Published online by Cambridge University Press

```
val ALL_CONV = REFL;

val NO_CONV : conv = fn _ => raise CONV_ERR "NO_CONV";

fun conv1 THENC conv2 : conv =
   fn tm => let val th1 = conv1 tm
                val th2 = conv2 (rhs (concl th1))
            in  TRANS th1 th2
            end;

fun conv1 ORELSEC conv2 : conv =
   fn tm => (conv1 tm handle _ => conv2 tm);

fun RATOR_CONV conv tm =
   let val {Rator,Rand} = dest_comb tm
                          handle _ => raise CONV_ERR "RATOR_CONV"
   in  AP_THM (conv Rator) Rand
   end;

fun RAND_CONV conv tm =
   let val {Rator,Rand} = dest_comb tm
                          handle _ => raise CONV_ERR "RAND_CONV"
   in  AP_TERM Rator (conv Rand)
   end;

fun ABS_CONV conv tm =
   let val {Bvar,Body} = dest_abs tm
                         handle _ => raise CONV_ERR "ABS_CONV"
       val Bodyth = conv Body
   in  (ABS Bvar Bodyth handle _ => raise CONV_ERR "ABS_CONV")
   end;
```

Fig. 1. Original conversions for HOL.

```
fun COMB_CONV conv tm =
   let val {Rator,Rand} = dest_comb tm
                          handle _ => raise CONV_ERR "COMB_CONV"
   in  MK_COMB (conv Rator, conv Rand)
   end;
```

Fig. 2. Function for applying a conversion to both operator and operand.

is more efficient to make it a basic conversion function. The definition is given in figure 2. The `MK_COMB` rule is a combination of the two congruence rules for function applications. It can be more efficient than separate calls to `AP_TERM` and `AP_THM`.

These basic conversions can be used to define term traversal strategies for rewriting (depth rewrites; see section 1.3.3). First it is convenient to define a function for applying a conversion to all the subterms of a term (figure 3). If the term is a function application, `COMB_CONV` is used. If it is an abstraction, the conversion is applied to the body. Otherwise, the term is left as it is.

https://doi.org/10.1017/S0956796899003391 Published online by Cambridge University Press

```
fun SUB_CONV conv tm =
   if (is_comb tm) then COMB_CONV conv tm
   else if (is_abs tm) then ABS_CONV conv tm
   else ALL_CONV tm;
```

Fig. 3. Function for applying a conversion to subterms.

```
fun ONCE_DEPTH_CONV conv tm =
   (conv1 ORELSEC (SUB_CONV2 (ONCE_DEPTH_CONV conv)) ORELSEC
   ALL_CONV) tm;

fun REPEATC conv tm = ((conv THENC REPEATC conv) ORELSEC ALL_CONV) tm;

fun TRY_CONV conv = conv ORELSEC ALL_CONV;

fun TOP_DEPTH_CONV conv tm =
   (REPEATC3 conv THENC4
    TRY_CONV5
       (CHANGED_CONV (SUB_CONV6 (TOP_DEPTH_CONV conv)) THENC7
        TRY_CONV8 (conv9 THENC10 TOP_DEPTH_CONV conv)))
   tm;
```

Fig. 4. Depth conversions.

The function ONCE_DEPTH_CONV (figure 4) implements a top-down traversal in which the conversion is applied at most once to any subterm. The function is defined recursively. It attempts to apply the conversion to the entire term. If this succeeds, nothing more is done. Otherwise, the function is applied recursively to the subterms. In the event that the recursive call fails, the term is left unchanged. The function TOP_DEPTH_CONV (figure 4) is similar but it attempts to apply the conversion as much as it can instead of just once on any subterm. It uses auxiliary functions including CHANGED_CONV which fails if the application of its argument does not change the term. The subscripts in the bodies of the functions are not part of the code. They are included for use in the following example.

3 An example using depth rewriting

Suppose a user wants to simplify the term:

$$\lambda n.\ (n * 0) + n$$

using the base cases of the normal recursive definitions of addition and multiplication over the natural numbers:

$$0 + x = x \tag{1}$$

$$0 * x = 0 \tag{2}$$

Unfortunately, the equation for multiplication has the '0' in the wrong argument position, so it is necessary to commute the arguments of '$*$' in the term using the

https://doi.org/10.1017/S0956796899003391 Published online by Cambridge University Press

```
ONCE_DEPTH_CONV (REWR_CONV <Equation 3>) THENC11
TOP_DEPTH_CONV (REWR_CONV <Equation 1> ORELSEC REWR_CONV <Equation 2>)
```

Fig. 5. Conversion to rewrite the example.

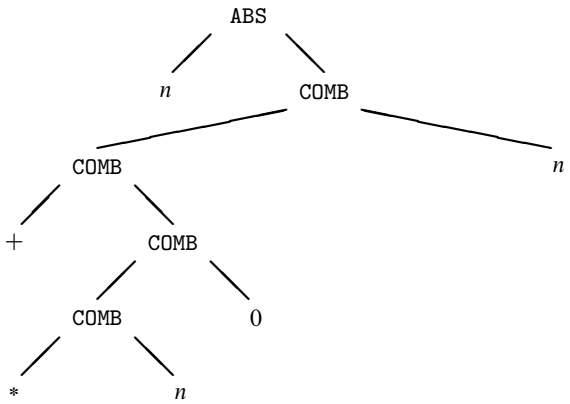

Fig. 6. The structure of the term ‘‘$\lambda n.\ (n * 0) + n$‘‘.

following rule:

$$x * y \quad = \quad y * x \tag{3}$$

The user could indicate explicitly where, within the term, rewrites are to take place, but even for a small term that is very tedious (unless a graphical user interface is available). It is therefore usual in HOL to use a function such as TOP_DEPTH_CONV that applies the specified rewrites everywhere it can within the term. However, TOP_DEPTH_CONV cannot be used with equations like (3) because it will loop due to the use of REPEATC. The usual solution to this is to use ONCE_DEPTH_CONV (actually the corresponding tactic) for the commutative law and then TOP_DEPTH_CONV for the other equations, as illustrated in figure 5.

The structure of the example term is given in figure 6. Table 1 lists the theorems generated when the conversion in figure 5 is applied. The table is in three parts. The first part corresponds to the application of ONCE_DEPTH_CONV and the second to the application of TOP_DEPTH_CONV. The last part is the application of the transitivity rule that produces an overall result from the first two parts. Transitivity rules are introduced by using THENC (section 2) to sequence conversions. The rightmost column of numbers refers to the subscripts in the figures. They indicate the position in the code at which the corresponding theorem is generated.

It should be clear from Table 1 that many unnecessary theorems are generated by the conversions, especially redundant combinations of reflexivity and transitivity. For example, Theorems 24–27 are unnecessary; Theorem 28 could be generated directly from Theorem 23 using the AP_THM inference rule (see section 1.3.2). The problem is even more acute when the term being rewritten contains large subterms that are not changed by the rewriting. Instead of being dealt with by a single

https://doi.org/10.1017/S0956796899003391 Published online by Cambridge University Press

Table 1. *Theorems generated for conventional rewriting*

1	$\vdash + = +$	Reflexivity	at	2
2	$\vdash n * 0 = 0 * n$	Rewriting with Equation 3	at	1
3	$\vdash (+ (n * 0)) = (+ (0 * n))$	`MK_COMB` on 1 and 2	at	2
4	$\vdash n = n$	Reflexivity	at	2
5	$\vdash (n * 0) + n = (0 * n) + n$	`MK_COMB` on 3 and 4	at	2
6	$\vdash \lambda n.\ (n * 0) + n = \lambda n.\ (0 * n) + n$	`ABS` on 5	at	2
7	$\vdash \lambda n.\ (0 * n) + n = \lambda n.\ (0 * n) + n$	Reflexivity	at	3
8	$\vdash (0 * n) + n = (0 * n) + n$	Reflexivity	at	3
9	$\vdash (+ (0 * n)) = (+ (0 * n))$	Reflexivity	at	3
10	$\vdash + = +$	Reflexivity	at	3
11	$\vdash + = +$	Reflexivity	at	6
	`CHANGED_CONV` fails			
12	$\vdash + = +$	Reflexivity	at	5
13	$\vdash + = +$	Transitivity on 10 and 12	at	4
14	$\vdash 0 * n = 0$	Rewriting with Equation 2	at	3
15	$\vdash 0 = 0$	Reflexivity	at	3
16	$\vdash 0 * n = 0$	Transitivity on 14 and 15	at	3
17	$\vdash 0 = 0$	Reflexivity	at	6
	`CHANGED_CONV` fails			
18	$\vdash 0 = 0$	Reflexivity	at	5
19	$\vdash 0 * n = 0$	Transitivity on 16 and 18	at	4
20	$\vdash (+ (0 * n)) = (+ 0)$	`MK_COMB` on 13 and 19	at	6
	`CHANGED_CONV` *succeeds*			
21	$\vdash (+ 0) = (+ 0)$	Reflexivity	at	8
22	$\vdash (+ (0 * n)) = (+ 0)$	Transitivity on 20 and 21	at	7
23	$\vdash (+ (0 * n)) = (+ 0)$	Transitivity on 9 and 22	at	4
24	$\vdash n = n$	Reflexivity	at	3
25	$\vdash n = n$	Reflexivity	at	6
	`CHANGED_CONV` fails			
26	$\vdash n = n$	Reflexivity	at	5
27	$\vdash n = n$	Transitivity on 24 and 26	at	4
28	$\vdash (0 * n) + n = 0 + n$	`MK_COMB` on 23 and 27	at	6
	`CHANGED_CONV` *succeeds*			
29	$\vdash 0 + n = n$	Rewriting with Equation 1	at	9
30	$\vdash n = n$	Reflexivity	at	3
31	$\vdash n = n$	Reflexivity	at	6
	`CHANGED_CONV` fails			
32	$\vdash n = n$	Reflexivity	at	5
33	$\vdash n = n$	Transitivity on 30 and 32	at	4
34	$\vdash 0 + n = n$	Transitivity on 29 and 33	at	10
35	$\vdash (0 * n) + n = n$	Transitivity on 28 and 34	at	7
36	$\vdash (0 * n) + n = n$	Transitivity on 8 and 35	at	4
37	$\vdash \lambda n.\ (0 * n) + n = \lambda n.\ n$	`ABS` on 36	at	6
	`CHANGED_CONV` *succeeds*			
38	$\vdash \lambda n.\ n = \lambda n.\ n$	Reflexivity	at	8
39	$\vdash \lambda n.\ (0 * n) + n = \lambda n.\ n$	Transitivity on 37 and 38	at	7
40	$\vdash \lambda n.\ (0 * n) + n = \lambda n.\ n$	Transitivity on 7 and 39	at	4
41	$\vdash \lambda n.\ (n * 0) + n = \lambda n.\ n$	Transitivity on 6 and 40	at	11

https://doi.org/10.1017/S0956796899003391 Published online by Cambridge University Press

Table 2. *Theorems generated when optimising for unchanged subterms*

1	$\vdash n * 0 = 0 * n$	Rewriting with Equation 3	at 1
2	$\vdash (+\ (n * 0)) = (+\ (0 * n))$	`AP_TERM` on ‘‘+‘‘ and 1	at 2
3	$\vdash (n * 0) + n = (0 * n) + n$	`AP_THM` on 2 and ‘‘n‘‘	at 2
4	$\vdash \lambda n.\ (n * 0) + n = \lambda n.\ (0 * n) + n$	`ABS` on 3	at 2
	`CHANGED_CONV` fails		
5	$\vdash 0 * n = 0$	Rewriting with Equation 2	at 3
	`CHANGED_CONV` fails		
6	$\vdash (+\ (0 * n)) = (+\ 0)$	`AP_TERM` on ‘‘+‘‘ and 5	at 6
	`CHANGED_CONV` *succeeds*		
	`CHANGED_CONV` fails		
7	$\vdash (0 * n) + n = 0 + n$	`AP_THM` on 6 and ‘‘n‘‘	at 6
	`CHANGED_CONV` *succeeds*		
8	$\vdash 0 + n = n$	Rewriting with Equation 1	at 9
	`CHANGED_CONV` fails		
9	$\vdash (0 * n) + n = n$	Transitivity on 7 and 8	at 7
10	$\vdash \lambda n.\ (0 * n) + n = \lambda n.\ n$	`ABS` on 9	at 6
	`CHANGED_CONV` *succeeds*		
11	$\vdash \lambda n.\ (n * 0) + n = \lambda n.\ n$	Transitivity on 4 and 10	at 11

reflexivity rule, the reflexive theorem is constructed from theorems for the leaf nodes by application of congruence rules. This behaviour is a consequence of using the most straightforward algorithm:

1. If a rewrite is applicable to the term, apply it; otherwise,
2. if the term is a variable or a constant apply the reflexivity rule; otherwise,
3. recursively apply the algorithm to the body of the abstraction or the operator and operand of the combination (as appropriate), then combine the results.

The redundancies illustrated in the example are due to sequencing of conversions that may leave the term unchanged.

Another, less common, inefficiency arises when conversions are sequenced. Terms may be traversed more than once resulting in redundant applications of congruence rules. In the example, only one application of the `ABS` rule should be required but there are two, one arising in `ONCE_DEPTH_CONV` and the other in `TOP_DEPTH_CONV`.

Thus there are two kinds of inefficiency:

- application of inference rules when a subterm has not been changed;
- repeated traversals of a term when one would be sufficient.

These inefficiencies are a consequence of the implementation of conversions. The first kind can be avoided by means of a more delicate rewriting algorithm (section 4). Table 2 lists the inferences performed, a reduction in number from 41 to 11. The second inefficiency really requires proof steps to be combined and reordered, for which it is necessary to delay some of the computation (section 6). The inferences performed when both inefficiencies are avoided are given later (section 6.1) in Table 3.

https://doi.org/10.1017/S0956796899003391 Published online by Cambridge University Press

4 Optimisation using exceptions

The application of inference rules when a subterm has not been changed can be avoided by propagating a value representing 'unchanged' back up the term structure. This value is generated at any leaf node (a variable or a constant) that is unchanged. At an abstraction the value is allowed to propagate. At a function application the value is only allowed to propagate when 'unchanged' is obtained for both the operator and the operand. This modification to the basic algorithm is discussed by Huet (1989) and was first used for HOL by Roger Fleming of Hewlett Packard Labs, Bristol, England.

In 1990, Fleming developed a rewriting package for HOL which exploited ML exceptions to simply and efficiently encode the 'unchanged' value. An exception is raised at any unchanged leaf node and caught at application nodes. Fleming's code had a different behaviour to the original HOL rewriter and lost much of the modularity of Paulson's code. It was not installed in the main part of the HOL system because, due to the difference in behaviour, to have done so would have been very disruptive to users. Tom Melham also used exceptions in 1990 to optimise one of the rewriting functions in HOL. Then, in 1991, the current author combined the exception technique with the modular approach allowing the whole of rewriting in HOL to be optimised without changing the behaviour. Using exceptions to avoid rebuilding unchanged data structures is a technique that may be known to many programmers. The contribution in this paper is showing how the technique can be used when programs that manipulate the data structures are constructed from combinators. In such situations, the exceptions should only be mentioned inside the implementation of the combinators.

Fleming's technique is made modular by reimplementing the basic conversions so that they raise and trap an 'unchanged' exception (figure 7). The new conversions have the property that they may generate an 'unchanged' exception rather than return a theorem. This simply means that the argument term has not been changed by the conversion. A 'wrapper' function `RULE_OF_CONV` is provided to trap the exception and return a theorem of the form $\vdash t = t$ instead by applying a *single* primitive inference rule (`REFL`). The function `RULE_OF_CONV` should only be used when it is absolutely necessary to have a theorem, i.e. when there is no more equational reasoning to be done, since it localises the optimisation.

The `UNCHANGED` exception is raised by the identity conversion `ALL_CONV` and by other basic conversions which can leave terms unchanged.

In the implementation of `THENC`, if the application of the first conversion does not change the term, an exception will be raised. This is caught by the handler in the last line, and the second conversion is then applied to the *original* term. Since the first conversion left the original term unchanged, the result will be correct, but it is obtained without having to generate a theorem for the result of the first conversion. If the first conversion does change the term, but the second one does not, another exception handler causes the result of applying the first conversion to be returned as the full result.

The alternation operator on conversions, `ORELSEC`, is the most delicate of the

https://doi.org/10.1017/S0956796899003391 Published online by Cambridge University Press

```
val ALL_CONV : conv = fn _ => raise UNCHANGED;

val NO_CONV : conv = fn _ => raise CONV_ERR "NO_CONV";

fun conv1 THENC conv2 : conv =
   fn tm => (let val th1 = conv1 tm
              in  TRANS th1 (conv2 (rhs (concl th1)))
                  handle UNCHANGED => th1
              end
              handle UNCHANGED => conv2 tm);

fun conv1 ORELSEC conv2 : conv =
   fn tm => (conv1 tm
              handle UNCHANGED => raise UNCHANGED
                   | _ => conv2 tm);

fun RATOR_CONV conv tm =
   let val {Rator,Rand} = dest_comb tm
                          handle _ => raise CONV_ERR "RATOR_CONV"
   in  AP_THM (conv Rator) Rand
   end;

fun RAND_CONV conv tm =
   let val {Rator,Rand} = dest_comb tm
                          handle _ => raise CONV_ERR "RAND_CONV"
   in  AP_TERM Rator (conv Rand)
   end;

fun ABS_CONV conv tm =
   let val {Bvar,Body} = dest_abs tm
                         handle _ => raise CONV_ERR "ABS_CONV"
       val Bodyth = conv Body
   in  (ABS Bvar Bodyth handle _ => raise CONV_ERR "ABS_CONV")
   end;

fun RULE_OF_CONV conv tm = conv tm handle UNCHANGED => REFL tm;
```

Fig. 7. Conversions using exceptions.

new definitions. It is meant to trap the exceptions that are generated because of errors. If the exception is UNCHANGED, it is regenerated. Otherwise the result of the ORELSEC becomes the result of applying the second conversion to the original term. So, ORELSEC applies the first conversion and if an exception is raised because of an error the second conversion is applied instead. If the first conversion simply leaves the term unchanged then the UNCHANGED exception used to indicate this is allowed to propagate.

The definitions of RATOR_CONV, RAND_CONV and ABS_CONV are unchanged. Note, however, that the exception handler for dealing with terms of the wrong form is kept close to the term destructor function. If it were placed around the whole body of RATOR_CONV (etc.), it would also trap the UNCHANGED exception, which is not the

https://doi.org/10.1017/S0956796899003391 Published online by Cambridge University Press

```
let val th1 = conv1 tm
    val tm1 = rhs (concl th1)
in  if (is_neg tm1)
    then TRANS th1 (RAND_CONV conv2 tm1)
    else TRANS th1 (conv2 tm1)
end
```

Fig. 8. Selective application of a conversion.

desired behaviour. For RATOR_CONV and RAND_CONV the UNCHANGED exception can be allowed to propagate because they only affect either the operator or the operand of an application, never both. So, if an UNCHANGED exception is raised for the part that can be affected, the whole application must be unchanged.

By using the UNCHANGED exception a user could interfere with the workings of the optimisation but this is very unlikely to happen by accident and could be prevented entirely by hiding the exception using the type system of ML.

So, the new implementations of the basic conversion functions behave in the same way as the originals (provided RULE_OF_CONV is used at the top level), but are more efficient. The code for depth conversions remains unchanged since these conversions are implemented in terms of the basic functions. This also applies to the majority of users' code. Thus, the optimisation is largely transparent to the user.

One way in which the technique is not a transparent change is if the programmer wishes to perform some special operation within a conversion as opposed to building it entirely from other conversions. In such a case, the programmer may wish to bind an intermediate result using a let-expression. The code fragment in figure 8 is an example of this. The right-hand side of the result of applying conv1 is obtained explicitly so that its form can be examined before applying conv2. If the term is a negation then conv2 is applied only to the argument of the negation, rather than to the whole term.

The problem with the code is that application of conv1 may raise an UNCHANGED exception instead of returning a theorem. As the code stands, this exception will escape from the let-expression without conv2 being applied. The obvious solution is to ensure that a theorem is generated by applying RULE_OF_CONV. However, this destroys the optimisation. Another solution is to introduce an explicit exception handler but that is messy and the optimisation is no longer transparent. The code for applying conv2 has to be duplicated (though not exactly) as shown in figure 9. In the handler code it is known that the original term is the same as the term that would have been bound to the variable tm1.

Close inspection reveals that the exception traps introduced are similar in form to those appearing in the definition of THENC. This is not surprising since two conversions are being sequenced. In fact, the code can be rewritten using THENC in such a way that explicit handling of exceptions is not required (figure 10). The function CHANGED_CONV introduced in section 2 can be implemented in this way. However, the approach is not always practical.

In the next section an alternative implementation of conversions is presented that allows an UNCHANGED value to be propagated without using exceptions.

https://doi.org/10.1017/S0956796899003391 Published online by Cambridge University Press

```
let val th1 = conv1 tm
    val tm1 = rhs (concl th1)
in  if (is_neg tm1)
    then (TRANS th1 (RAND_CONV conv2 tm1) handle UNCHANGED => th1)
    else (TRANS th1 (conv2 tm1) handle UNCHANGED => th1)
end
handle UNCHANGED =>
if (is_neg tm)
then RAND_CONV conv2 tm
else conv2 tm
```

Fig. 9. Selective application with exceptions.

```
(conv1 THENC (fn tm1 => if (is_neg tm1)
                        then RAND_CONV conv2 tm1
                        else conv2 tm1))
tm
```

Fig. 10. Selective application using basic conversions.

5 Optimisation using an ML data type

An alternative to using exceptions for optimisation of equational reasoning is to use an ML data type such as the one in figure 11. A value of this type is either a theorem or an indicator that the original term has not been changed by the conversion. The definitions of the basic conversion functions for this new type are given in figure 12. A slight variation is for the unchanged term to be omitted from the `Unchanged` constructor, i.e. for `Unchanged` to have no argument. Without the term argument an `equation` is incomplete as a stand-alone object. There may then be a loss of information in going from a term to an equation. This is important if the `equation` type is to be developed further, as described in section 7.

The disadvantage with this approach to avoiding unnecessary inferences for unchanged subterms is that conversions no longer return theorems. Their new type is:

```
term -> equation
```

In respect of their types, conversions are no longer simply a special form of inference rule. Hence, conversions do not interface directly with general rules such as `TRANS`. The function `RULE_OF_CONV` acts as an interface by converting values of type `equation` to values of type `thm`. The constructor `Equation` performs a conversion in the other direction. The conversion between the two types is required wherever conversions are used as rules. Adding calls to the interface functions is not difficult but the changes required to users' code are tedious. This issue is discussed at greater depth in section 7.

```
datatype equation = Equation of thm | Unchanged of term
```

Fig. 11. ML data type for optimisation.

https://doi.org/10.1017/S0956796899003391 Published online by Cambridge University Press

```
val ALL_CONV = Unchanged;

val NO_CONV : conv = fn _ => raise CONV_ERR "NO_CONV";

fun conv1 THENC conv2 : conv =
   fn tm => let val eq1 = conv1 tm
            in  case eq1
                of Equation th1 =>
                      (case (conv2 (rhs (concl th1)))
                       of Equation th2 => Equation (TRANS th1 th2)
                        | Unchanged _ => eq1)
                 | Unchanged _ => conv2 tm
            end;

fun conv1 ORELSEC conv2 : conv =
   fn tm => (conv1 tm handle _ => conv2 tm);

fun RATOR_CONV conv tm =
   let val {Rator,Rand} = dest_comb tm
                          handle _ => raise CONV_ERR "RATOR_CONV"
   in  case (conv Rator) of Equation th => Equation (AP_THM th Rand)
                          | Unchanged _ => Unchanged tm
   end;

fun RAND_CONV conv tm =
   let val {Rator,Rand} = dest_comb tm
                          handle _ => raise CONV_ERR "RAND_CONV"
   in  case (conv Rand) of Equation th => Equation (AP_TERM Rator th)
                         | Unchanged _ => Unchanged tm
   end;

fun ABS_CONV conv tm =
   let val {Bvar,Body} = dest_abs tm
                         handle _ => raise CONV_ERR "ABS_CONV"
   in  case (conv Body)
       of Equation th =>
          Equation (ABS Bvar th handle _ => raise CONV_ERR "ABS_CONV")
        | Unchanged _ => Unchanged tm
   end;

fun RULE_OF_CONV conv tm =
   case (conv tm) of Equation th => th | Unchanged tm => REFL tm;
```

Fig. 12. Conversions using an ML data type.

6 Laziness enables further optimisation

Once the move has been made to using a new ML data type to represent the result of applying a conversion, it is straightforward to extend the data type to provide optimisation of repeated traversals over the term. The optimisation requires the application of inference rules to be delayed. This was described by the author in an earlier paper (Boulton, 1993).

https://doi.org/10.1017/S0956796899003391 Published online by Cambridge University Press

In that earlier description, the postponement of application of inference rules was mixed with more general laziness. In fact there were two levels of laziness: (i) the application of (lazy) inference rules was delayed by use of a special data structure for equational reasoning, and (ii) when the structure was processed to obtain a general theorem, the application of real inference rules was postponed again, until the user forced the inferences to take place. Here, only the first kind of laziness is used, which is sufficient to allow restructuring of the proof or even entirely different proof strategies to be used if they are more efficient.

Repeated traversals of the same subterm do not normally occur within a basic proof procedure because the programmer will have tried to avoid them. However, when component procedures are combined, as is common with conversions, repeated traversals may occur, with one coming from the application of one component and another coming from a second component. The number of repeated traversals will depend a lot on how the component procedures are combined.

Repeated traversals may also arise when a theorem prover like HOL is interfaced to another system. Such systems include graphical user interfaces in which subterms are selected for rewriting using the mouse (Bertot *et al.*, 1994), and proof planners such as CLAM (Bundy *et al.*, 1991). In both cases rewrites may be expressed as operations at a subterm in which the subterm is addressed by means of a path from the root of the term. For example, here is a fragment of a proof script generated by an interface of CLAM with the HOL system (Boulton *et al.*, 1998):

```
OCC_RW_TAC "hol_APPEND1" [1,1,1] LEFT
  THEN OCC_RW_TAC "hol_REV2" [1,1] LEFT
  THEN OCC_RW_TAC "hol_REV1" [1,1,1] LEFT
  THEN OCC_RW_TAC "hol_APPEND1" [1,1] LEFT
  THEN OCC_RW_TAC "hol_REV1" [2,2,1] LEFT
```

This compound tactic, generated from a CLAM proof plan, is executed by HOL to obtain a theorem. So the efficiency of the tactic affects the speed of the combined system. The tactic `OCC_RW_TAC` rewrites with the named equation (first argument) at the specified position (second argument). The third argument specifies the direction in which the equation is to be used. In this sequence of rewrites each rewrite is addressed from the root of the term, so a literal interpretation by the theorem prover results in multiple traversals of the term. Some of these traversals are unnecessary because, for example, the first four rewrites all occur within the subterm at position `[1,1]`.

Each traversal involves reconstructing the term around the changed parts. This is costly in both time and memory usage. In a fully expansive theorem prover the cost is especially high because logical inference is required to reconstruct the term (as a theorem). For small examples, like the one above, the cost is not significant but it could be for large terms.

Efficiency of proof creation is not the only issue; the size and form of the proof itself may also be important. Size matters if a log of the proof is created (e.g. for independent checking) or if a proof object is extracted, and if it is a synthesis proof the form may affect the efficiency of the synthesized program.

```
datatype change = Rewrite of thm
                | Changed_comb of (change)list * (change)list
                | Changed_abs of (change)list
```

Fig. 13. ML type of rewriting structures for equational reasoning.

Table 3. *Theorems generated using a rewriting structure*

1	$\vdash n * 0 = 0 * n$	Rewriting with Equation 3
2	$\vdash 0 * n = 0$	Rewriting with Equation 2
3	$\vdash n * 0 = 0$	Transitivity on 1 and 2
4	$\vdash (+\ (n * 0)) = (+\ 0)$	`AP_TERM` on ``+'' and 3
5	$\vdash (n * 0) + n = 0 + n$	`AP_THM` on 4 and ``n''
6	$\vdash 0 + n = n$	Rewriting with Equation 1
7	$\vdash (n * 0) + n = n$	Transitivity on 5 and 6
8	$\vdash \lambda n.\ (n * 0) + n = \lambda n.\ n$	`ABS` on 7

6.1 Rewriting structures

Optimisation of repeated traversals is achieved by extending the data type described in section 5 to create a *rewriting structure*. The new type, `change`, is shown in figure 13. A `change` is essentially the result of a rewrite. A `(change)list` is a sequence of rewrites. Sequencing may occur at any level in the term structure.

A rewriting structure mimics the structure of HOL terms (see section 1.2). There are internal nodes (`Changed_comb` and `Changed_abs`) for function applications (combinations) and λ-abstractions respectively, and a leaf node for subterms that have had a basic rewrite applied to them (`Rewrite`). At each internal node a sequence of rewrites can occur.

When used as the result of conversions instead of a theorem, the rewriting structure allows the same optimisations as the techniques in sections 4 and 5. An empty `change` list indicates that there has been no change. The rewriting structure provides additional optimisations, as can be seen in Table 3, which is based on the example in section 3. To achieve this optimisation it is necessary to delay the application of congruence rules. Because of this delay the references to positions in the conversion code are no longer applicable. The table lists the theorems generated using an algorithm acting over the rewriting structure. The repeated traversals of the term and subterms are eliminated. The algorithm is described in section 6.3. Before that the implementations of the basic conversion functions for the rewriting structure are presented.

6.2 Lazy conversions

To provide the performance benefits of the rewriting structure and the ease of programming offered by conversions, the latter have been modified to return a rewriting structure together with a representation of the theorem as terms. The type

https://doi.org/10.1017/S0956796899003391 Published online by Cambridge University Press

of these *lazy conversions* is:

```
term -> (((term)list * term * term) * (change)list)
```

The first component of the result is a triple consisting of a list of hypotheses generated by the rewrites, the left-hand side of the equation, and the right-hand side. The two sides of the equation are maintained as separate terms for ease of programming. The term structure of the equation has to be present in addition to the rewriting structure so as to allow functions to test the result. The rewriting structure can not provide this information unless it is evaluated, and to do that would neutralise the optimisation.

Since conversions conventionally return a theorem it is necessary to include a representation for hypotheses in the type of lazy conversions. A conversion might introduce hypotheses for a number of reasons, but application of a conditional rewrite rule is the most likely one. For example, in applying the theorem

$$0 < x \vdash x \texttt{ DIV } x = 1$$

as a rewrite rule, the hypothesis $0 < x$ will (in an instantiated form) become a hypothesis of the result of the conversion. Typically, a conditional rewriter will try to prove the hypothesis immediately but it might alternatively allow the hypothesis to be propagated so that it can eventually be presented to the user to prove.

The implementations of the basic conversion functions are given in figures 14 and 15. The identity lazy conversion, `ALL_CONV`, simply takes a term and returns the same term as the result with an empty hypothesis list and an empty list of changes (`change` list).

The implementation of `THENC` uses a set-theoretic union operation to combine the hypotheses generated by the two conversions it sequences. The `sequence_changes` function is at the heart of the optimisation. It is described in detail in section 6.3. The code for `THENC` illustrates the fact that it is not necessary to return the argument term as part of the result of the conversion. In `THENC` the values returned are thrown away by pattern matching with the wildcard character (an underscore). However, in practice it is better to keep the argument term (the left-hand side of the equation) explicitly in the structure. It is required, for example, in order to print the structure as an equation.

The functions `RATOR_CONV` and `RAND_CONV` test to see if any changes have been brought about by the conversion. If not, an empty sequence is returned instead of a `Changed_comb` with empty sequences for both its arguments.

The implementation of `ABS_CONV` has to test explicitly for the presence of the bound variable of the abstraction in the hypotheses (as a free occurrence). It cannot rely on the test in the `ABS` rule (see section 1.3.2 for the side-condition on `ABS`) because `ABS` is not applied until later. If the bound variable test was not done at this stage the behaviour of `ABS_CONV` with respect to `ORELSEC` would change. For example, the conversion:

```
(ABS_CONV conv1) ORELSEC conv2
```

would produce a structure based on the application of `conv1` when it should produce

https://doi.org/10.1017/S0956796899003391 Published online by Cambridge University Press

```
val ALL_CONV : conv = fn tm => (([],tm,tm), []);

val NO_CONV : conv = fn _ => raise CONV_ERR "NO_CONV";

fun conv1 THENC conv2 : conv =
   fn tm => let val eq1 as ((hyps1,_,tm1),chgs1) = conv1 tm
               val eq2 as ((hyps2,_,tm2),chgs2) = conv2 tm1
            in  case (chgs1,chgs2)
                of ([],_) => eq2
                 | (_,[]) => eq1
                 | _ => ((union hyps1 hyps2,tm,tm2),
                         sequence_changes chgs1 chgs2)
            end;

fun conv1 ORELSEC conv2 : conv =
   fn tm => (conv1 tm handle _ => conv2 tm);

fun RATOR_CONV conv tm =
   let val {Rator,Rand} = dest_comb tm
                          handle _ => raise CONV_ERR "RATOR_CONV"
       val ((hypsf,_,tmf),chgsf) = conv Rator
   in  case chgsf
       of [] => (([],tm,tm), [])
        | _ => ((hypsf,tm,mk_comb {Rator = tmf,Rand = Rand}),
                [Changed_comb (chgsf, [])])
   end;

fun RAND_CONV conv tm =
   let val {Rator,Rand} = dest_comb tm
                          handle _ => raise CONV_ERR "RAND_CONV"
       val ((hypsx,_,tmx),chgsx) = conv Rand
   in  case chgsx
       of [] => (([],tm,tm), [])
        | _ => ((hypsx,tm,mk_comb {Rator = Rator,Rand = tmx}),
                [Changed_comb ([], chgsx)])
   end;

fun ABS_CONV conv tm =
   let val {Bvar,Body} = dest_abs tm
                         handle _ => raise CONV_ERR "ABS_CONV"
       val ((hypsb,_,tmb),chgsb) = conv Body
   in  case chgsb
       of [] => (([],tm,tm), [])
        | _ => if (mem Bvar (free_varsl hypsb))
               then raise CONV_ERR "ABS_CONV"
               else ((hypsb,tm,mk_abs {Bvar = Bvar,Body = tmb}),
                     [Changed_abs chgsb])
   end;
```

Fig. 14. Conversions using a rewriting structure.

https://doi.org/10.1017/S0956796899003391 Published online by Cambridge University Press

```
fun RULE_OF_CONV conv tm =
   let val ((hyps,tm,tm'),chgs) = conv tm
       val th = prove_changes chgs tm
       val (h,c) = dest_thm th
       val {lhs,rhs} = dest_eq c
   in  if (h = hyps) andalso (lhs = tm) andalso (rhs = tm') then th
       else raise CONV_ERR "RULE_OF_CONV"
   end;
```

Fig. 15. Interface function for rewriting structures.

one based on `conv2`. An exception would be raised when an attempt is made to generate a theorem from the structure based on `conv1`. One of the real difficulties with delaying computation is this kind of interaction with exceptions.

The conversion functions assume that the hypothesis list must be empty if no changes have occurred. Note also that no inference rules are applied. The application of inference rules takes place later, when a theorem is generated from the structure by the function `prove_changes` (section 6.4). This function is called by `RULE_OF_CONV`, which could simply return the theorem. However, it first verifies that the theorem generated corresponds to the term structure. This is a consistency check for user-defined conversions.

6.3 Sequencing rewriting structures

Lazy conversions are applied in sequence using the infix function `THENC` (figure 14). The lazy conversion generated by sequencing `conv1` and `conv2` takes a term and applies `conv1` to it. The conversion `conv2` is then applied to the resulting term. The function tests to see if either of the `change` lists are empty (indicating that the term was not changed). If this is so, the computation can be optimised. In particular, if both `change` lists are empty then the list returned will be empty. If neither list is empty the set-theoretic union of the hypotheses is formed and the `change` lists are combined using the function `sequence_changes` (figure 16).

Concatenating two `change` sequences requires two mutually recursive functions: one to traverse the first sequence until it encounters the last `change`; the other to combine this `change` with the first one in the second sequence. If either sequence is empty the other is returned. Two `changes` can be merged if they are of the same kind, i.e. both abstraction nodes, both combination nodes, or both leaf nodes. In the first two cases, merging is done simply by making recursive calls to concatenate the sequences for the subterms. In the last case, a new `Rewrite` node is formed by applying the transitivity rule, `TRANS`, to the theorems in the original nodes.

Thus, `sequence_changes` constructs a new sequence of `changes` in such a way that any repeated traversals of subterms are avoided. The mechanism is capable of merging two traversals into one provided they are not separated by a structure-destroying operation. So, two rewrites within the operator of a function application can be merged into one even when separated by a rewrite within the operand. This is reordering of rewrites.

https://doi.org/10.1017/S0956796899003391 Published online by Cambridge University Press

```
fun sequence_change (Rewrite th1) (Rewrite th2) =
       [Rewrite (TRANS th1 th2)]
  | sequence_change (Changed_comb (chgs1f,chgs1x))
                    (Changed_comb (chgs2f,chgs2x)) =
       [Changed_comb (sequence_changes chgs1f chgs2f,
                      sequence_changes chgs1x chgs2x)]
  | sequence_change (Changed_abs chgs1b) (Changed_abs chgs2b) =
       [Changed_abs (sequence_changes chgs1b chgs2b)]
  | sequence_change chg1 chg2 = [chg1,chg2]

and sequence_changes [] chgs2 = chgs2
  | sequence_changes chgs1 [] = chgs1
  | sequence_changes [chg1] (chg2 :: chgs2) =
       (sequence_change chg1 chg2) @ chgs2
  | sequence_changes (chg1 :: chgs1) chgs2 =
       chg1 :: (sequence_changes chgs1 chgs2);
```

Fig. 16. Sequencing rewriting structures.

There is a price to pay for the elimination of repeated traversals: the rewriting structures have to be manipulated in addition to the theorems. So, there is extra computation and extra garbage. In fact, lazy conversions would not be an optimisation in many theorem provers because the manipulation of the rewriting structures would be all that was required. It is only because theorem generation is so costly in a fully expansive theorem prover that the technique may be worthwhile. In principle, use of rewriting structures may be a de-optimisation even in HOL. However, in most cases, sufficient primitive inferences are avoided to outweigh the overheads.

6.4 Evaluation using congruence rules

When the result of some equational reasoning is required as a theorem so that more general reasoning can be performed, the final rewriting structure must be processed. This can be done by applying the congruence rules. An implementation using mutually recursive functions is given in figure 17.

The functions take a term as an argument in addition to the rewriting structures. This is required because the rewriting structures do not necessarily record the full term structure. The rewriting structure is checked for consistency with the term. If the two are not consistent an exception is raised. This should only occur if the functions are incorrectly applied (or there is a bug in the implementation).

At each leaf node (`Rewrite`) the theorem is tested to make sure that its left-hand side is equal to the argument term. These theorems are combined at each of the internal nodes (`Changed_comb` and `Changed_abs`) using the appropriate congruence rule. Different rules are used for `Changed_comb` nodes depending on whether one or other or both or neither of the arguments is an empty list. In fact, the case of both lists being empty should not arise because of the way `RATOR_CONV` and `RAND_CONV` are defined, but it is handled for completeness.

Sequences are processed by `prove_changes` using the transitivity rule, `TRANS`. An

https://doi.org/10.1017/S0956796899003391 Published online by Cambridge University Press

```
fun prove_change (Rewrite th) tm =
       if (lhs (concl th) = tm)
       then th
       else raise CONV_ERR "prove_change"
  | prove_change (Changed_comb (chgsf,chgsx)) tm =
       let val {Rator = f,Rand = x} =
              dest_comb tm handle _ => raise CONV_ERR "prove_change"
       in  case (chgsf,chgsx)
           of ([],[]) => REFL tm
            | ([],_) => AP_TERM f (prove_changes chgsx x)
            | (_,[]) => AP_THM (prove_changes chgsf f) x
            | (_,_) => MK_COMB (prove_changes chgsf f,
                                prove_changes chgsx x)
       end
  | prove_change (Changed_abs chgsb) tm =
       let val {Bvar = var,Body = b} =
              dest_abs tm handle _ => raise CONV_ERR "prove_change"
       in  case chgsb
           of [] => REFL tm
            | _ => ABS var (prove_changes chgsb b)
       end

and prove_changes [] tm = REFL tm
  | prove_changes (chg :: chgs) tm =
       let val th = prove_change chg tm
       in  case chgs
           of [] => th
            | _ => TRANS th (prove_changes chgs (rhs (concl th)))
       end;
```

Fig. 17. Evaluating rewriting structures.

optimisation is made when the sequence has only one element. Instead of making a recursive call on an empty sequence, causing `REFL` to be applied to the term, and then using transitivity, the theorem for the single `change` in the sequence is returned directly.

Rewriting structures (as implemented by the `change` type) can be constructed that do not represent a valid proof. So, when inference rules are applied in the `prove_change` and `prove_changes` functions, an exception may be raised. In practice, however, the functions are used in a way that will not lead to a failure.

7 Abstract type for equational reasoning

Within this paper, a number of different ways of implementing equational reasoning in HOL have been presented, but in each case the behaviour of the functions `ALL_CONV`, `NO_CONV`, `THENC`, `ORELSEC`, `RATOR_CONV`, `RAND_CONV` and `ABS_CONV` has remained largely the same. They thus form the basis for an abstract type of equations for which users have no access to the underlying representation. In this section, the minimum set of functions required to permit flexible programming while retaining

https://doi.org/10.1017/S0956796899003391 Published online by Cambridge University Press

```
signature Equation =
sig
   type equation

   val THM_OF_EQN   : equation -> thm
   val EQN_OF_THM   : thm -> equation

   val RULE_OF_CONV : (term -> equation) -> (term -> thm)
   val CONV_OF_RULE : (term -> thm) -> (term -> equation)

   val EQUATION     : (((term)list * term * term) * (term -> thm)) ->
                         equation
   val EQN_HYP      : equation -> (term)list
   val EQN_LHS      : equation -> term
   val EQN_RHS      : equation -> term
   val EQN_PROOF    : equation -> (term -> thm)

   val SYM_EQN      : equation -> equation

   val NO_CONV      : term -> equation
   val ALL_CONV     : term -> equation
   val THENC        : (term -> equation) * (term -> equation) ->
                         (term -> equation)
   val ORELSEC      : (term -> equation) * (term -> equation) ->
                         (term -> equation)
   val RATOR_CONV   : (term -> equation) -> (term -> equation)
   val RAND_CONV    : (term -> equation) -> (term -> equation)
   val ABS_CONV     : (term -> equation) -> (term -> equation)

   val REWR_CONV    : thm -> (term -> equation)

   val COMB_CONV    : (term -> equation) -> (term -> equation)
end
```

Fig. 18. SML-style signature for equations.

portability across the various optimisation techniques is presented. These functions constitute the code that has to be changed to achieve the optimisations. A number of additional functions that are commonly used and whose performance benefits from being defined directly are also included in the abstract type.

7.1 Signature for equations

The ML type `equation` was introduced in section 5 as the result type of conversions. By making this type abstract, the implementation of conversions can be optimised in the ways already discussed, and in other ways not yet thought of, without the need to change any of the higher-level system code or users' code. The abstract type for equations is presented in figure 18 as a Standard ML signature. The purpose of each function in the abstract type is specified below.

The functions `THM_OF_EQN` and `EQN_OF_THM` convert an equation to a theorem and

https://doi.org/10.1017/S0956796899003391 Published online by Cambridge University Press

vice versa. The former may involve proof. It cannot be implemented properly when exceptions are used as the optimisation technique. This is because an `equation` for an unchanged term is an exception rather than a concrete value. Similar remarks apply to conversions based on an ML data type when unchanged terms are not included explicitly in the representation. Thus, the use of `THM_OF_EQN` is to be avoided if one wants to allow these optimisation techniques. Wherever possible the function `RULE_OF_CONV`, described below, should be used instead. It has access to the initial term; hence it has complete information.

The functions `RULE_OF_CONV` and `CONV_OF_RULE` perform similar tasks to those of `THM_OF_EQN` and `EQN_OF_THM`, but at the level of rules. The former produces a HOL rule from a conversion and is equivalent to the function of the same name described earlier. The functions provide an interface between equational reasoning and more general reasoning. They must be added to existing ML code wherever there is a switch between these two kinds of reasoning. In the current versions of HOL with no optimisation of equational reasoning, the type `equation` corresponds to the type `thm` of theorems and the four interface functions are simply the identity function.

The function `EQUATION` is used when defining primitive conversions. Four things must be constructed: a list of hypotheses Γ, the left-hand side l of the equation, the right-hand side r, and a function f that given l will return a theorem $\Gamma \vdash l = r$. The function f should be constructed using general HOL inference rules. It must justify the triple (Γ, l, r) by generating a theorem with the same structure. This approach allows the inferences involved in the proof to be delayed. For example, a conversion to evaluate additions, e.g. to generate $\vdash 1 + 2 = 3$ from the term ``$1 + 2$'', might be implemented as follows:

```
fun PLUS_CONV tm =
   let val (arg1,arg2) = dest_plus tm
       val n1 = int_of_term arg1
       and n2 = int_of_term arg2
       val tm' = term_of_int (n1 + n2)
   in  EQUATION (([],tm,tm'),fn tm => ... tm ...)
   end;
```

The functions `EQN_HYP`, `EQN_LHS`, `EQN_RHS`, and `EQN_PROOF` reverse the operation of `EQUATION` by extracting components. A function for each component is provided, rather than a single inverse function, in order to avoid building structures when extracting components. The internal structure of equations is not known, so it may be the case that returning a structure of type

```
((term)list * term * term) * (term -> thm)
```

requires destruction and construction, while obtaining just one component only requires destruction. Thus, using separate functions leaves the issue of unnecessary garbage in the hands of the user.

It is often easier to perform certain kinds of equational reasoning by proving the symmetric equation to the one required and then reversing the left and right-hand

sides. A simple example of this is the introduction of a double negation. Assuming a procedure already exists to eliminate double negations, the easiest way to write a procedure to introduce double negations is to construct a double negation around the argument term and apply the elimination procedure to this new term. Thus, for a term t the following equation is obtained:

$$\neg\neg t = t$$

Reversing this using the symmetry rule produces the required equation. The problem with this is that the symmetry rule operates over values of type `thm` as opposed to values of type `equation`. So, a conversion using this technique has to switch between equational and general reasoning, thus limiting optimisation. The function `SYM_EQN` is included in the abstraction so that equations can be reversed directly. However, it is still not easy to allow optimisations to propagate. For example, if using rewriting structures in the implementation they would have to be reversed. It is not clear how to do this in general, or that it is even possible. However, the most common optimisation to be propagated is the one for unchanged subterms. In this case, the rewriting structure is an empty list, so it does not need to be reversed. In other cases it may be necessary to generate the theorem in order to reverse the equation.

The purposes of `NO_CONV`, `ALL_CONV`, `THENC`, `ORELSEC`, `RATOR_CONV`, `RAND_CONV`, and `ABS_CONV` were explained in Section 2. The function `REWR_CONV` is the primitive rewriting conversion. It takes a theorem (which must be an equation) and uses it as a rewrite rule. In most circumstances `REWR_CONV` could be defined using `EQUATION`. The function `COMB_CONV` applies a conversion to the operator and operand of an application. It too can be derived from other functions in the signature. The reason for its inclusion is discussed in section 7.3.

7.2 Defining new conversions

There are two ways to define a new conversion. The first is to construct a proof (a justification function) and apply `EQUATION` to it. This requires the result (right-hand side of the equation) to be computed by some other means since `EQUATION` also requires this information. The second approach is to build up the conversion from others, e.g. by combining applications of `REWR_CONV` using `THENC`, etc. The first approach has the advantage that all the computation required for justification is delayed (assuming the implementation of equational reasoning allows this). The second has the advantage that non-local optimisations may be possible when the conversion is used as part of more extensive equational reasoning. However, in this case, some computation (such as the generation of rewriting structures) will have to take place immediately so that the value of the right-hand side can be obtained.

7.3 Complex primitives

In section 2 the function `COMB_CONV` was introduced. This applies a conversion to both the operator and the operand of a combination. As stated in that section,

https://doi.org/10.1017/S0956796899003391 Published online by Cambridge University Press

COMB_CONV can be defined in terms of other basic conversion functions:

```
fun COMB_CONV conv = (RATOR_CONV conv) THENC (RAND_CONV conv);
```

However, this definition may not be as efficient as defining COMB_CONV as a primitive. That is why COMB_CONV is included in the signature for equations. The inefficiency of the derived version manifests itself when both the operator and the operand are changed by the conversion. Consider the term ‘‘$f\ x$‘‘ and suppose that the conversion conv transforms f to g and x to y. The application of COMB_CONV to this conversion and the term proceeds as follows:

1. $\vdash f = g$ by application of conv to the operator
2. $\vdash f\ x = g\ x$ by application of AP_THM to 1
3. $\vdash x = y$ by application of conv to the operand
4. $\vdash g\ x = g\ y$ by application of AP_TERM to 3
5. $\vdash f\ x = g\ y$ by transitivity between 2 and 4

Three inferences are required in addition to those involved in the application of conv to f and x. A primitive definition of COMB_CONV can exploit the combined congruence rule for applications (called MK_COMB in HOL) so that only one inference is required in addition to those used by conv.

When rewriting structures are used to optimise conversions, the derived version of COMB_CONV is optimised transparently so that MK_COMB is applied instead of the three rules. So, in this case, there is no need to have COMB_CONV as a basic conversion function, but it is included for the benefit of other implementations. This also demonstrates the effectiveness of the optimisation using rewriting structures.

Similar minimisation of inferences is possible for more complex conversions. However, a compromise has to be made between performance and the ease of implementing and maintaining the code.

8 Results

Table 4 is a comparison of the various techniques for optimising equational reasoning described in this paper. The results are for the HOL benchmark. The benchmark is a verification of a multiplier circuit. Part of the proof involves the use of rewriting and some specialised equational reasoning, but much of it is general tactic-based reasoning.

The table should be interpreted as follows. ‘Run’ is the run time in seconds. This does not include garbage collection time. The garbage collection time in seconds is given separately (‘GC’). The total time is also given. ‘Infs’ is the number of applications of primitive inference rules used to prove the theorems. The inferences are considered to have taken place only when the real theorems have been generated. The figures are for a Sun Ultra-1 with 128Mbytes of real memory.

Results are given for version 7 of the HOL90 system and for a number of modified versions of it. Each of the modified versions has the abstract type for equations described in section 7. They differ only in the concrete implementation of the type. Since HOL90 version 7 already uses exceptions to optimise depth conversions (and

https://doi.org/10.1017/S0956796899003391 Published online by Cambridge University Press

Table 4. *Results for the HOL multiplier benchmark*

Conversion type	Run	GC	Total	Infs
HOL90.7	8.02	0.92	8.94	16287
Ordinary	11.42	1.00	12.42	71357
Exceptions	8.05	0.96	9.01	16277
ML data type	8.16	0.93	9.09	16277
Rewriting structures	8.14	0.98	9.12	16035

Table 5. *Results for rewriting a large sum of 0's and 1's*

Conversion type	Run	GC	Total	Infs
HOL90.7	0.44	0.04	0.48	3395
Ordinary	3.17	0.08	3.25	46525
Exceptions	0.53	0.05	0.58	3395
ML data type	0.49	0.05	0.54	3395
Rewriting structures	0.51	0.12	0.63	3395

hence rewriting) there is no gain in performance for implementations of the `equation` type that optimise for unchanged subterms. (In fact, there is a little overhead, probably due to converting between the `equation` and `thm` types.) Comparison with the version using 'ordinary' conversions offers a better indication of the relative performance. The total execution time is reduced from over 12s to about 9s. The reduction in number of inferences is much more significant.

Table 5 presents the results for rewriting the 1's in a large sum of 0's and 1's to `SUC`(0) where `SUC` is the successor function. This example was chosen because it is pure rewriting and involves a lot of unchanged subterms. The reduction in execution time of optimised conversions relative to ordinary conversions is very significant: over five times faster.

There is little discernible difference between the garbage collection figures for use of exceptions versus use of an ML data type. This is somewhat surprising as one might expect more garbage to be generated by the latter technique. (The figures in Table 5 are averages over a large number of executions, so the behaviour is not simply a consequence of the computation being too small for a major garbage collection to occur.) This surprising result is probably a consequence of the particular compiler (Standard ML of New Jersey 0.93) and different results might be obtained with other compilers. This is also true of the run times. The GC time is noticeably higher when the more complex rewriting structures are used.

Optimisation of repeated traversals does not appear to arise much in practice. Only a small amount occurs in the benchmark example. To a certain extent this may be due to the optimisation not being taken up to the tactic level, i.e. sequences of rewriting-tactic applications are not optimised. It is possible to contrive examples

Table 6. *Results for repeated application of* `ONCE_DEPTH_CONV`

Conversion type	Run	GC	Total	Infs
HOL90.7	8.45	0.39	8.84	16852
Ordinary	9.80	0.50	10.30	32594
Exceptions	8.51	0.36	8.87	16849
ML data type	8.58	0.36	8.94	16849
Rewriting structures	7.60	0.43	8.03	2487

where there is a substantial optimisation of repeated traversals (see Table 6, for example), but even then there is only a small reduction in the execution time despite a large reduction in the number of inferences. This suggests that the manipulation of rewriting structures is almost as costly as the inferences they eliminate. The conclusion, then, is that rewriting structures are not worthwhile unless the *size* of the proof is important.

9 Related work

Higher-order functions are commonly used in functional programming to produce programs that are easy to change, the most common example being the use of monads. (See, for example, Wadler (1992).) Monads are used extensively in pure functional languages to implement features such as input/output and exceptions that are done imperatively in an impure functional language such as Standard ML, but they are also of use for structuring Standard ML programs. The rewriting combinators described here are similar to monads in that they localise the changes required to a program in order to produce a different behaviour. They differ in that monads are usually promoted as a means of readily changing the *functionality* of a program, while the rewriting combinators are intended to keep the functionality the same and change only the *resource usage* of the program.

Monads appear to provide a framework in which optimisations such as the ones described here could be achieved transparently. However, for rewriting in HOL the special purpose rewriting combinators seem more appropriate. It is conceivable, though, that monads could be used instead.

Welinder (1995) has investigated the use of partial evaluation to produce efficient conversions. His work only addresses the basic rewriting operations; it does not consider traversal strategies and their impact on efficiency. For a sequence of alternative rewrites (applications of `REWR_CONV` (section 7.1) to equational theorems) he generates specialised code for the particular equations being used. The use of Welinder's system is limited by the overhead of adding new equations: the code has to be regenerated whenever an equation is added. It may be possible to extend his method to depth rewriting but he does not address that.

Huet (1989) addresses unnecessary rebuilding of terms in his description of the implementation of a proof checker for the Calculus of Constructions. The focus is on maintaining sharing of subterms to be more economical with storage. He

https://doi.org/10.1017/S0956796899003391 Published online by Cambridge University Press

describes an implementation of substitution that pairs the resultant term with a Boolean value indicating whether it is the same as the original term. This is done for all subterms and the Booleans are used to avoid reconstructing nodes of the structure when all the subterms are unchanged. This approach is very similar to the one taken in section 5. Because of the storage overhead involved in pairing every subterm with a Boolean value, Huet goes on to describe the use of exceptions (cf. section 4). More generally, Huet's paper is a valuable discussion of the issues involved in implementing representations of, and operations over, (λ-)terms.

There are two important differences between the work described in this paper and that of Huet. First, the work here is concerned with more than just unnecessary rebuilding of data structures; the costly application of inference rules is also being avoided. Hence, there are different performance trade-offs to be made. For example, Huet's technique of pairing subterms with Boolean values would be more reasonable in the context of optimising conversions because the cost of the pairing is likely to be small relative to the cost of the unnecessary inferences that would be avoided. Secondly, this paper is concerned with the encapsulation of optimisations within existing combinators. Huet is not concerned with such modularity. However, he does present CAML macros (CAML is a dialect of ML) for systematically transforming simple copying algorithms into algorithms that maintain sharing. There is no analogous macro facility in Standard ML.

In Chapter 4 of his PhD thesis, Jackson (1995) describes an implementation of rewriting in the Nuprl proof development system. He describes two forms of rewriting, one that uses congruence rules and which may be used for general congruence relations, and another that uses direct computation but which is only applicable for the built-in computational equality. The latter is considerably faster and so is used wherever possible. A single set of functions is used for combining conversions regardless of which form of justification they use. These combinators appear to use tactic-driven congruence proofs to combine the justifications generated by their argument conversions. There is no mention of them being optimised to propagate the performance benefits of direct computation. A nice feature of Jackson's work is the ability to handle the use of different relations for different subterms.

Nipkow (1989) describes an implementation of rewriting for Isabelle (Paulson, 1994) that uses higher-order unification. The rules for reflexivity, symmetry, and transitivity have metavariables for the arguments of the equality relation. There is also a congruence rule for each operator. The rules are applied using the resolution tactic but the full power of higher-order unification is not required; first-order unification is sufficient. Specialised tactics are generated for each of the rules. A rewriting engine is obtained from these using tacticals structured in a very similar way to the depth conversions in HOL and LCF (Paulson, 1983). It is conceivable, therefore, that optimisations along the lines of those described in this paper would be applicable.

Nipkow also describes an implementation that exploits the higher-order capabilities of Isabelle's resolution. In this, the congruence rules are represented generically by a single rule that uses a function-valued metavariable in place of the operator.

The metavariable acts as a general context, allowing depth rewriting to be done with a single higher-order unification. However, the complexity of this operation compared to applications of simple congruence rules might make it just as costly.

Another piece of related work is the investigation of proof techniques (Kromodimoeljo and Pase, 1992) for the Eves verification environment (Craigen *et al.*, 1991). The Eves implementors optimise innermost rewriting, which is naturally an eager process, by arranging for the rewrite rules to be applied lazily. This is very similar to the technique described in the author's earlier paper (Boulton, 1993) but is not intended to be as general. The Eves implementors are looking to avoid only rewrites, not arbitrary proof steps, that will later be thrown away.

10 Conclusions

Equational reasoning forms a substantial part of many proofs in the HOL system. It is usually in the form of rewriting, but more specialised procedures are also used. Paulson's use of combinators provides a flexible way of constructing these procedures. The work described in this paper shows that the use of combinators also allows transparent optimisation of the procedures. The efficiency of the procedures can be improved by changes to the implementations of the combinators only. Programs that use the combinators need not be changed, except that for some optimisations it is necessary to distinguish the result type of equational proof procedures from the type used to represent general theorems of the logic. Functions to convert between the two types must then be added to the application programs but once this has been done the programs become independent of the underlying implementation of the combinators. The changes to the implementation may be as radical as using a different proof strategy, e.g. a substitution rule in place of congruence rules.

With respect to the relative advantages of specific implementations, the results in section 8 show that avoiding processing of unchanged subterms (either by use of exceptions or by means of a new data type) is a worthwhile optimisation, providing a speed-up of over 500% on some examples. Avoiding repeated traversals of terms has little effect on the execution time, so is probably not worth using if the aim is faster proof procedures. However, in a theorem prover for a constructive logic, the optimisation of the proof that corresponds to avoiding repeated traversals may be significant, since it could, in the proofs-as-programs paradigm (Bates and Constable, 1985), lead to more efficient proof objects.

As has already been noted, reducing the number of primitive inferences may be worthwhile even if execution times are not reduced. As another example of this, if it is necessary to output the entire proof for use by another tool (such as a simple verified proof checker (Wong, 1995)), a reduction in the number of inferences may be helpful. Full proof logs for typical verification proofs are so large as to be almost unmanageable, so any reduction in size is to be welcomed. In such situations avoiding repeated traversals may be worth using since on some examples it produces a drastic reduction in the number of inferences.

Another interesting result for implementors of LCF-style theorem provers is that the execution times are nowhere near proportional to the number of primitive

inferences performed. A significant decrease in the number of inferences may have little impact on the execution time. This suggests that (at least for a Standard ML implementation) the use of primitive inference rules for all proof may not entail as large an overhead as has previously been thought. On the other hand, the results may be due to most of the eliminated inferences being simple ones like reflexivity.

Finally, although some of the proposed optimisation techniques are specific to fully expansive theorem proving, avoiding processing of unchanged subterms is more generally applicable within functional programming and beyond. Any algorithm that performs transformations on an immutable data structure and that may leave part of the structure unchanged may be optimised by the techniques described here, in particular by the raising and handling of an exception when a substructure is unchanged. The substructure still has to be traversed but the allocation of memory to build a new copy is eliminated and any sharing of unchanged structures is retained (Huet, 1989). If the data structure is mutable, as is typical in an imperative language, the optimisation is less useful because the structure would not normally be copied anyway.

Acknowledgements

I am grateful to Mike Gordon and Tom Melham for their guidance while this research was conducted and for help with the HOL system. I also thank Rob Arthan, Martin Richards, and the referees for their comments.

References

Allen, S. F., Constable, R. L., Howe, D. J. and Aitken, W. E. (1990) The semantics of reflected proof. *Proc. 5th Annual Symposium on Logic in Computer Science*, pp. 95–105. IEEE Press.

Baader, F. and Nipkow, T. (1998) *Term Rewriting and All That*. Cambridge University Press.

Bates, J. L. and Constable, R. L. (1985) Proofs as programs. *ACM Trans. Programming Lang. & Syst.*, **7**(1), 113–136.

Bertot, Y., Kahn, G. and Théry, L. (1994) Proof by pointing. In: Hagiya, M. and Mitchell, J. C. (eds), *Proc. Int. Symposium on Theoretical Aspects of Computer Software (TACS'94): Lecture Notes in Computer Science 789*, pp. 141–160. Sendai, Japan. Springer-Verlag.

Boulton, R., Slind, K., Bundy, A. and Gordon, M. (1998) An interface between CLAM and HOL. In: Grundy, J. and Newey, M. (eds), *Proc. 11th Int. Conference on Theorem Proving in Higher Order Logics (TPHOLs'98): Lecture Notes in Computer Science 1479*, pp. 87–104. Canberra, Australia. Springer-Verlag.

Boulton, R. J. (1993) Lazy techniques for fully expansive theorem proving. *Formal Methods in System Design*, **3**(1/2), 25–47.

Boyer, R. S. and Moore, J S. (1981) Metafunctions: Proving them correct and using them efficiently as new proof procedures. In: Boyer, R. S. and Moore, J S. (eds), *The Correctness Problem in Computer Science*, pp. 103–184. Academic Press.

Bundy, A., van Harmelen, F., Hesketh, J. and Smaill, A. (1991) Experiments with proof plans for induction. *J. Automated Reasoning*, **7**(3), 303–324.

Church, A. (1940) A formulation of the simple theory of types. *J. Symbolic Logic*, **5**(1), 56–68.

Craigen, D., Kromodimoeljo, S., Meisels, I., Pase, W. and Saaltink, M. (1991) EVES: An

overview. In: Prehn, S. and Toetenel, H. (eds), *VDM'91, Formal software development methods: Lecture Notes in Computer Science 551*, pp. 389–405. Springer-Verlag.

Davis, M. and Schwartz, J. T. (1979) Metamathematical extensibility for theorem verifiers and proof-checkers. *Computers & Mathematics with Applic.*, **5**, 217–230.

Gordon, M. J., Milner, A. J. and Wadsworth, C. P. (1979) *Edinburgh LCF: A mechanised logic of computation: Lecture Notes in Computer Science 78.* Springer-Verlag.

Gordon, M. J. C., & Melham, T. F. (eds). (1993). *Introduction to HOL: A theorem proving environment for higher order logic.* Cambridge University Press.

Huet, G. (1989) The constructive engine. In: Narasimhan, R. (ed), *Perspectives in theoretical computer science. Commemorative volume for Gijt Siromoney.* World Scientific. (Also in *The Calculus of Constructions. Documentation and User's Guide. Version 4.10.* Technical Report RT-0110, INRIA, France, 1989.)

Jackson, P. B. (1995) *Enhancing the Nuprl proof development system and applying it to computational abstract algebra.* PhD thesis, Department of Computer Science, Cornell University, Ithaca, New York. (Technical Report TR 95-1509.)

Kromodimoeljo, S. and Pase, W. (1992) *Final report for the investigation of proof techniques within the Eves verification technology.* Final Report FR-92-5451-02, ORA Canada, 267 Richmond Road, Suite 100, Ottawa, Ontario K1Z 6X3, Canada. ftp://ftp.ora.on.ca/pub/doc/92-5451-02.ps.Z.

Milner, R., Tofte, M. and Harper, R. (1990) *The Definition of Standard ML.* MIT Press.

Nipkow, T. (1989) Equational reasoning in Isabelle. *Science of Computer Programming*, **12**, 123–149.

Paulson, L. (1983) A higher-order implementation of rewriting. *Science of Computer Programming*, **3**, 119–149.

Paulson, L. C. (1994) *Isabelle: A generic theorem prover: Lecture Notes in Computer Science 828.* Springer-Verlag.

Slind, K. (1991) *An implementation of higher order logic.* Masters thesis, Department of Computer Science, The University of Calgary, 2500 University Drive NW, Calgary, Alberta, Canada T2N 1N4. (Research Report 91/419/03.)

Slind, K. (1992) Adding new rules to an LCF-style logic implementation. In: Claesen, L. J. M. and Gordon, M. J. C. (eds), *Higher order logic theorem proving and its applications: Proceedings of the IFIP TC10/WG10.2 workshop*, pp. 549–559. Leuven, Belgium.

Wadler, P. (1992) The essence of functional programming. *Conference Record of the 19th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 1–14, Albuquerque, NM.

Welinder, M. (1995) Very efficient conversions. In: Schubert, E. T., Windley, P. J. and Alves-Foss, J. (eds), *Proc. 8th International Workshop on Higher Order Logic Theorem Proving and its Applications: Lecture Notes in Computer Science 971.* Aspen Grove, UT. Springer-Verlag.

Wong, W. (1995) Recording and checking HOL proofs. In: Schubert, E. T., Windley, P. J. and Alves-Foss, J. (eds), *Proc. 8th International Workshop on Higher Order Logic Theorem Proving and its Applications: Lecture Notes in Computer Science 971.* Aspen Grove, UT. Springer-Verlag.