Formal Aspects of Computing (1989) 1: 320-338
© 1989 BCS

Formal Aspects
of Computing

Term Rewriting and Beyond – Theorem Proving in Isabelle

Tobias Nipkow
University of Cambridge, Computer Laboratory, Pembroke Street, Cambridge CB2 3QG, UK

Keywords: Theorem proving; Term rewriting; Isabelle; Quicksort

Abstract. The subject of this paper is theorem proving based on rewriting and induction. Both principles are implemented as tactics within the generic theorem prover Isabelle. Isabelle's higher-order features enable us to go beyond first-order rewriting and express rewriting with conditionals, induction schemata, higher-order functions and program transformers. Applications include the verification and transformation of functional versions of insertion sort and quicksort.

1. Introduction

This research is concerned with two aspects of automated theorem proving: the particular one of term rewriting based approaches, and the more general question of the design and use of generic theorem proving systems. By "generic" I mean a system that can be parametrised by a user supplied description of the object logic. Although the research has been carried out using a particular such system, Isabelle, many of the points I will discuss apply equally well to other, less generic systems.

The two main claims in this study are:

Term rewriting is an important and effective technique in interactive and automated theorem proving. But...
Theorem proving from first principles is desirable and possible. But...

The fact that term rewriting is an extremely important concept in theorem proving is widely accepted and has been demonstrated successfully in systems like

Correspondence and offprint requests to: T. Nipkow, University of Cambridge, Computer Laboratory, Pembroke Street, Cambridge CB2 3QG, UK.

Boyer-Moore [BoM79, BoM88], LP [GaG89], and REVE [Les86]. However, term rewriting, i.e. equational logic, on its own is a very weak logical system. In order to be useful for theorem proving it needs to be embedded in richer systems, e.g. full first-order logic, and complemented by inference rules like induction. Both Boyer-Moore and LP provide such extensions.

The term "theorem proving from first principles" describes the idea of justifying every inference by some composition of the basic rules of the object logic. Thus soundness is axiomatic, provided that the basic rules and the composition mechanism have been implemented correctly. However, in order to be able to prove theorems that require thousands of inferences, it must be possible to automate the process of searching for proofs and carrying them out. This ideal was first realised in LCF [GMW79, Pau87]: `thm` is an abstract data type of theorems whose instances can only be created by combining the basic rules of the logic; theorem proving "tactics" can be programmed in the functional language ML. Thus the system is sound but easily extensible. The importance of both properties hardly needs emphasising. Isabelle goes one step further than LCF in that it also abstracts from the particular object logic used. However, there is a certain performance penalty one has to pay.

In order to make both claims meet, we present the design of a collection of rewriting tactics with the same functionality as many of the commands in systems like LP, REVE, or Boyer-Moore. In contrast to the latter systems, Isabelle's architecture permits rewriting to be derived from first-order logic while retaining extensibility both with respect to the logic (e.g. adding induction schemata) and the tactics (e.g. adding new unification algorithms). Any user can interactively define new tactics, and, as in LCF, the type system of Isabelle's implementation language ML guarantees soundness.

The structure of the paper reflects the concern with the following three aspects:

Logic. How to derive term rewriting tactics from the rules of equational logic; how to integrate these tactics with richer logical systems. This is the subject of Sections 3 to 5.

Architecture. What capabilities are required to support theorem proving from first principles; which features are particularly important; what are the limitations, especially in performance terms. Isabelle's architecture is presented in Section 2, an evaluation of its features and limitations is the subject of Section 10.

Method. Is rewriting a viable approach to theorem proving? What other principles are useful? Is there some small set of powerful tactics that can deal with many problems? Of course these questions can only be answered with reference to a fixed application domain. In this paper we have concentrated on the area of functional programming, since it is most closely related to rewriting. Section 5 presents a small number of tactics embodying frequently used principles like simplification or induction. In Sections 6 to 8 functional versions of insertion sort and quicksort are proved correct using those tactics.

2. Isabelle

Isabelle is an interactive theorem prover developed and implemented in Standard ML [HMM86] 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

[Gro87, Nip89a]. A first explanation of the principles underlying Isabelle is contained in [Pau86], a formalisation of Isabelle's meta-logic using higher-order logic is given in [Pau89]. The version of Isabelle described in this paper is Isabelle-86. Meanwhile, Isabelle has been extended significantly and [Pau89] pertains to the latest version.

2.1. Representing Logics

What distinguishes Isabelle from most other theorem provers is the fact that it can be parametrised 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 types and constants define the syntax, the inference rules the "semantics" of a logic. The central notion in Isabelle is that of a *rule*, written as

$$\frac{P_1 \quad \ldots \quad P_m}{P}$$

where the P_i and P are simply-typed λ-calculus terms over the logical constants and variables. The Isabelle syntax for $\lambda x_1, \ldots, x_n.E$ is `%(x1, . . . , xn)E`. 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 [FGJ85] to present logics. A simple example is:

```
EQLog = SORTS term, form
        OPS _=_: term*term -> form
        RULES
```

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

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. Initially Isabelle is empty. New logics can be defined either from scratch, as EQLog above, or as enrichments of already existing systems. The latter 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

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

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

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

To support resolution, Isabelle is based on unification rather than just matching (as for example LCF). Since Isabelle formulae are λ-expressions, Isabelle contains an implementation of higher-order unification which is described in [Pau86]. 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 "?". Propositions are usually stated and proved with ordinary variables, for example $P \Rightarrow P$. Otherwise one might easily instantiate the logical variables during the proof, thus specialising the proposition. Once the proposition has been proved, Isabelle provides a function for generalising theorems which turns ordinary into logical variables. Thus one can turn $P \Rightarrow P$ into $?P \Rightarrow ?P$. The latter theorem can be used in further proofs because its logical variable $?P$ can be instantiated during resolution, in contrast to P in $P \Rightarrow P$. For readability reasons we have omitted most of the ?'s. Because variables have to be renamed during resolution, numerical suffixes are automatically appended to variables, e.g. x may become x3.

All this sounds very much like logic programming, and in fact Isabelle can be seen as an implementation of typed higher-order logic programming (see also [Pau86]).

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. A proof of some formula P starts with the trivially correct rule $\frac{\mathrm{P}}{\mathrm{P}}$ and seeks to transform it into P by successive resolution with other rules. In order to automate this tedious process, algorithmic sequences of rule applications, i.e. resolution steps, can be coded as ML functions which are known as *tactics*. Tactics are a concept originating with LCF [GMW79, Pau87]. They are the functional programmer's answer to the challenge posed by the length and repetitiveness of proofs from first principles. Just as pure Isabelle does not contain any fixed object logics, there are no predefined tactics either. Both are defined by the user in the process of creating a proof environment for a particular logic.

Interactive proofs are supported by a simple "subgoal package" which maintains a proof state, one of the few non-functional components of Isabelle. The function `goal` takes a string representing a formula P and makes $\frac{\mathrm{P}}{\mathrm{P}}$ the current proof state. The function `by` takes a tactic and applies it to the current proof state, replacing it with the first rule returned by the tactic.

Much of the system output is omitted in the examples we present. Suffice it to say that after each interaction the current subgoals are displayed one by one, numbered from `1` to `n`. Lines starting with `>` are system output.

3. Rewriting

Rewriting is central to the proof style this paper advocates. A detailed description of the implementation of rewriting in Isabelle is the subject of [Nip89a], where the realisation of various rewriting techniques as tactics is examined. This shows how rewriting is reduced to (justified by) equational logic. In fact, we use full first-order logic with equality to implement rewriting. The reason is that the tactics are intended to simplify not just terms (for example $x+0$ to x) but arbitrary logical formulae (for example $P \wedge P$ to P). Sections 3 to 5 give a top-level view of the functionality of the tactics, ignoring most of the implementation detail.

Our basic logic is a classical first-order logic with equality, in the sequel referred to by `FOLE`. It is axiomatised as a single conclusion sequent calculus. The definition of `FOLE` in Isabelle is one of the examples that Larry Paulson has provided. Paulson [Pau88] describes the actual Isabelle input; the underlying logical system is described for example in [Pau87]. Paulson's definition of `FOLE` was complemented by me with a selection of rewriting tactics which are described in detail in [Nip89a] and summarised in Section 5.

It should be noted that `FOLE` is a logic for *total* functions. Any extension of `FOLE` by definitions of functions which are necessarily partial leads to inconsistencies. This point is taken up again in Section 9.

`FOLE` introduces three syntactic categories (types): terms (`term`) and formulae (`form`) as in `EQLog` in Section 2.1, and *sequents.* The constant formulae T and F denote truth and falsity. The syntax for formulae uses `~`, `&`, `|`, `==>`, `<=>`, `ALL` and `|-` to denote $\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$, $\forall$ and $\vdash$. Sequents are objects of the `form`

$$A_1, \ldots, A_n \vdash A$$

where the formulae A_i are the *assumptions* and the formula A is the *conclusion.* The premises and the conclusion of `FOLE` rules are sequents. The sequent $\vdash A$ is mostly written A, blurring the distinction between formulae and sequents. The individual rules axiomatising `FOLE` are not listed because their precise formulation is irrelevant for our purposes. Some familiarity with first order logic is sufficient to follow the paper.

In `FOLE`, theorem proving by rewriting means the simplification of the conclusion of a sequent with the help of its assumptions and user supplied theorems. The tactics described in this paper allow rewriting of both terms and formulae, so we write *expression* when no distinction is to be made.

Rewriting proceeds via *equalities* between terms ($s = t$) or formulae ($P \Leftrightarrow Q$). A *conditional rewrite rule* is an implication $P \Rightarrow Q$ where Q is an equality. Arbitrary formulae are converted into conditional rewrite rules in two stages. Outer conjunctions and universal quantifiers are stripped off first:

$P \wedge Q$ yields the union of the formulae extracted from P and Q.
$\forall x.P$ yields the formulae extracted from $P[?v/x]$.
$P \Rightarrow \forall x.Q$ yields the formulae extracted from $P \Rightarrow Q[?v/x]$.
All other formulae are returned unchanged.

$?v$ is a new variable which can be instantiated by unification with the expression to be rewritten. For example the formula $(\forall x. x < x+1) \wedge (x+0=x)$ yields the set $\{?x < ?x+1,\ ?y+0=?y\}$.

The resulting set of formulae is translated into conditional rewrite rules. $P \Rightarrow Q$ (for non-implicational formulae assume P is T) yields

$P \Rightarrow Q$ if Q is an equality.

No rewrite rule if Q is T or F.

$P \Rightarrow (R \Leftrightarrow F)$ if Q is $\neg R$.

$P \Rightarrow (Q \Leftrightarrow T)$ otherwise.

Our example translates to $\{T \Rightarrow (?x < ?x+1 \Leftrightarrow T),\ T \Rightarrow ?y+0=?y\}$.

This extraction of conditional rewrite rules is applied automatically to each assumption formula in a sequent during a rewrite proof and can be invoked explicitly for all newly proved theorems.

The actual rewriting tactics consist of two components:

1. A tree traversal scheme for locating a "redex" (reducible subexpression). The standard strategy is bottom-up. Subexpressions are accessed via congruence rules like

$$\frac{\Gamma \vdash P \Leftrightarrow P' \quad \Gamma \vdash Q \Leftrightarrow Q'}{\Gamma \vdash (P \Rightarrow Q) \Leftrightarrow (P' \Rightarrow Q')} \tag{2}$$

2. Rule application itself - in our case conditional rewriting. Given an expression S and a conditional rewrite rule $P \Rightarrow Q$ such that S is an instance of the left-hand side of Q and the corresponding instance of P can be rewritten to T by a recursive invocation of rewriting, then S is replaced by the correspondingly instantiated right-hand side of Q.

Notice that all rewrite rules are implicitly oriented from left to right.

Exactly how this informal explanation of rewriting is realised by a sequence of resolution steps with the basic inference rules in FOLE is described in [Nip89a].

An important device in formula rewriting is the accumulation of *local assumptions*: in rewriting an implication $P \Rightarrow Q$, we can assume P while rewriting Q:

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

This derived rule replaces the almost identical congruence rule (2). The same rule holds with $\wedge$ instead of $\Rightarrow$. In rewriting a disjunction $P \vee Q$ we can assume $\neg P'$ while rewriting Q. Note that this last rule concerning $\vee$ entails the law of the excluded middle and is therefore only valid in classical logic.

Term rewriting is traditionally very much concerned with the notions of *termination* and *confluence* [HuO82]. In the presence of these two properties, equality becomes decidable. In their absence, proofs by rewriting may fail to reduce two equal expressions to the same normal form, or the rewriting process may not terminate. In contrast to systems like REVE [Les86], which support the generation of confluent and terminating sets of rewrite rules, our Isabelle instantiation does not offer any such help. The burden of checking for termination and confluence lies with the user. However, it turns out that in practice nontermination does not occur very often (we had no such problem in the proofs described in this paper) and lack of confluence rarely matters. In only one case (see Section 8.1) did we need an additional lemma to compensate for the absence of confluence.

Similar case studies in the verification of functional [MaN89] and distributed [Nip89b] programs confirm our conviction that rewriting is a highly effective theorem proving method even without automatic termination and confluence checks.

3.1. Unification and Matching

So far I have not specified how the left-hand side of a rewrite rule is matched agains the expression to be rewritten. In many cases it is sufficient to rely on Isabelle's unification algorithm to perform the matching. However, this means that laws like commutativity, which yield nonterminating rewrite rules, cannot be dealt with. For certain a common properties one can get around this problem by incorporating them into the unification or matching algorithm. Although Isabelle's unification algorithm is fixed, we provide a tactic to perform unification in special equational theories. This tactic is driven by a set of rules which depend on the particular equational theory.

The basic idea is due to Martelli and Montanari [MaM82]: systems of equations are solved by successive transformation, where each transformation simplifies the equations in some sense. This idea has been elevated to a principle in Claude Kirchner's work on unification (e.g. [Kir86]). In Isabelle the transformations are realised as rules of the form

$$\frac{u_1 = w_1 \quad \ldots \quad u_n = w_n}{u = w} \tag{3}$$

where the hypotheses should be simpler than the conclusion. The process terminates when all equations have disappeared, i.e. have been solved. The variable bindings created on the way constitute a solution.

The simplest nontrivial case is commutativity. If $+$ is commutative, the two rules

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

together with reflexivity, provide a complete set of decomposition rules for commutative unification. Currently the system provides decomposition rules for commutativity, associativity, associativity + commutativity (AC), and left and right commutativity. However, termination of unification modulo those rules is only guaranteed for commutativity. The other rules merely yield matching algorithms. Fortunately, that is all we need for rewriting. A thorough theoretical investigation of unification and matching by inference rules is contained in [Nip89c].

It should be noted that commutative, associative, and AC matching is NP complete [BKN87]. In addition the proof rule approach to matching yields something of a blind search procedure. Consequently the performance of these matching tactics is far from satisfactory.

4. Beyond

The theme of this section is the exploitation of Isabelle's higher-order features in various ways.

The most important application concerns induction. Equational reasoning alone is not very expressive, even if extended to arbitrary first-order formulae. Most theorem provers rely on some kind of induction: LP uses structural, Boyer-Moore well-founded, and LCF fixpoint induction. In Isabelle induction schemata can be expressed as inference rules:

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

The important point here is that P is a function variable of type *term → form* and that resolution of this rule with some goal requires higher-order unification.

A second application of higher-order functions allows a smooth integration of conditionals with rewriting. As part of the definition of FOLE we have declared a constant *if*: *form* * *term* * *term* → *term* and added the rule

$$P(if(Q,x,y)) \Leftrightarrow (Q \Rightarrow P(x)) \wedge (\neg Q \Rightarrow P(y)) \tag{4}$$

Again P is of type *term → form*. Defining functions via *if* and using (4) as a rewrite rule provides a simple way of automating case distinctions.

A final application is the definition of data types with higher-order functions. This will turn out to be crucial for an elegant treatment of specification and verification in the realm of functional programming.

5. Tactics

Before embarking on any real proof the actual ML tactics are introduced. They are not part of Isabelle or the definition of FOLE but have been developed on top of both by the author.

The proofs in this paper are strictly by induction and rewriting, with the latter performing virtually all inferences. The standard rewriting strategy is the one described above: bottom-up conditional rewriting with user supplied rules, assumptions, local assumptions, and conditionals (see Section 4). In addition a set of rules for formula simplification is built in. This set contains simple rules like $P \wedge T \Leftrightarrow P$ and $(\forall x.T) \Leftrightarrow T$, but is by no means complete.

Although there exists a complete rewrite system for propositional logic [Zhe27, HsD83], its use in interactive theorem proving is problematic. The reasons are its unavoidably high complexity and the fact that it is phrased in terms of conjunction and exclusive or. The procedure works well if the given proposition is indeed a tautology. But in most cases the proposition is either not true or needs some more massaging before it becomes a propositional tautology. In those cases it is not very helpful if the "simplification procedure" goes away for a long time only to come back with a formula that has become unintelligible due to the presence of exclusive or. It seems that the utility of a rewrite system for propositions requires a fine balance between incompleteness and incomprehensibility.

There are two top-level rewriting tactics:

```
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.

Rather than supplying the rewrite tactics with exactly the rules required to solve a particular goal, we have a top-level ML variable rwrls which contains the set of all currently available rewrite rules. Every time a new lemma is proved which can be used as a rewrite rule, it is preprocessed according to the scheme outlined in Section 3 and added to rwrls. Although this slows down rewriting it ensures that no rule is accidentally ignored. This is exactly the same principle as followed in LP or REVE. In contrast to the latter systems, the classification of lemmas into those that can safely be used as rewrite rules and those that may lead to divergence is currently not supported. For applications from the area of functional programming this has turned out not to be a problem. In fact, all theorems proved in the application sections on sorting are terminating rewrite rules. Therefore the reader may assume that they are all accumulated in rwrls.

Induction is performed by

```
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. This tactic is in fact more widely applicable than just to induction. In particular case distinction instead of induction schemas can be supplied.

6. Sorting

To determine the power and the limitations of our set of tactics, we turn to the correctness proof of two sorting algorithms, insertion sort and quicksort. In order to minimise the amount of additional logical machinery, and to get the maximum benefit out of rewriting, all algorithms are expressed as functional programs. Rather than using a logic for domain theory like LCF does, we stick to a first-order system with equality. The limitations of this approach will become apparent in Section 8.2.

Imperative programmers sort arrays, functional programmers sort lists:

```
List = FOLE +
       SORTS elem, list
       OPS nil: list
           _:_: elem*list -> list
           _@_: list*list -> list
       RULES
           nil @ z=z
           (x:z)@z1=x: (z@z1)
           P(nil) & (ALL z. ALL x. P(z)==> P(x:z))
                                        ==> ALL z. P(z)
```

The last rule is the structural induction principle for lists. For future reference it is bound to the ML identifier LIST_IND1. The derived rule

```
P(nil) & (ALL z. ALL x. P(x:z)) ==> ALL z. P(z)
```

is used to reason by exhaustion and stored in the ML variable LIST_EXH. Although induction subsumes exhaustion, it is interesting to see which theorems do require induction and which can be proved by exhaustion.

In order to sort lists, the elements must be ordered according to some total relation:

```
TRel = FOLE +
       SORTS elem
       OPS _=<_: elem*elem -> form
       RULES x =< y | y =< x
```

It turns out that we do not need to require transitivity provided we adopt a correspondingly weaker definition of when a list is sorted.

An important rule derived from totality of =< is the implication ~ x =< y ==> y =< x, which, unfortunately, does not terminate if used as a conditional rewrite rule: modulo variable renaming the conclusion is a subterm of the hypothesis. There is a simple trick to get around this problem. We introduce the predicate >, define ~ x =< y <=> y > x, and derive x > y ==> y =< x. The last two rules terminate and achieve the same effect as the nonterminating implication.

The correctness of sorting can be specified in many different ways. The formulation we chose comes in two parts. The first requirement is that the result of sorting a list should be sorted:

```
SList = TRel + List +
        OPS sorted: list -> form
        RULES sorted(nil)
              sorted(x:nil)
              sorted(x: (y:z)) <=> x =< y & sorted(y:z)
```

Notice that the ordering has been made a globally available predicate rather than a function that is explicitly passed to sorted. This simplifies the statement of many theorems because the properties of =< are given by the context and do not clutter up the statement itself.

A stronger definition of *sorted* might insist that every element in the list is =< any element to the right of it, not just its immediate right neighbour. The advantage of the weaker version is that the sorting algorithms can be shown to conform to it without having to assume transitivity of =<. If =< is transitive, the stronger statement is implied by the weaker one.

The second requirement has to capture that the output list contains the same elements as the input. This prevents trivial implementations like always returning nil (which is sorted). This can be expressed with the help of the data type of *bags* or *multi-sets*.

```
Bag = SORTS elem, bag
      OPS mt: bag
          inj: elem*bag -> bag
  RULES inj(x, inj(y, z)) = inj(y, inj(x, z))
     P(mt) & (ALL z. P(z) ==> P(inj(x, z))) ==> ALL z. P(z)
```

The first rule states left-commutativity of injection, the second rule is the structural induction principle. Left-commutativity cannot be used as a rewrite rule. Fortunately there is a predefined matching tactic (see Section 3.1) for left-commutativity. In the sequel we assume that the instantiation of this tactic with inj is bound to the ML identifier bag_match_tac.

Lists can be turned into bags using

```
LBag = List + Bag +
       OPS bag: list -> bag
       RULES bag(nil) = mt
             bag(x:z) = inj(x, bag(z))
```

The correctness of a sorting algorithm sort can now be established by proving

```
sorted(sort(z)) and bag(sort(z)) = bag(z)                    (5)
```

All proofs are presented in a bottom-up manner, i.e. lemmas before theorems. This is in keeping with mathematical tradition, but has the disadvantage that even if the reader is familiar with the statement of the main theorem, it is not always obvious why a particular lemma is necessary for its proof. In fact, the lemma may not be required at all if one follows a different proof. Of course the discovery of a proof usually proceeds much more in a top-down way: in trying to prove the main goal one gets stuck, analyses the problem, concludes that a certain lemma needs to be proved first, and proceeds to prove it.

7. Insertion Sort

Insertion sort has a very simple definition by primitive recursion on lists:

```
Isort = TRel + List +
        OPS isort: list -> list
            ins: elem * list -> list
        RULES isort(nil) = nil
            isort(x:z) = ins(x,isort(z))
            ins(x,nil) = x:nil
            ins(x,y:z) = if(x =< y, x:(y:z), y:ins(x,z))
```

The function ins inserts an element x to the left of the first element y in a list such that x =< y. As in the definition of sorted, the ordering is global. Thus the correctness requirements can be stated exactly as in (5) above, with isort for sort. In order to formulate and prove correctness as in (5), all proofs below are conducted in the theory Isort + TRel + SList + LBag.

As we shall see shortly, insertion sort presents a particularly simple verification problem because both isort and ins are defined by primitive recursion over lists.

Let us first prove that isort returns a sorted list. Looking at the recursion equation for isort, it is not difficult to see that one of the lemmas needed for this proof is that ins preserves sortedness. A first attempt at a proof proceeds by exhaustion:

```
goal "sorted(z) ==> sorted(ins(x,z))";
by(IND_TAC (SIMP_TAC rwrls) LIST_EXH "z");
> 1. ALL u.sorted(x4 : z4) ==> ~ u =< x4 ==>
             sorted(x4 : ins(u,z4))
```

Unfortunately, one subgoal has remained. It might be tempting to try an induction on z4, but it is in general preferable to prove a separate lemma. This lemma may come in handy later on or provide some additional insight. The particular lemma that seems to be necessary in our case is the following one:

```
goal "x =< y & sorted(x:z) ==> sorted(x:ins(y,z))";
by(IND_TAC (SIMP_TAC rwrls) LIST_IND1 "z");
```

A single structural induction suffices and the resulting lemma can be added to the global list of rewrite rules. Equipped with this new lemma the above attempt to prove `sorted(z) ==> sorted(ins(x,z))` succeeds now.

As for the main theorem, it does now follow directly by structural induction:

```
goal "sorted(isort(z))";
by(IND_TAC (SIMP_TAC rwrls) LIST_IND1 "z");
```

The proof of `bag(isort(x))=bag(z)` is even simpler. The only lemma we need is

```
goal "bag(ins(x,z)) = inj(x,bag(z))";
by(IND_TAC (EQ_SIMP_TAC bag_match_tac rwrls)LIST_IND1 "z");
```

The theorem itself follows in exactly the same way:

```
goal  "bag(isort(z)) = bag(z)";
by(IND_TAC (EQ_SIMP_TAC bag_match_tac rwrls)LIST_IND1 "z");
```

This concludes the correctness proof of `isort`.

8. Quicksort

The following definition of quicksort and its correctness proof rely on two higher-order functions on lists:

```
HOList = List +
          OPS filter: list * (elem -> form) -> list
              forall: list * (elem -> form) -> form
         RULES filter(nil,P) = nil
              filter(x:z,P) = if(P(x), x:filter(z,P),
                                       filter(z,P))
              forall(nil,P)
              forall(x:z,P) <=> P(x) & forall(z,P)
```

Both `filter` and `forall` are basic building blocks in any functional programmer's library. Given a list `l` and a predicate P, `forall` tests whether all elements in `l` satisfy P, and `filter` returns the list of those that do. These two functions enable us to state a number of general lemmas which should be useful beyond the correctness proof of quicksort.

Although the above definition of `filter` is quite elegant, it is not really a functional program. The second argument should be of type `elem -> bool`, where `bool` is the type of computational booleans. Identifying the latter with the type of formulae is slightly cavalier but admissible, since we reason in a logic of total functions.

Quicksort can now be given a very simple definition:

```
Qsort = TRel + HOList +
        OPS qsort: list -> list
        RULES qsort(nil) = nil
              qsort(x:z) = qsort(filter(z,%(y)~ x =< y))
                           @ (x:nil)
                           @ qsort(filter(z,%(y)x=< y))
```

In contrast to all functions encountered so far, qsort is not defined by primitive recursion. Nevertheless it is total because the length of the argument list decreases with each recursive call. The corresponding induction principle is induction over the length of a list. In order to formulate this rule we need to introduce natural numbers:

```
Nat = FOLE +
      SORTS nat
      OPS 0: nat
          s: nat -> nat
        _<_: nat *  nat -> form
      RULES 0 < s(x)
            s(x) < s(y) <=> x < y
            ~ x < 0
```

The definition of the length function len and of the induction principle is straightforward:

```
List2 = List + Nat +
        OPS len: list -> nat
        RULES len(nil) = 0
              len(x:z) = s(len(z))
              (ALL z. (ALL z1. len(z1) < len(z) ==> P(z))
              ==> ALL z. P(z)
```

Note that the above induction principle on its own is of no use at all since it does not capture the fact that lists are generated by nil and :. The latter can be incorporated by resolution with LIST_EXH which, after simplification, yields

```
P(nil) & (ALL z.ALL x.(ALL z1. len(z1) < s(len(z))
   ==> P(z1)) ==> P(x:z))
   ==> ALL z. P(z)
```

This induction rule is bound to the ML identifier LIST_IND2.

For the same reason that structural induction is insufficient for qsort, a richer theory of bags is required:

```
Bag2 = Bag +
       OPS _+_: bag *  bag -> bag
           forallB: bag *  (elem -> form) -> form
       RULES mt + z = z
           inj(x,z) + z1 = inj(x,z + z1)
           forallB(mt,P)
           forallB(inj(x,z),P) <=> P(x) & forallB(z,P)
```

The operation + is the union on bags and forallB tests whether a predicate holds for all members of a bag.

By structural induction it follows that + is AC. This gives rise to a new bag_match_tac which covers both left-commutativity of inj and associativity-commutativity of +. This tactic is used in the proof of all theorems involving bags.

8.1 The Correctness Proof

Before the main theorems (5) are tackled, three sets of lemmas are proved. Of course the need for these lemmas was only discovered when trying to do the main proof.

The first group deals with simple properties of forall and filter:

```
forall(z@z1,P) <=> forall(z,P) & forall(z1,P)
(ALL x.P(x) ==> Q(x)) ==> forall(filter(z,P),Q)
```

Both lemmas are proved by a single application of structural induction on z.

The second group of lemmas shows that filter cannot increase the length of a list. This fact will enable us to use induction on the length of a list in proofs about qsort. The only lemma we need is one about natural numbers: x < y ==> x < s(y). The latter is proved by structural induction on x followed by exhaustion. The main lemma

```
len(filter(z,P)) < s(len(z))                          (6)
```

follows immediately by structural induction on z.

The third group of lemmas concerns the interaction of bags and lists:

```
bag(z@z1) = bag(z) + bag(z1)
bag(filter(z,P)) + bag(filter(z,%(x)~P(x))) = bag(z)      (7)
forallB(bag(z),P) <=> forall(z,P)                          (8)
```

All three of them follow immediately by structural induction on z. Note that we cannot write ~P instead of %(x)~P(x) as in (7) because P is of type term -> form but negation can only be applied to formulae.

In contrast to insertion sort, where the two halves of the correctness statement (5) are proved independently, for quicksort it is convenient to use one in the proof of the other. The proof of the second theorem is immediate:

```
goal "bag(qsort(z)) = bag(z)";
by(IND_TAC (EQ_SIMP_TAC bag_match_tac rwrls)
   LIST_IND2 "z");
```

The proof of the first half of (5) requires some more lemmas, one of which is

```
forall(qsort(z),P) <=> forall(z,P)
```

Term rewriting specialists will immediately realise that this equation is simply the critical pair between rule (8) and bag(qsort(z)) = bag(z). Although there is no tactic for generating critical pairs per se, the simplification tactics can be used for this derivation. The proof is short but a bit obscure and is not presented here.

The next two lemmas help to "unfold" the sorted predicate. The first one,

```
sorted(z) & forall(z,%(y)x=<y) ==> sorted(x:z)
```

follows immediately by structural induction on z. It would be a proper unfolding rule if the reverse implication held as well. That, however, holds only if =< is transitive. The same is true for the second lemma, which we attack by induction on z1:

```
goal "sorted(z1) & forall(z1,%(y)y =<x) &
      sorted(z2) & forall(z2,%(y)x=<y)
     ==> sorted(z1@(x:z2))";
by(IND_TAC (SIMP_TAC rwrls) LIST_IND1 "z1");
> 1. (ALL u.ALL v.sorted(z6) & forall(z6,%(y)y =< v) &
   sorted(u) & forall(u,%(y)v =< y)
        ==> sorted(z6 @ : u)) ==>
     ALL u.ALL v. sorted(x6 : z6) & x6 =< v &
              forall(z6,%(y)y =< v) &
                  sorted(u) & forall(u,%(y)v =< y)
                     ==>
                  sorted(x6 : (z6 @ v : u))
```

The remaining subgoal is the induction step. No simplification could take place because the definition of sorted requires z6 to be instantiated further. This time a simple case distinction

```
by(IND_TAC (SIMP_TAC rwrls) LIST_EXH "z6");
```

suffices to solve the subgoal.

Finally, a single induction establishes that qsort actually sorts:

```
goal "sorted(qsort(z))";
by(IND_TAC (SIMP_TAC rwrls) LIST_IND2 "z");
```

8.2. Program Equivalence and Transformation

The version of quicksort presented above has two major drawbacks: it is not in the least tail-recursive and requires explicit concatenation of the two sorted sublists. Both sources of inefficiency can be attacked by introducing a second parameter which accumulates the result:

```
qsort'(nil,z1) = z1
qsort'(x:z,z1) = qsort'(filter(z,%(y)~ x =< y),
                        x:qsort'(filter(z,%(y)x =< y),z1))
```

This definition eliminates @ and moves one of the two calls to qsort' into a tail-recursive position. The claim is that `qsort'(z,nil)` returns the sorted version of z. Proving `qsort'` correct from scratch does not look like a good idea. Instead one should try to relate it back to `qsort`. A first attempt might be to establish that `qsort'(z,nil) = qsort(z)`. It turns out that this needs to be generalised to

```
qsort'(z,z1) = qsort(z) @ z1                                    (9)
```

But this is not the only possible generalisation. The relationship between the two versions of quicksort is an instance of the following general schema. Given functions `f`, `f'`, `g` and `h` such that

```
f(nil) = nil
f(x:z) = f(g(x,z)) @ (x:f(h(x,z)))
f'(nil,z1) = z1
f'(x:z,z1) = f'(g(x,z), x:f'(h(x,z),z1))
```

then

```
f'(z,z1) = f(z) @ z1                                            (10)
```

holds for all g and h.

It turns out that the above implication is not valid, let alone provable, in first-order logic with equality. The reason is divergence. If `g(x,z)` was for example `x:z`, `f` would not terminate on any non-nil argument. But neither would `f'`, and hence (10) still holds. The problem is that there is no notion of nontermination in our logic. It is obvious that the situation calls for a logic designed to deal with partial functions, for example LCF.

This highlights a problem that is likely to come up in any framework for reasoning about total functions only. In this case there is a simple fix: if we require additionally that g and h return lists that are no longer than their second argument, totality of `f` and `f'` is guaranteed. The implication can now be proved by a single induction on the length of z in (10).

To put this implication to good use, we need to show that `qsort` and `qsort'` satisfy the requirements for `f` and `f'` respectively. At this point higher-order unification comes to the rescue: since `f`, `f'`, `g` and `h` are free, they become ?-variables, and the four equations are shown to hold simply by unification with the corresponding clauses for the two quicksorts. As a result `?f` and `?f'` become instantiated by `qsort` and `qsort'` respectively, and ?q and ?h by some functions involving `filter`. The two additional requirements on g and h are solved by unification with lemma (6). Now all assumptions have been discharged and we are left with exactly the conclusion (9).

Although this is really a program equivalence proof, it can also be used to derive the definition of `qsort'`: after unifying the two equations for `?f` and `qsort`, the remaining two premises tell us exactly what that definition of `?f'`, i.e. `qsort'` ought to be.

However, there is a more direct way to derive `qsort'` from `qsort`: Burstall and Darlington's unfold/fold technique [BuD77]. In this case we start with the two clauses for `f` and the relationship (10) – the latter is the "Heureka" step in [BuD77]. The clauses for `f'` can now be derived by simple equational reasoning. For `f'(nil,z1)=z1` this is trivial. Using (10), unfolding the definition of `f`, and associativity of @ yields:

```
    f'(x:z,z1) = f(g(x,z)) @ (x: (f(h(x,z)) @ z1))
```

Two applications of (10) in the opposite direction (the folding) yield the second clause for f'. It is interesting to see that the program transformation schema was easier to derive (pure equational reasoning) than the program equivalence (induction). The application of the transformation schema to qsort again reduces to higher-order matching.

Finally I would like to point out that the transformation has applications beyond quicksort. Instantiating h by %(x,z)nil, the behaviour of f and f' on nil implies

```
    f(x:z) = f(g(x,z)) @ (x:nil)
    f'(x:z,z1) = f'(g(x,z), x:z1))
```

This schema can be used to bring a function f into tail recursive form f', the typical example being list reversal.

The derivation of all this in Isabelle is not shown because the reader has by now seen enough applications of induction and simplification.

9. Related Research

This paper combines ideas and draws inspiration from many sources. Central to the whole endeavour is Larry Paulson's Isabelle work. Isabelle in turn is strongly influenced by the LCF-approach to theorem proving.

When I started my work on term rewriting in Isabelle [Nip89a], I did not fully appreciate [Pau83]. The implementation presented in Section 3 is now much closer to [Pau83] in functionality. In particular the use of local assumptions is borrowed from there. The extension to rewriting modulo equations as presented in Section 3.1 is closely related to Claude Kirchner's work on equational unification, e.g. [Kir86].

The underlying logic is a classical first order logic with equality as described in [Pau87]. What distinguishes it from LCF is the absence of domain theory, making it a logic for total functions. Thus its quantifier free fragment is quite close to LP or Boyer-Moore, except that the latter system enforces totality of all functions. In a sense the current state gives us the worst of both worlds: neither can we reason properly about partial functions, nor does the system check that all functions are actually total. Although reasoning in a logic of total functions is definitely simpler than for example in LCF, Section 8.2 shows the limitations when dealing with schemata rather than particular functions.

Theorem proving based on induction and rewriting was first championed in the Boyer-Moore system which still seems to be the most sophisticated theorem prover based on this style. Although LCF and LP come from rather different backgrounds, it turns out that in practice they mainly rely on induction and rewriting too. Not surprisingly, the proofs in this paper follow the same pattern.

Peter Padawitz has combined resolution, paramodulation and induction into a calculus for reasoning about what he calls "constructor based Horn clauses". Among the many examples presented in [Pad88a,b] there is also an insertion sort correctness proof which is very similar to the one in Section 7.

The idea of using higher-order matching for program transformation is due to Huet and Lang [HuL78]. More recently the idea has been picked up again in [PfE88, HaM88] because more implementations of higher-order unification are available. Isabelle seems an ideal implementation vehicle for these purposes because it offers both the capabilities for proving transformations correct and for applying them by higher-order matching. The same is true with respect to Burstall and Darlington's work on program transformation [BuD77].

10. Conclusion

We have shown that a generic system like Isabelle is a suitable implementation basis for rewriting techniques. Some nontrivial examples of program correctness proofs and transformations provide empirical evidence that the resulting tactics constitute powerful theorem proving primitives. For our purposes the single most important feature in Isabelle is higher-order unification. It enables us to go beyond first-order rewriting and express rewriting with conditionals, induction schemata, higher-order functions and program transformers. Although Isabelle is a generic and FOLE is just one sample instantiation, our rewriting tactics are FOLE-specific. Larry Paulson has recently gone one step further and developed an ML module that generates rewriting tactics from the basic equational laws in an arbitrary logic.

The biggest problem with the "generic" approach is efficiency. This has some unfortunate consequences for theorem proving. If the system is too slow, it will not be used to find proofs but merely to check them. The construction will take place outside the system rather than by experiments within it. Furthermore, users will tend to stick to faster but less powerful tactics, thus complicating many proofs. An example of this problem is the AC-rewriting tactic. Certain proofs would have simplified if rewriting of formulae took associativity-commutativity of conjunction into account. In general, however, this turns out to be too expensive, due to the tactic-level implementation of AC-matching.

Since rewriting is so common, it would be advantageous if some of its functionality was already built into Isabelle, e.g. in the form of paramodulation or special unification algorithms. The latter poses a certain theoretical problem because these new unification algorithms would have to coexist with higher-order unification.

Acknowledgments

This research was carried out at the Laboratory for Computer Science, MIT, with support from NYNEX, NSF grant CCR-8706652, and the Advanced Research Projects Agency of the DoD, monitored by the ONR under contract N00014-83-K-0125.

I am indebted to Larry Paulson for his Isabelle system and much helpful e-mail, to Dave Matthews for his Poly/ML system, to John Guttag and Stephen Garland for many discussions on theorem proving in general and term rewriting in particular, to Deepak Kapur for showing me how the quicksort proof could be simplified significantly, to Rolf Bahlke for carefully reading a draft, and to Mario Wolczko for `under.sty`.

References

[BKN87] Benanav, D., Kapur, D. and Narendran P.: Complexity of Matching Problems. *J. Symbolic Computation*, **3**, 203-216 (1987).

[BoM79] Boyer, R. S. and Moore, J. S.: *A Computational Logic*, Academic Press, 1979.

[BoM88] Boyer, R. S. and Moore, J. S.: *A Computational Logic Handbook*, Academic Press, 1988.

[BuD77] Burstall, R. M. and Darlington, J.: A Transformation System for Developing Recursive Programs. *J. ACM*, **24**, 44-67 (1977).

[Gro87] de Groote, Ph.: How I Spent my Time in Cambridge with Isabelle, Report RR 87-1, Unité d'Informatique, Université Catholique de Louvain, Belgium, 1987.

[FGJ85] Futatsugi, K., Goguen, J. A., Jouannaud, J.-P. and Meseguer, J.: Principles of OBJ2, *Proc. 12th ACM Symp. on Principles of Programming Languages*, 52-66, 1985.

[GaG89] Garland, S. J. and Guttag, J. V.: An Overview of LP, The Larch Prover, *Proc. 3rd Intl. Conf. Rewriting Techniques and Applications*, LNCS 355, 137-151, Springer-Verlag, 1989.

[GMW79] Gordon, M. J. C., Milner, R. and Wadsworth, C. P.: *Edinburgh LCF: A Mechanised Logic of Computation*, LNCS 78, Springer-Verlag, 1979.

[HaM88] Hannan, J. and Miller, D.: Uses of Higher-Order Unification for Implementing Program Transformers, *Proc. 5th Intl. Logic Programming Conf.*, 1988.

[HMM86] Harper, R., MacQueen, D. and Milner, R.: Standard ML, Report ECS-LFCS-86-2, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, 1986.

[HuL78] Huet, G. and Lang, B.: Proving and Applying Program Transformations Expressed with Second Order Patterns. *Acta Informatica*, **11**, 31-55 (1978).

[HuO82] Huet, G. and Oppen, D. C.: Equations and Rewrite Rules - A Survey. In: *Formal Languages: Perspectives and Open Problems*, R. Book (ed.), Academic Press, 1982.

[HsD83] Hsiang, J. and Dershowitz, N.: Rewrite rules for clausal and non-clausal theorem proving, *Proc. 10th Intl. Colloq. on Automata, Languages, and Programming*, LNCS 154, 431-446, Springer-Verlag, 1983.

[Kir86] Kirchner, C.: Computing Unification Algorithms, *Proc. Symp. on Logic in Computer Science*, Cambridge, MA, 206-217, 1985.

[Les86] Lescanne, P.: REVE: A Rewrite Rule Laboratory, *Proc. 8th Intl. Conf. on Automated Deduction*, LNCS 230, 695-696, Springer-Verlag, 1986.

[MaM82] Martelli, A. and Montanari, U.: An Efficient Unification Algorithm. *ACM TOPLAS*, 4(2), 258-282 (1982).

[MaN89] Martin, U. and Nipkow, T.: Automating Squiggol, Report 179, Computer Laboratory, University of Cambridge, September 1989. To appear in Proc. IFIP TC2 Working Conf. Programming Concepts and Methods, April 1990.

[Nip89a] Nipkow, T.: Equational Reasoning in Isabelle. *Science of Computer Programming*, **12**, 123-149 (1989).

[Nip89c] Nipkow, T.: Proof Transformations for Equational Theories, Report 181, Computer Laboratory, University of Cambridge, September 1989.

[Nip89b] Nipkow, T.: Formal Verification of Data Type Refinement - Theory and Practice, *Proc. REX Workshop on Refinement of Distributed Systems*, to appear in LNCS.

[Pad88a] Padawitz, P.: Inductive Proofs of Constructor-Based Horn Clauses, Report MIP-8810, Fakultät für Mathematik und Informatik, Universität Passau, 1988.

[Pad88b] Padawitz, P.: Inductive Proofs by Resolution and Paramodulation, Tech. Report, Fakultät für Mathematik and Informatik, Universität Passau, 1988.

[Pau83] Paulson, L. C.: A Higher-Order Implementation of Rewriting. *Science of Computer Programming*, **3**, 119-149 (1983).

[Pau86] Paulson, L. C.: Natural Deduction as Higher-Order Resolution. *Journal of Logic Programming*, **3**, 237-258 (1986).

[Pau87] Paulson, L. C.: *Logic and Computation*, Cambridge University Press, 1987.

[Pau88] Paulson, L. C.: A Preliminary User's Manual for Isabelle, Report 133, Computer Laboratory, University of Cambridge, May 1988.

[Pau89] Paulson, L. C.: The Foundation of a Generic Theorem Prover. *Journal of Automated Reasoning*, in press.

[PfE88] Pfenning, F. and Elliot, C.: Higher-Order Abstract Syntax, *Proc. ACM-SIGPLAN Conf. on Programming Language Design and Implementation*, 199-208, 1988.

[Zhe27] Zhegalkin, I. I. On a Technique of Evaluation of Propositions in Symbolic Logic. *Matematicheskii Sbornik*, **34**(1), 9-27 (1927).

Received July 1988
Accepted in a revised form in October 1989 by M.-C. Gaudel and C. B. Jones