Constructive Rewriting

T. NIPKOW
Computer Laboratory, University of Cambridge, Pembroke Street, Cambridge CB2 3QG, UK

The subject of this paper is rewriting in an LCF-like general theorem proving framework. It is shown how rewriting of both terms and formulae can be implemented by simple tactics in the generic theorem prover Isabelle. These tactics can easily be combined with induction to yield powerful theorem proving primitives. As a sample application the verification of an n-bit ripple-carry adder is demonstrated.

Received January 1989

1. INTRODUCTION

At the heart of all systems concerned with equational reasoning (and many theorem provers in general) we find a 'rewrite engine', i.e. some module which rewrites or normalises terms w.r.t. some set of rules. In most implementations this rewrite engine has two characteristics: it operates directly on the internal representation of terms, and it is not easily extensible. Consequently the correctness of rewriting depends on the correctness of this code and needs to be re-established for every extension.

An alternative approach to rewriting proceeds via explicit invocation of equational logic inference rules: rewriting a term s to a term t is achieved by constructing a proof of $s = t$. Hence the title of this paper. To enforce this style of rewriting, equations are abstract data types whose values can only be generated by composition of equational logic inference rules. Provided that all rules of inference and their composition mechanism are implemented correctly (and provided the abstract data type mechanism of the implementation language cannot be subverted), this implies correctness of all derived theorems. Thus any extension by new proof procedures is admissible since the theorems they produce are by definition correct. The advantage of this approach is guaranteed soundness, its disadvantage is a possibly serious loss in efficiency.

The above discussion can be generalised from rewriting to theorem proving. In particular one can categorise theorem provers according to the mechanisms they offer for introducing new proof procedures. Interestingly enough, none of the equational logic based systems like REVE,[18] LP,[7] RRL[14] or RAP[8] offer facilities for sound user-level extensions. The reason is that they were originally designed to solve special problems efficiently and not so much with a general theorem proving perspective. Systems that do allow sound extensions use one of two approaches:

1. New proof procedures must be verified in a formalised meta logic. If the object logic is strong enough, this verification may even be carried out *within* the system. In the Boyer–Moore system[2] this is possible because both levels are expressed in terms of Lisp, in Nuprl[17] the rich type theory allows this reflection.
2. Theorems are abstract data types whose generators are the primitive rules of the logic. Proof procedures are so called *tactics* which construct inferences by reduction to the primitive rules. Hence any tactic is sound by definition. This concept goes back to LCF[9, 26] and is also used in Isabelle,[27] Nuprl[5] and HOL.[10]

* This research was supported in part by NYNEX, NSF grant CCR-8706652, and by the Advanced Research Projects Agency of the DoD, monitored by the ONR under contract N00014-83-K-0125.

This paper is dedicated to the second approach. First it is shown how term rewriting tactics can be derived from the laws of equational logic. In order to turn these tactics into general theorem proving procedures they are lifted to predicate logic, and integrated with inductive theorem proving. Finally a simple ripple–carry adder is verified with the resulting tactics. The whole development is carried out within the Isabelle system but the principles are equally applicable to other LCF-like systems. In fact a similar study done in LCF is reported in Ref. 24.

The structure of the paper is as follows. Section 2 introduces the generic theorem prover Isabelle. Section 3 explains in some detail how various rewriting techniques can be implemented as tactics for first-order logic. Section 4 presents the user-interface of rewriting and induction tactics which are used in section 5 to verify a ripple–carry adder.

A word concerning the typographic conventions: for most formulae ordinary mathematical notation in math-italics is used. We resort to `typewriter` style only for ML code and Isabelle I/O.

2. ISABELLE

Isabelle is an interactive theorem prover developed and implemented in ML by Larry Paulson at the University of Cambridge. This section gives only a very sketchy account of Isabelle, just enough to make the paper self-contained. A more detailed introduction can be found for example in Refs. 11 and 21. A first explanation of the principles underlying Isabelle is contained in Ref. 25, a formalisation of Isabelle's meta–logic using higher-order logic is given in Ref. 27. The version of Isabelle described in this paper is Isabelle-86. Meanwhile Isabelle has been extended significantly and Ref. 27 pertains to the latest version.

2.1. Representing Logics

What distinguishes Isabelle from most other theorem provers is the fact that it can be parameterised by the object-logic to be used. The definition of a logic consists of the declaration/definition of all

- basic types (for example terms, formulae, etc.);

- logical constants (operators like $=$, $\Rightarrow$ and $\forall$) with their arity; valid arities are the basic types and function types over them;
- inference rules.

The central notion in Isabelle is that of a *rule*, written as

$$\frac{P_1, \dots, P_m}{P},$$

where the P_i and P are simply-typed λ-calculus terms over the logical constants and variables. The P_i are called the *premises* and P the *conclusion*. If $m = 0$, the rule is called *theorem* and the horizontal line is omitted.

In this paper we use an OBJ-like syntax (Ref. 6) to present logics. A simple example is:

```
EQLog=SORTS term, form
      OPS_=_: term*term→form
      RULES
```

$$\mathtt{x=x} \qquad \frac{\mathtt{x=y}}{\mathtt{y=x}} \qquad \frac{\mathtt{x=y} \quad \mathtt{y=z}}{\mathtt{x=z}} \qquad \frac{\mathtt{P(x)} \quad \mathtt{x=y}}{\mathtt{P(y)}}$$

Equational logic, a fragment of first-order predicate calculus, is based on the two basic types `term` and `form` of terms and formulae. The only logical constant is $=$ of type `term*term→form`, which is equivalent to `term→ term→form`. The four inference rules of equational logic are reflexivity, symmetry, transitivity and congruence. Notice that x, y, and z are variables of type `term`, whereas P is of type `term→form`.

In the rest of the paper OBJ's modularisation facilities are used to present logics incrementally. This feature is not available in Isabelle-86 and was only introduced in later versions.

2.2. Theorem Proving

Theorem proving in Isabelle amounts to combining the basic rules to form derived rules. The principal method for combining two rules is *resolution*: given two rules

$$p = \frac{P_1, \dots, P_m}{P} \quad \text{and} \quad q = \frac{Q_1, \dots, Q_n}{Q},$$

and a substitution σ which unifies P with Q_i for some i, resolving p and q yields the new rule

$$\sigma\left(\frac{Q_1, \dots, Q_{i-1}, P_1, \dots, P_m, Q_{i+1}, \dots, Q_n}{Q}\right) \qquad (1)$$

To support resolution, Isabelle is based on unification rather than just matching (as for example LCF). Since Isabelle formulae are λ-terms, Isabelle contains an implementation of higher-order unification which is described in Ref. 25. This means that unification may yield a potentially infinite stream of unifiers; it may even be undecidable. Fortunately, this turns out not to be a problem in practice, in particular if all terms are first-order.

Isabelle provides two kinds of variables: ordinary and *logical* variables. The latter can be instantiated during the resolution process whereas the former act like constants. Logical variables are distinguished from ordinary ones by being prefixed with a '?'. For readability reasons we have omitted most of the ?'s. In the sequel we follow the convention that, unless noted otherwise, the letters P and Q and u through z denote logical variables, other letters stand for arbitrary expressions.

All this sounds very much like logic programming, and in fact Isabelle can be seen as an implementation of typed higher-order logic programming.[25]

In Isabelle the state of a proof is just a rule, where the premises should be thought of as the goals to be solved, and the conclusion the formula to be proved. The proof of some formula R starts with the trivially correct rule $\frac{R}{R}$ and seeks to transform it into $\frac{}{R}$ by successive resolution with other rules. Due to this backwards style of theorem proving inference rules should be read as transformations which replace the conclusion with the premises. In order to automate this tedious process, algorithmic sequences of rule applications can be coded as ML functions which are known as *tactics*. Tactics are a concept originating with LCF.[9, 26] They are the functional programmer's answer to the challenge posed by the length and repetitiveness of proofs from first principles.

An Isabelle tactic is a function of type `tactic= rule→rule sequence`, where `sequence` is an abstract type implementing lazy lists. Tactics need to produce sequences of rules to allow for backtracking and also because resolution may produce an infinite number of results due to higher-order unification. The most basic Isabelle tactic performs resolution:

```
>val res_tac=fn: rule list→tactic
```

Applying `res_tac rl` to some rule `r` yields the stream of resolvents of rules in `rl` with the first premise of `r`. There is a corresponding infix operator `RES: rule* rule→rule`; given two rules `p` and `q`, `q RES p` yields the result (provided it is unique) of resolving `p` with the first premise of `q` as in (1) but with $i = 1$.

Although all derived rules are ultimately proved via single resolution steps, Isabelle provides tacticals (functions for combining tactics) to build up complex proof strategies. The basic ones perform sequencing, alternation and repetition: `tac1 THEN tac2` applies `tac1` and then `tac2` to the result; `tac1 ORELSE tac2` applies `tac1` or, if that fails (returns the empty sequence), `tac2`; `REPEAT tac` applies `tac` until it fails. In addition there is the basic tactic `all_tac` which is the identity element w.r.t. `THEN` because it maps any rule to the singleton sequence containing just that rule. A precise definition of the functionality of these and other tacticals can be found in Refs. 28 and 21. For the understanding of the simple tactics in this paper the above intuitive explanation should suffice.

3. REWRITING

We will now show, first for equational logic and then for full predicate logic, how rewriting can be reduced to the basic laws of a logic. We assume that the ML identifiers `refl`, `trans` and `cong` are bound to the corresponding rules of equational logic as defined in Section 2.1.

3.1. Term Rewriting

Traditionally, term rewriting is seen as a process which takes a term s and produces some equivalent term t. In an LCF-like theorem proving context that is not enough: we also need a proof of the fact that $s = t$. In Ref. 24 Paulson introduces the type of *conversions* which are functions from a term s to a theorem $\vdash s = t$. Then he shows that conversions behave a lot like tactics and presents combinators for them which mirror the ones for tactics: sequencing, alternation, repetition. In Isabelle we can achieve the same elegance by equating conversions with certain special tactics. In the sequel a conversion is a tactic which performs the following transformation:

$$\frac{s = r \quad H}{R} \Rightarrow \frac{t = r \quad H}{R} \tag{2}$$

Here s, t and r are terms, H is a list of formulae and R a formula. In particular we assume that the transformation from s to t is insensitive to what r, H and R look like. Therefore we often identify a conversion with the transformation from s to t it achieves.

The simplest conversions are the ones that can be extracted from rewrite rules, i.e. rules of the form $l = r$. The function

```
fun mk_trans rule=trans RES rule;
```

maps any rule $l = r$ to $\frac{r = z}{l = z}$. If this rule is resolved with the first premise of the left-hand side of (2), and s is an instance of l, the result is the right-hand side, where t is the corresponding instance of r. An example should make this clear.

Example 1. If `s_rew` is the rule `s(?x)+?y=s(?x+?y)`, `mk_trans s_rew` yields $\frac{\texttt{s(?x+?y)=?z}}{\texttt{s(?x)+?y=?z}}$. Applying the conversion `res_tac [mk_trans s_rew]` to the left-hand side we obtain the right-hand side of the following transformation:

$$\frac{\texttt{s(0)+0=r} \quad \texttt{H}}{\texttt{R}} \Rightarrow \frac{\texttt{s(0+0)=r} \quad \texttt{H}}{\texttt{R}}$$

The conversions extracted from rewrite rules are basic building blocks which can be combined by tacticals: given two conversions `conv1` and `conv2` which transform r to s and s to t respectively, `conv1 THEN conv2` transforms r to t. Similarly a conversion `conv` can be applied until it fails by `REPEAT conv`. A perfect example of this style of combining conversions is a tactic for normalising a term completely w.r.t. some conversion. In order to apply conversions not just to the root of a term (as the ones we have seen so far do) but also to its subterms, we assume the existence of a tactical `ALL_SUBTS` which maps a conversion to a conversion: if `conv` transforms each s_i to t_i, `ALL_SUBTS conv` transforms a term $f(s_1, \ldots, s_n)$ to $f(t_1, \ldots, t_n)$. Thus we obtain the following simple definition of a bottom-up normalisation tactic:

```
fun BU conv rule=
  (ALL_SUBTS(BU conv) THEN ((conv THEN
    BU conv) ORELSE all_tac)) rule;
```

This can be paraphrased as: First convert all subterms. If the resulting term can be converted, start the conversion process again; otherwise return the result.

`BU` is practically identical to `REDEPTH_CONV` in Ref. 24, except that the former is expressed in terms of tactics and tacticals instead of the special type of conversions with their own combinators.

It should be emphasised that `BU` is not a conversion but a particular term traversal strategy. Many other strategies are possible and Ref 24 gives an example of a more top-down oriented one which could also be translated almost literally. To be truly general one might introduce a parameter determining the reduction strategy in many of the tactics to come. I have chosen to follow the implementation and use `BU` throughout the paper.

We still have to explain the working of `ALL_SUBTS conv`. It relies on congruence rules of the form

$$\frac{x_1 = y_1, \ldots, x_n = y_n}{f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n)}$$

which can be obtained by composing n instances of the general rule `cong`:

$$\cfrac{\cfrac{\cfrac{f(x_1, \ldots, x_n) = f(x_1, \ldots, x_n) \quad x_1 = y_1}{f(x_1, \ldots, x_n) = f(y_1, \ldots, x_n) \quad x_2 = y_2}}{\ddots \qquad\qquad x_n = y_n}}{f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n)} \tag{3}$$

Let this rule be called `fcong`. Resolving `mk_trans fcong` with some premise $f(s_1, \ldots, s_n) = r$ yields the premises $s_1 = y_1, \ldots, s_n = y_n f(y_1, \ldots, y_n) = r$. The first n of these are solved by `conv THEN res_tac [refl]`, instantiating y_i with t_i and leaving the premise $f(t_1, \ldots, t_n) = r$. The exact definition of this function can be found in Ref. 21.

3.2. Formula Rewriting

Now it is time to generalise from term rewriting to formula rewriting. The syntax of our language is that of ordinary predicate logic with the connectives $\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow, \forall$ and the constants T and F. In Isabelle they are written as `~`, `&`, `|`, `=>`, `<=>`, `ALL`. The inference rules used below are derived rules in some suitable axiomatisation of predicate logic.

Due to careful design, the above ideas and tactics are general enough to deal with both term and formula rewriting. The key is the simple observation that $\Leftrightarrow$ is a congruence relation on formulae just as $=$ is one on terms. Thus it suffices to provide both kinds of reflexivity, transitivity, symmetry and congruence rules wherever either might be required. The result are conversion tactics which transform the left-hand sides of both equalities and bi-implications. What remains to be done is to turn them into tactics which simplify whole formulae. In the sequel let `iff_refl` be the rule `?P<=>?P`. Using the rule

$$\texttt{iff\_elim} = \frac{\texttt{?P} \Leftrightarrow \texttt{?Q} \quad \texttt{?Q}}{\texttt{?P}}$$

we define the formula normalisation tactical `NORM`:

```
fun NORM conv=res_tac [iff_elim] THEN
  conv THEN res_tac [iff_refl];
```

Applying `NORM conv` to some rule with first premise R we can observe 3 distinct stages. Resolution with `iff_elim` replaces R by the two formulae $R \Leftrightarrow ?Q$ and ?R. Then `conv` rewrites the first premise to $S \Leftrightarrow ?Q$ for some formula S. Finally resolution with `iff_refl` removes the first premise while instantiating $?Q$ with S, thus leaving S as the new first premise.

The complete translation from a list of rewrite rules `rwrls` of the form $l = r$ or $P \Leftrightarrow Q$ to a simplification tactic for formulae can now be expressed as

```
fun SIMP_TAC rwrls=NORM(BU(res_tac
  (map mk_trans rwrls)));
```

In order to complete the picture we need to explain how term rewriting arises as a subtask of formula rewriting. The transition is achieved by the following congruence-like rule:

$$\frac{x = u \quad y = v}{x = y \Leftrightarrow u = v} \tag{4}$$

Together with the proper congruence rules it is used in `ALL_SUBTS` to turn the task of rewriting a formula $s = t$ into two term rewriting tasks for s and t. If s and t have the same normal form, say r, then $s = t$ has normal form $r = r$.

Ideally, simplification should rewrite valid formulae to T, an immediately solvable goal because it is a predicate calculus axiom (stored in the ML identifier `T`). If we also supply the rewrite rule $x = x \Leftrightarrow T$, $r = r$ above can be rewritten one more time to T. In general we have the following tactics for proving the first premise by rewriting:

```
fun PROVE conv=NORM conv THEN
  res_tac_[T];
```

This concludes the description of the unified machinery for term and formula rewriting. In the sequel all discussions about equality are phrased in terms of $=$ but apply to any congruence, especially $\Leftrightarrow$.

3.3. Extensions

There are 3 important extensions to the basic rewriting methodology presented in the previous sections: conditional rewriting, rewriting with assumptions, and rewriting modulo equations, all of which are presented in some detail in Ref. 21.

Conditional rewriting allows rewriting with implications $R \Rightarrow S$, where S is an equality between terms or formulae. This rule is applicable if R can be proved. This leads to a short but highly recursive function for conditional rewriting involving the basic tactics `BU` and `PROVE`.

Rewriting with assumptions is based on a sequent calculus formalisation of predicate logic as for example in Ref. 26. Formulae in this logic are of the form $\Gamma \vdash R$, where Γ is a list of formulae (the assumption) and R a formula (the conclusion) as we know them. Any rewrite rule among the assumptions may be used to simplify the conclusion. This extension is of particular interest for rewriting formulae: in rewriting an implication $R \Rightarrow S$ one may assume R (or its normalised form) while rewriting S. The justification is the derived sequent calculus rule

$$\frac{\Gamma \vdash P \Leftrightarrow P' \quad \Gamma, P' \vdash Q \Leftrightarrow Q'}{\Gamma \vdash (P \Rightarrow Q) \Leftrightarrow (P' \Rightarrow Q')}$$

Although rewriting with 'local assumptions' is most important for $\Rightarrow$, similar rules hold for conjunctions and disjunction.

Both conditional rewriting and rewriting with assumptions is subtle in some details but on the whole straightforward. The interested reader is referred to Ref. 21. It is less obvious how to achieve rewriting modulo equations in an LCF-like framework, which is why we look at it more closely.

The general idea of rewriting modulo equations is to build certain equational theories into the rewriting engine by providing special purpose matching algorithms for them. These algorithms are taken into account when matching the left-hand sides of rules and the term to be rewritten. This is motivated by troublesome axioms like commutativity which cannot be dealt with by ordinary rewriting.

This leaves us with the question of how to do equational matching in Isabelle. More generally, we are looking for a framework for describing unification algorithms by proof rules. For a particular case, unification in the empty theory, an answer was given by Martelli–Montanary:[19] a unification problem is a set of equations which are solved by repeated application of some fixed set of transformation rules. This idea has been generalised to arbitrary equational theories in the work of Claude Kirchner, for example Ref. 15. In our context the equations to be solved are the premises of some rule, and their transformation is achieved by resolution with rules of the form

$$\frac{s_1 = t_1, \ldots, s_n = t_n}{s = t}$$

which are also called *decomposition* rules. These rules should be read like Prolog clauses whose procedural interpretation says: solving $s = t$ can be achieved by solving all $s_i = t_i$. In order to make this process terminate, the equational problem should be simplified by each resolution with a decomposition rule.

Equations of the form $x = t$ can be solved directly by reflexivity, instantiating x with t. Notice that reflexivity fails if x occurs in t (occur-check!). Hence this principle is only adequate for matching problems (where t must be ground) and for unification in so called *simple* theories, i.e. theories where the above equation does not have a solution. The theories discussed in this section happen to be simple.

A second observation is that all congruence rules of the form (3) are decomposition rules and do not jeopardise termination.

Thus a general unification tactic is obtained: equations of the form $x = t$ are solved by reflexivity, others are replaced by a set of simpler equations by resolution with some congruence or theory-specific decomposition rule. This process continues until all equations have been solved. The variable bindings created on the way constitute a unifier. This is a nondeterministic algorithm and may involve much backtracking! A precise formulation as an Isabelle tactic can be found in Ref. 21.

Within this framework the search for a unification algorithm is reduced to finding a suitable set of decomposition rules. Suitable means complete and terminating. Of course this is the really hard bit. Fortunately, for a number of frequently used equational theories, suitable decomposition rules can be found.

The simplest case is the empty theory where the congruence rules alone suffice. Of course Isabelle's built-in higher order unification subsumes first-order unification in the empty theory. Decomposition rules for a number of practically relevant theories are listed below.

Commutativity:

$$\frac{x = v \quad y = u}{x+y = u+v} \tag{5}$$

Associativity:

$$\frac{x = u+w \quad w+y = v}{x+y = u+v} \quad \frac{x+w = u \quad y = w+v}{x+y = u+v} \tag{6}$$

Associativity + Commutativity: rules (5), (6), and

$$\frac{x = v+w \quad w+y = u}{x+y = u+v} \qquad \frac{x+w = v \quad y = w+u}{x+y = u+v}$$

$$\frac{x = w_1+w_2 \quad y = z_1+z_2 \quad w_1+z_1 = u \quad w_2+z_2 = v}{x+y = u+v}$$

Right-Commutativity $((x+y)+z = (x+z)+y)$:

$$\frac{x = w+v \quad w+y = u}{x+y = u+v}$$

The decomposition rule for commutativity can be found in Ref. 15, the one for right-commutativity in Ref. 16. The inference rules for all four theories yield a complete unification algorithm, but only the one for commutativity is guaranteed to terminate. Fortunately, they all terminate if applied to matching problems, which is sufficient for our purposes.

Examples of other equational theories which admit this treatment can be found in Refs 15 and 20.

3.4. A Conditional

Isabelle's higher-order features allow a rewriting-oriented treatment of conditionals. As part of the definition of predicate logic we have declared a constant *if*: *form* × *term* × *term* → *term* and added the rule

$$P(if(Q,x,y)) \Leftrightarrow (Q \Rightarrow P(x)) \wedge (\neg Q \Rightarrow P(y))$$

Note that P is of type *term→form*.

This rule provides a simple way of automating case distinctions. Unfortunately, it has to be treated separately from the first order rewrite rules considered so far. The reason is that the left-hand side matches *any* formula: given a formula R which does not contain an *if*-subterm, higher-order unification will instantiate P by $\lambda u.R$, where u does not occur free in R, i.e. P becomes a constant function. Therefore this rule is connected with a special applicability check. The implementation is straightforward and is not shown here.

There is a second, more practical, reason for separating the expansion of conditionals: while the standard rewriting tactic proceeds bottom-up, conditionals are best expanded in a top-down fashion. Consider some formula $R \wedge S(if(C,x,y))$. Expanding the conditional at the innermost point produces $R \wedge ((C \Rightarrow S(x)) \wedge (\neg C \Rightarrow S(y)))$, at the outermost point it yields $(C \Rightarrow (R \wedge S(x))) \wedge (\neg C \Rightarrow (R \wedge S(y)))$. Now consider what happens if both formulae are further simplified using rewriting with local assumptions. In the first case only $S(x)$ $(S(y))$ can be rewritten under the additional assumption $C(\neg C)$, whereas in the second case $C(\neg C)$ is also available when rewriting R.

Defining functions via *if* and using (7) as a rewrite rule turned out to be an important factor in automating many proofs.

3.5. In Practice

The tactics presented above are not very efficient. Fortunately there are two simple optimisations which improves their performance significantly. The first one is the use of an efficient data structure for rule selection. If the list of rewrite rules `rwrls` grows long, `res_tac rwrls` may take a long time to find a matching one. Isabelle provides a data type for fast rule selection and the required changes to the rewriting tactics are minimal.

The second improvement is on the logical level and is connected with bottom-up rewriting. Rewriting a term t at the root according to a rule $l = r$ means that t is of the form $\sigma(l)$ and is rewritten to $\sigma(r)$. If bottom-up rewriting is used, all subterms of t must have been in normal form, therefore all terms in the range of σ, i.e. all variable instances in $\sigma(r)$ are in normal form too. Unfortunately, `BU` as defined above does not realise this and, in rewriting $\sigma(r)$, visits all of its subterms. A simple solution to this problem requires an extension of the logical system by a new constant of type *term→term* which is used to indicate that a subterm is in normal form. Let us call this constant N. The only new axiom is that N is the identity function i.e. $N(x) = x$. Now all variables in r are tagged with N and `BU` does not need to descend into subterms labelled with N. Let us look at an example:

Example 2. Tagging the right-hand side of the rule $s(x)+y = s(x+y)$ turns it into $s(x)+y = s(N(x)+N(y))$ (using $x = N(x)$). Applying the new rule to $s(t_1)+t_2$ (where t_1 and t_2 are already in normal form) results in $s(N(t_1)+N(t_2))$. Bottom-up normalisation of this term has does not look at t_1 or t_2 but simply removes the N's (using $N(x) = x$) and immediately tries to rewrite t_1+t_2 at the root.

Unfortunately this optimisation is not compatible with rewriting modulo equations. Given the rule $x+x+y = y$ where $+$ is AC, the term $(a+b)+(a+b+c)$ rewrites to $b+b+c$. The latter is not in normal form although all subterms of the former were.

The question remains how efficient the resulting tactics are. The following data was obtained with Dave Matthews' Poly/ML system on a Vax 3600 and includes garbage collection. Simple unconditional rewriting proceeds at a rate of about 5 reductions per second. The speed of AC-rewriting depends largely on the size of the terms and the number of AC-operators in them. For small terms it is typically an order of magnitude slower.

4. TACTICS

Having studied the anatomy of rewriting, we consider the user-interface level. There are two rewriting tactics which combine all the features introduced in Section 3:

```
SIMP_TAC: rule list→tactic
EQ_SIMP_TAC: tactic→rule list→tactic
```

Both take a list of rewrite rules but `EQ_SIMP_TAC` is also supplied with a matching tactic as described above. Although `SIMP_TAC` is subsumed by `EQ_SIMP_TAC`, the former is significantly more efficient than the latter. Therefore both are offered. In addition to the rewrite rules given as an argument both tactics employ a built-in set of rules for formula simplification. This set contains simple rules like $P \wedge T \Leftrightarrow P$ and $(\forall x.T) \Leftrightarrow T$, but is by no means complete.

Rewriting on its own is not very powerful. Most nontrivial proofs rely on some kind of induction. In Isabelle induction is just another proof rule, for example

$$\frac{P(0) \quad \forall x.P(x) \Rightarrow P(x+1)}{\forall x.P(x)}$$

P is of higher type, namely *term→form*. Resolution with this rule depends on higher-order unification for instantiating P. The design of a general induction tactic is discussed in Ref. 21. A slightly revised version is

```
IND_TAC: tactic→rule→string→tactic
```

which takes a (simplification) tactic, an induction schema, and the name of the variable to induct on. It applies the induction rule and tries to solve as many subgoals as possible by simplification.

The combination of induction and rewriting is sufficient to prove all theorems in Section 5.

5. AN EXAMPLE

To lend some credibility to the claim that the tactics of the preceding sections do constitute useful theorem proving primitives, we look at a simple example of hardware verification: a ripple-carry adder. In order to express the correctness of the hardware we use the following specification of natural numbers as a reference model:

```
Nat=Predicate_Logic+
     SORTS nat
     OPS 0, 1, 2: nat
        s: nat→nat
        _+_, _−_, _^_, _//_: nat*nat→nat
        _<_: nat*nat→form
     RULES
        ...
        2^0=1
        2^s(n)=2^n+2^n
        n<m⇒n//m=n
   ~    n<m⇒n//m=(n−m)//m
        P(0) & (ALL n.P(n)⇒P(s(n)))
          ⇒All n.P(n)
```

The primitive recursive definitions of +, − and < have been omitted and can be found in any book on algebraic specifications, e.g. Ref. 23. m^n denotes m^n, although we have only bothered to axiomatise 2^n, and n//m stands for *n mod m*. The last formula is the induction principle for natural numbers.

Figure 1 shows the circuit of a 1 bit full adder built from exor (#) and nand gates (~&).

Modelling hardware in predicate logic is easy: the hardware states 1 and 0 are identified with the truth-values T and F, gates are identified with logical connectives. Thus we arrive at the following faithful representation of the full adder in propositional logic;

```
FA=Predicate_Logic+
   OPS _#_, _~&_: form*form→form
       sum, carry: form*form*form→form
   RULES x#y⇔~(x ⇔ y)
         x ~& y⇔~(x & y)
         sum(a, b, c)⇔(a#b)#c
         carry(a, b, c)
           ⇔((a#b) ~& c) ~& (a ~& b)
```

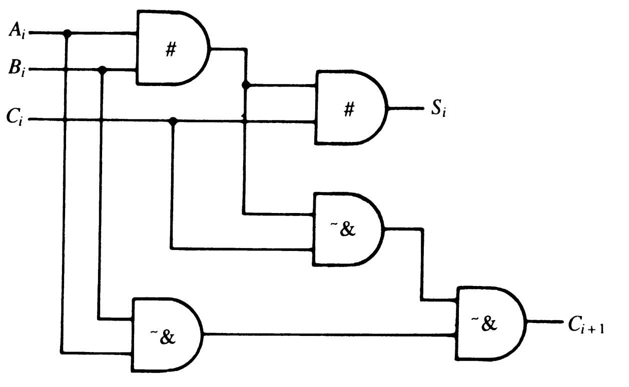

Figure 1. A 1 bit *Full Adder* (FA).

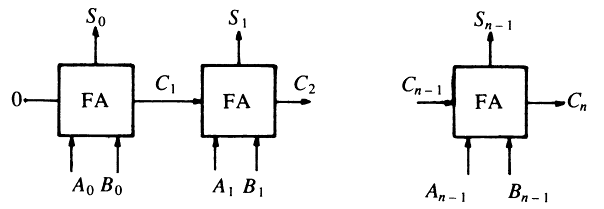

Figure 2. An *n* bit *ripple-carry adder*.

An n-bit ripple-carry adder is a simple cascade of n full adders as shown in figure 2. In order to model bit-vectors of arbitrary length we simply consider them as functions from natural numbers to bits. Thus we arrive at the following definitions:

```
Adder=FA+Nat+
        SORTS bv=nat→form
        OPS add, oflow: bv*bv→bv
            bin: bv*nat→nat
        RULES
              add(A, B, n)
                ⇔sum(A(n), B(n),
              oflow(A, B, n))
              oflow(A, B, 0)⇔F
              oflow(A, B, s(n))
                ⇔carry(A(n), B(n),
               oflow(A, B, n))
              bin(A, 0)=0
              bin(A,s(n))=if(A(n), 2^n, 0)
                 +bin (A, n)
```

The functions add and oflow correspond to the S and C outputs in figure 2. It is easy to see that Adder is a correct translation of the circuit diagram into logic. Note that the syntax for function application is uncurried: instead of add(A, B) (n) we write add(A, B, n).

If we want to prove the adder correct, the question arises what that means. In the language of abstract data types bit-vectors with add are a concrete realisation of natural numbers with +. To show the correctness of this

$$\begin{tikzcd}
\mathtt{nat} & (.,.) \arrow[r, "+"] & \bullet \\
\mathtt{bv} & (.,.) \arrow[u, "\mathtt{bin}"] \arrow[r, "\mathtt{add}"'] & \bullet \arrow[u, "\mathtt{bin}"']
\end{tikzcd}$$

Figure 3. Relating bv **and** nat.

implementation we need to exhibit a homomorphism (an 'abstraction function' in the terminology of Ref. 12) from bv to nat. This missing link is the function bin ('bv into nat'). The required relationship between bv and nat is expressed pictorially by the commuting diagram in figure 3. Since n-bit arithmetic provides a correct implementation of natural numbers only modulo 2^n, bin has a second parameter which determines the number of bits to convert. The correctness assertion is stated by the following two formulae:

```
bin(add(A, B), n)=(bin(A, n)
  +bin(B, n)) // 2^n                    (8)
oflow(A, B, n)<=>~bin(A, n)
  +bin(B, n)<2^n                        (9)
```

Equation (8) expresses that the diagram in figure 3 commutes modulo 2^n, (9) says that the nth carry bit is on if and only if the n-bit sum of the two arguments is not less than 2^n. Let us now look at the proof of both formulae.

It turns out that almost all effort goes into constructing a lemma library for Nat. As a first step it is shown that + is associative and commutative. This proof is detailed e.g. in Ref. 21. Now we can use AC-rewriting as explained in Section 3.3 for the rest of the proofs. The complete list of lemmas is shown below.

```
x<y=>x<s(y)
y<z=>x+y<z
x<u & y<v=>x+y<u+v
~x+y<x
x+y<x+z<=>y<z
x-x=0
(x+y)-y=x
(x+y)-(x+z)=y-z
~y<z=>x+(y-z)=(x+y)-z
~z=0=>x-y<z<=>x<z+y
x+y=0<=>x=0 & y=0
~2^n=0
bin(A, n)<2^n
```

They constitute a terminating set of (conditional) rewrite rules.

To give the reader a better impression of how these proofs were carried out we look at a particularly simple example. The following two lines are input to the Isabelle system, i.e. they are calls of ML functions in the interface to the system.

```
goal 'x-x=0';
by(IND_TAC N_IND (SIMP_TAC RWLS) 'x');
```

The function goal establishes the formula x−x=0 as the current goal, by takes a tactic and applies it to the current goal. In this case the tactic is an induction over x. The ML identifier N_IND holds the induction schema in the definition of Nat and rwrls contains the current list of rewrite rules. In this example a single induction suffices. In many of the proofs one or two additional case distinctions are required. A more detailed exposition of this style of theorem proving can be found in Ref. 22.

Having built up the lemma library above, it may come as a bit of a surprise that the proofs of (9) and (8) (in that order!) go through with a single induction on n. Analysing the proofs in detail one discovers that, apart from the proper choice of lemmas, this is largely due to automatic case splits caused by the use of if in the definition of bin.

The above correctness proof is only a first step towards verified hardware. More ambitious efforts are reported in Refs. 4 and 13 which describe the correctness proofs of two microprocessors in HOL and the Boyer–Moore system respectively. In those proofs the adder is just one subproblem among many.

Acknowledgements

This research would have been impossible without Larry Paulson's Isabelle system. Special thanks are due to John Guttag who provided the stimulating environment.

REFERENCES

1. G. Birtwistle and P. A. Subramanyam (eds), *VLSI Specification, Verification and Synthesis: Proceedings of the Calgary Workshop*, Kluwer Academic Publishers (1987).
2. R. S. Boyer and J. S. Moore, *Metafunctions: Proving Them Correct and Using Them Efficiently as New Proof Procedures.* In Ref. 3.
3. R. S. Boyer and J. S. Moore (eds), *The Correctness Problem in Computer Science*, Academic Press (1981).
4. A. J. Cohn, *A Proof of Correctness of the Viper Microprocessor: the First Level.* In Ref. 1.
5. R. L. Constable *et al. Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall (1986).
6. K. Futatsugi, J. A. Goguen, J.-P. Jouannaud and J. Meseguer, Principles of OBJ2, *Proceedings 12th POPL* pp. 52–66 (1985).
7. S. J. Garland and J. V. Guttag, *An Overview of LP, The Larch Prover. Proceedings RTA-89*, Lecture Notes in Computer Science. Springer, 355, pp. 137–151 (1989).
8. A. Geser and H. Hussmann, *Experience with the RAP System – a Specification Interpreter Combining Term Rewriting and Resolution. Proceedings ESOP 86*, Lecture Notes in Computer Science. Springer, Heidelberg, **213**, pp. 339–350 (1986).
9. M. J. C. Gordon, R. Milner and C. P. Wadsworth, *Edinburgh LCF: A Mechanised Logic of Computation.* Lecture Notes in Computer Science. Springer Verlag, **78** (1979).
10. M. J. C. Gordon, *HOL: A Proof Generating System for Higher-Order Logic.* In Ref. 1.
11. Ph. de Groote, *How I Spent my Time in Cambridge with Isabelle.* Report RR 87-1, Unité d'Informatique, Université Catholoque de Louvain, Belgium (1987).
12. C. A. R. Hoare, Proof of Correctness of Data Representations, *Acta Informatica* **1** (1972).
13. W. A. Hunt, *FM8501: A Verified Microprocessor.* Ph.D. Thesis, University of Texas at Austin (1985).
14. D. Kapur and H. Zhang, RRL: A Rewrite Rule Laboratory, *Proceedings CADE-9*, Lecture Notes in Computer Science. Springer Verlag, **310**, 768–769 (1988).
15. C. Kirchner, *Computing Unification Algorithms.* Proceedings Symposium on Logic in Computer Science, Cambridge, MA, pp. 206–216 (1986).
16. C. Kirchner, Méthodes et outils de conception systématique d'algorithmes d'unification dans les théories équationnelles, Thèse d'état de l'Université de Nancy I (1985).
17. D. J. Howe, Computational Metatheory in Nuprl. *Proceedings CADE-9*, Lecture Notes in Computer Science. Springer Verlag, **310**, pp. 238–257 (1988).
18. P. Lescanne, REVE: A Rewrite Rule Laboratory. *Proceedings CADE-8*, Lecture Notes in Computer Science. Springer Verlag, **230**, pp. 695–696 (1986).
19. A. Martelli and U. Montanari, *An Efficient Unification Algorithm*, ACM TOPLAS **4** (2), pp. 258–282 (1982).

20. J. Mzali, Matching with Distributivity. *Proceedings CADE-8*, Lecture Notes in Computer Science. Springer Verlag, **230**, pp. 496–505 (1986).
21. T. Nipkow, *Equational Reasoning in Isabelle*, Science of Computer Programming. 12 (1989), pp. 123–149.
22. T. Nipkow, *Term Rewriting and Beyond – Theorem Proving in Isabelle*. Formal Aspects of Computing 1 (1989), pp. 320–338.
23. P. Padawitz, *Computing in Horn Clause Theories*, ETACS Monographs in Theoretical Computer Science **16**, Springer (1988).
24. L. C. Paulson, A Higher-order implementation of Rewriting, *Science of Computer Programming* **3**, 119–149 (1983).
25. L. C. Paulson, Natural deduction as higher-order resolution. *Journal of Logic Programming* **3**, pp. 237–258 (1986).
26. L. C. Paulson, *Logic and Computation*. Cambridge University Press (1987).
27. L. C. Paulson, *The Foundation of a Generic Theorem Prover. J. of Automated Reasoning* 5 (1989), pp. 363–397.
28. L. C. Paulson, *A Preliminary User's Manual for Isabelle*. Report 133, Computer Laboratory, University of Cambridge (1988).

Book Review

DAVID S. MIALL (ed.)
Humanities and the Computer: New Directions
Clarendon Press, Oxford, 1990. £25.00
0-19-824244-1

This book consists of seventeen papers arising out of the CATH (Computers and Teaching in the Humanities) conference held at the University of Southampton in December 1988. The plethora of computer-related conferences and the accompanying rash of conference proceedings has led to a devaluation of such publications in the academic market place. The editor is therefore at pains to point out that all the chapters in this book were specially commissioned, subjected to a review process by editorial committee and, in most cases, partly rewritten as a result.

The chapters are organised so that the reader moves gradually from considerations of the broader theoretical issues to specific projects or pilot schemes. However, in spite of the obvious effort to weld a heterogeneous set of papers into a coherent whole, it is perhaps inevitable that the reader emerges with the impression that the domain is as yet somewhat unfocused. As the editor points out in his introduction, the question of whether something called 'Humanities Computing' exists as a mature discipline is an open one.

Nevertheless this is an interesting and rewarding book. The overall impression is that the contributors have approached the business of using computers in the humanities with enormous enthusiasm and imagination. The result is a fascinating and versatile repertoire of applications spread across the whole range of humanities disciplines.

Happily the enthusiasm is tempered by realism. None of the contributors is seduced by the illusion that the application of information technology is a cure-all for what has become an increasingly embattled area in higher education. Arthur Stutt in particular proposes two governing criteria for design of systems for the humanities, namely:

(i) Any system which is to be used in the humanities must take account of the nature of the humanities.

(ii) Any system which is to be used in the humanities must provide something by computational means which could not easily be provided in any other way.

In addition, other contributors point out a number of important practical difficulties. For example, the development of good-quality educational software is enormously time-consuming and is arguably at least as demanding as the writing of a large academic work. How therefore is the enterprise to be resourced? As David A. Bantz observes: 'These...examples represent quite substantial commitments of resources, most especially the time of faculty authors. Most faculties continue to believe, however, that such activity is inadequately supported or rewarded by their institutions; so long as this perception continues, the number of elaborate innovative software packages will remain small.'

In addition, crucial pedagogical and design criteria must be investigated and determined – a not inconsiderable task given the lack of concensus in the field. For example, considerable attention is given in the book to the empowering of students, which is made possible by the employment of non-directive techniques, especially through the use of hypertext as a delivery system. However, although the effect may be exhilarating, there is a real danger of student disorientation in some areas where the abandonment of a linear format can lead to navigational problems.

The occasional stylistic lapse ('new innovations' on page 3) and typographical error ('Gendel' for 'Grendel' on page 98) do not detract from the real merits of this book. It is a shame however that the bibliography is amalgamated at the end of the volume. It would have been much more reader-friendly if specific bibliographical references had been given at the end of the chapters to which they related. None the less, this work will be required reading for all those who believe that the computer has a significant rôle to play in the enhancement of humanities teaching.

TONY DRAPKIN