Automated Deduction in Arithmetic with the Omega Rule

Siani Baker
Computer Laboratory
University of Cambridge
Cambridge, CB2 3QG, UK
E-mail: slb1004@cl.cam.ac.uk

Abstract

An important technique for investigating derivability in formal systems of arithmetic has been to embed such systems into semi-formal systems with the ω-rule. This paper exploits this notion within the domain of automated theorem-proving and discusses the implementation of such a proof environment. This involves providing an appropriate representation for infinite proofs, and a means of verifying properties of such objects.

1 Introduction

Normally, proofs considered in theorem-proving are finite; however, there is a reasonable notion of infinite proof involving the ω-rule, which infers a proposition from an infinite number of individual cases of that proposition. The ω-rule involves the use of infinite proofs, and therefore poses a problem as far as implementation is concerned.

With the goal of automatic derivation of proofs within some formalisation of arithmetic in mind, an (implementable) representation for an arithmetical system including the ω-rule is proposed. The implemented system is useful as a proof environment (and also as a guide to generalisation in the more usual formalisation of arithmetic [Baker *et al* 92]).

2 The Constructive Omega Rule

A standard form of the ω-rule is

$$\frac{A(0), A(\underline{1}) \ldots A(\underline{n}) \ldots}{\forall x A(x)}$$

where $\underline{n}$ is a formal numeral, which for natural number n consists in the n-fold iteration of the successor function applied to zero, and A is formulated within the language of arithmetic. This rule is not derivable in Peano Arithmetic (PA)[1], since for example, for the Gödel formula $G(x)$, for each natural number n, $PA \vdash G(\underline{n})$ but it is not true that $PA \vdash \forall x G(x)$. This rule together with Peano's axioms gives a complete theory – the usual incompleteness results do not apply since this is not a formal system in the usual sense.

However, this is not a good candidate for implementation since there are an infinite number of premises. It would be desirable to restrict the ω-rule so that the infinite proofs considered possess some important properties of finite proofs. One suitable option is to use a **constructive ω-rule**. The ω-rule is said to be constructive if there is a recursive function f such that for every n, $f(n)$ is a Gödel number of $P(n)$, where $P(n)$ is defined for every natural number n and is a proof of $A(\underline{n})$ [Takeuti 87]. This is equivalent to the requirement that there is a uniform, computable procedure describing $P(n)$, or alternatively that the proofs are recursive (in the sense that both the prooftree and the function describing the use of the different rules must be recursive) [Yoccoz 89]. There is a primitive recursive counterpart[2] which is also a candidate for implementation. Note that in particular these rules differ from the form of the ω-rule (involving the notion of provability) considered by Rosser [Rosser 37] and subsequently Feferman [Feferman 62].

Various theoretical results are known for these systems. Shoenfield has shown that 'PA + ω-rule' (PA_ω)[3] is equivalent to 'PA + recursively restricted ω-rule' [Shoenfield 59]. The sequent calculus enriched with the recursively restricted ω-rule in place of the rule of induction (let us call it $PA_{r\omega}$[4]) has cut elimination, and is complete [Shoenfield 59].

[1] See for example [Schwichtenberg 77] for a formalisation.

[2] In other words, such that there is a primitive recursive function f for which, for every n, $f(n)$ is a Gödel number of the proof of $A(\underline{n})$, the nth numerator of the ω-rule.

[3] See Section 3 below for description.

[4] For a more formal description see [Baker 92a].

The primitive recursive variant has also been shown to be complete by Nelson [Nelson 71]. If one has the rule of repetition $\frac{\Gamma \vdash \Delta}{\Gamma \vdash \Delta}$ in PA_ω, any recursive derivation can be "stretched out" to a primitive recursive derivation using the same rules of inference, plus this rule [López-Escobar 76, P169]. Since my implementation is developed using effective operations over representations of object-level syntax (where effectiveness is an analogous concept to primitive recursion), and PA with the unrestricted ω-rule forms a conservative extension of this system, the (classical) system PA with a primitive recursive restriction on the prooftrees was chosen as a basis for implementation.

In the context of theorem proving, the presence of cut elimination for these systems means that generalisation steps are not required. In the implementation, although I do not claim completeness, some proofs that normally require generalisation can be generated more easily in $PA_{c\omega}$ than PA.

3 $PA_{c\omega}$: Arithmetic with the Constructive Omega Rule

The system PA_ω is essentially PA enriched with the ω-rule in place of the rule of induction. The derivations are then infinite trees of formulae; a formula is demonstrated in PA_ω by "exhibiting" a prooftree labelled at the root with the given formula. Syntactical details about this system PA_ω are given in [López-Escobar 76, P162] (see [Prawitz 71, P266–267] for a natural deduction representation). PA_ω has been described by Schütte as a semi-formal system to stress the difference between this and usual formal systems which use finitary rules [Schütte 77, P174].

For implementational purposes infinite proofs must be thought of in the constructive sense of being generated, rather than absolute. It is necessary to place a restriction on the prooftrees of PA_ω such that only those which have been constructively generated are allowed, in order to capture the notion of infinite labelled trees in a finite way. The normal approach when dealing with a system with infinitary proofs such as PA_ω is to work with numeric codes for the derivations rather than using the derivations themselves. See [Schwichtenberg 77, P886] for further details, including the case of the ω-rule. By adding the provability relation and numeric encoding, a reflection system which necessarily extends the original one may be formed [Kreisel 65, P163]. However, the necessity of using this Gödel numbering approach may be avoided by following Tucker in defining primitive recursion ("effectiveness") over various data-types that are better adapted to computational purposes [Tucker *et al* 90].

If an arithmetical encoding method were to be used, the primitive recursive constraint could be attached directly to the ω-rule. However, without using such an approach the restriction must be placed on the shape of the proof tree in which the ω-rule appears: only derivations which are "effective" will be accepted[5].

Hence I define

$$\vdash_{PA_{c\omega}} \Phi \quad \textit{iff} \quad \exists f.\ f \text{ is an 'effective' prooftree of } PA_{c\omega} \text{ with } \Phi \text{ as initial sequent.}$$

This does not define $PA_{c\omega}$ as a (semi-)formal system in the sense that it does not say what the axioms and rules of inference are. This replaces the usual approach of using numeric encoding (using notation $\ulcorner\ \urcorner$) to consider some statement used to strengthen PA of the form:

$$(\exists \Pi\ (\mathsf{prooftree}(\Pi)\ \wedge\ \mathsf{conc}(\Pi) = \ulcorner \Phi \urcorner)) \rightarrow \Phi.$$

It is now necessary to provide some means for reasoning about primitive recursive infinite proof trees. The objects of interest are prooftrees (in the sense of López-Escobar [López-Escobar 76]), labelled with formulae (namely, the sequents to be proved at each point) and rules. The notion of effectiveness of a tree, which corresponds to primitive recursion, is defined in [Baker 92a]. In addition, a (proof) tree must be well-founded, in the sense that it does not have an infinitely deep branch. The rules that relate the formulae between node and subnode are the standard rules for the logical connectives, the extra ω-rule with subgoals $\Phi(\underline{0}), \Phi(\underline{1}), \ldots$, and substitution. A formula in PA is demonstrated in the extended theory by exhibiting a prooftree labelled at the root with the given formula. Properties of such primitively recursively defined trees can be proved using induction principles associated with the datatypes. These are the sorts of proofs that have been automated by [Bundy *et al* 91], and I am able to automate the simpler proofs that arise. This involves, for example, giving a proof that a given rewrite applied a given number of times to a formula schema yields a particular formula schema. Details of this, and of how the implementation relates to the effective prooftrees are considered in [Baker 92a].

4 Implementational Representation

One use of the constructive ω-rule is to enable automated proof of formulae, such as $\forall x\ (x + x) + x = x + (x + x)$, which cannot be proved in the normal axiomatisation of arithmetic without recourse to the cut rule. In these cases the correct proof could be extremely difficult to find automatically. However, it is possible to prove this equation using the ω-rule since the proofs of the instances $(0 + 0) + 0 = 0 + (0 + 0), (1 + 1) + 1 = 1 + (1 + 1), \ldots$ are easily found, and the general pattern determined by inductive inference.

[5] [Baker 92a] provides a full definition of effectiveness.

Axioms

$$0 + y = y \quad (1)$$
$$s(x) + y = s(x + y) \quad (2)$$

Proof

$$(\underline{n} + \underline{n}) + \underline{n} = \underline{n} + (\underline{n} + \underline{n})$$

$\underline{n} \equiv s^n(0)$	$(s^n(0) + s^n(0)) + s^n(0) = s^n(0) + (s^n(0) + s^n(0))$
(2) *n TIMES ON LEFT* ([1, 1])	$s^n(0 + s^n(0)) + s^n(0) = s^n(0) + (s^n(0) + s^n(0))$
(1) *ON LEFT* ([2, 2, 1, 1])	$s^n(s^n(0)) + s^n(0) = s^n(0) + (s^n(0) + s^n(0))$
(2) *n TIMES ON RIGHT* ([2])	$s^n(s^n(0)) + s^n(0) = s^n(0 + (s^n(0) + s^n(0)))$
(1) *ON RIGHT* ([2, 2, 2])	$s^n(s^n(0)) + s^n(0) = s^n(s^n(0) + s^n(0))$
(2) *n TIMES ON LEFT* ([1])	$s^n(\underbrace{s^n(0)}_{\lambda} + \underbrace{s^n(0)}_{\lambda}) = s^n(\underbrace{s^n(0)}_{\lambda} + \underbrace{s^n(0)}_{\lambda})$
	EQUALITY

Figure 1: A General Proof of $\forall x\ (x + x) + x = x + (x + x)$

For the implementation it is necessary to provide (for the nth case) a description for the general proof in a constructive way (in this case a recursive way), which captures the notion that each $P(n)$ is being proved in a uniform way (from parameter n). This is done by manipulating $A(\underline{n})$, where $\forall x A(x)$ is the sequent to be proved, and using recursively defined function definitions of PA as rewrite rules, with the aim of reducing both sides of the equation to the same formula. The recursive function sought is described by the sequence of rule applications, parametrised over n. In practice, the first few proofs will be special cases, and it is rather the correspondence between the proofs of $P(99)$, say, and $P(100)$, which should be captured. The processes of generation of a (recursive) general proof from individual proof instances, and the (metalevel) checking that this is indeed the correct proof have been automated (see [Baker 92b]). Further details of the algorithms and representations used, together with the correspondence between the adopted implementational approach and the formal theory of the system are described in [Baker 92a]. Any appropriate inductive inference algorithm, such as Plotkin's least general generalisation [Plotkin 69], or that of Rouveirol, who has tackled the problem of controlling the hypothesis generation process to get only the most relevant candidates [Rouveirol 90], could be used to guess the general proof from the individual proof instances. In general, the complexity of the algorithm needed to guess a general proof from non-uniformly generated examples is exponential, whereas the stages of checking the general proof and suggesting a cut formula are only polynomially complex, and this is reflected in the time taken to produce the result. As an alternative, the user may bypass this whole stage by specifying the general proof directly.

The general proof representation represents $P(n)$, the proof of the nth numerator of the constructive ω-rule, in terms of rewrite rules applied $f(n)$ or a constant number of times to formulae (dependent upon the parameter n). As an example, the implementational representation of the general proof for $\forall x\ (x + x) + x = x + (x + x)$ takes the form given in Figure 1 (although it may be represented in a variety of ways) presuming that, within the particular formalisation of arithmetic chosen, one is given the axioms of addition of Figure 1.

By $s^n(0)$ is meant the numeral $\underline{n}$, ie. the term formed by applying the successor function n times to 0. The next stages use the axioms as rewrite rules from left to right, and substitution in the general proof, under the appropriate instantiation of variables, with the aim of reducing both sides of the equation to the same formula. The subpositions to which the rewrite rules are applied are given in parentheses, according to the *exp_at* notation of Clam [van Harmelen 89]. The general proof represents, and highlights, blocks of rewrite rules which are being applied. Meta-induction may be used (on the first argument) to prove the more general rewrite rules from one block to the next: for example, $\forall n\ s^n(x) + y = s^n(x + y)$ corresponds to n applications of axiom (2) above.

Effective trees selected by the implementation are shown to be correct, according to a given way of checking for correctness, which involves using induction over various datatypes. To reason about such trees, I work in a theory of trees and of the original syntax (which may be defined effectively). Defining equations for primitive recursive functions are taken as axioms, with a derived form as inference rules to allow rewriting of a formula in the obvious way. Furthermore, the system encapsulated by the implementation is shown to be sound with respect to $PA_{c\omega}$, although completeness is left as an open question. Casesplits and conditionals in the general proof correspond to branching in the effective tree representation.

Automated proof in such a system might be seen as a goal in itself, but it is also possible to use this sys-

tem as a guide to the provision of difficult proofs in more conventional systems. This is the concern of the following section.

5 An Application: Generalisation

Generalisation is a proof step which allows the postulation of a new theorem as a substitute for the current goal, from which the latter follows easily. It is a powerful tool with a variety of rôles, such as enabling proofs, defining new concepts, turning proofs for a specific example into ones valid for a range of examples and producing clearer proofs. Although generalisation is an important problem in theorem-proving, it has by no means been solved. It is important and still being investigated for reasons which have to do with cut elimination and the lack of heuristics for providing cut formulae. A cut elimination theorem for a system states that every proof in that system may be replaced by one which does not involve use of the cut rule[6]. Uniform proof search methods can be used for logical systems, in sequent calculus form, where the cut rule is not used. In general, cut elimination holds for arithmetical systems with the ω-rule, but not for systems with ordinary induction. Hence in the latter, there is the problem of generalisation, since arbitrary formulae can be cut in. This makes automatic theorem-proving very difficult, especially as there is no easy or fail-safe method of generating the required cut formula.

An important technique for investigating derivability in formal systems of arithmetic has been to embed such systems into semi-formal systems with the ω-rule. This section presents a new approach to the problems of generalisation by means of "guiding proofs" in the stronger system, which may succeed in producing proofs in the original system when other methods fail (cf. examples (9), (10), (13) of Table 1). The suggested methods are suitable for automation (and indeed the first two methods suggested below have been automated for simple arithmetical examples) and result in the suggestion of an appropriate cut formula.

Note that there is a class of proofs, including $\forall x\ (x + x) + x = x + (x + x)$, which are provable in PA only using the cut rule but which are provable in $PA_{c\omega}$ [Baker 92a]. I consider whether the proof in $PA_{c\omega}$ suggests a proof in PA, ie. in particular, what the *cut formula* would be in a proof in Peano Arithmetic? That is, what would the A be below?

$$\frac{A \vdash \forall x\ (x + x) + x = x + (x + x) \qquad \vdash A}{\vdash \forall x\ (x + x) + x = x + (x + x)} CUT$$

Ordinary induction does not work on $\forall x(x + x) + x = x + (x + x)$ (C), primarily because the second, third and sixth terms in the step case may not be broken down by the rewrite rules corresponding to (1) and (2) above, and hence fertilisation (substitution using the induction hypothesis) cannot occur. That is why we have to use the cut rule. We would wish A to be a more general version of C, so that we could prove $A \vdash C$, but on the other hand to be suitable to give an inductive proof, so that we could prove $\vdash A$ by induction. Hence we would be tackling the problem of generalisation by using an alternative (stronger) representation of arithmetic as a guide.

Three methods have been developed in order to suggest a cut formula from a general proof. The most basic is to see what remains unaltered in the nth case proof, and then write out the original formula, but with the corresponding term re-named. So, for the example in Figure 1, we would wish to rewrite the variable corresponding to λ as y. In this case, this would give

$$A \equiv \forall x \forall y\ (x + y) + y = x + (y + y).$$

A could then be proved by induction on x. Note that what is meant by 'unaltered' is defined by what is unaffected in structure by the rewrite rules. This procedure has been automated (all that is required is detection of the unaltered terms), and so the cut formula may be produced automatically. This method of generalisation will allow the proof of some theorems which pose a problem for other methods, such as $x \neq 0 \rightarrow p(x) + s(s(x)) = s(x) + x$, where p is the predecessor function (detailed comparisons of this 'unaltered term' method with other generalisation methods with regard to this example are given in [Baker 92a]).

A second method which encompasses this approach, but produces a more general generalisation, is to look at the rules of the general proof, and work out what the most general statement could be which was proved using these rules. This process has been applied in various other domains (see for example [Donat & Wallen 88]), and is the approach of explanation-based generalisation (denoted 'EBG' as an abbreviation). EBG is a technique for formulating general concepts on the basis of specific training examples, first described in [Mitchell 82]. In general terms the process works by generalising a particular solution to the most general possible solution which uses the rules of the original solution. It does this by applying these rules, making no assumptions about the form of the generalised solution, and using unification to fill in this form. The method is applied in this instance to a new domain, namely that of general proofs. The approach is described in detail in [Baker 92a], but as an illustration of the method, let us apply explanation-based generalisation to Figure 1, to give the process shown in Figure 2. The right hand column are the instantiations of variables, which are finally to be filtered back up into the original expression. Essentially what is happening is that the application of the rules are matched to see what the generalisation could be. So if (2) is applied m times, this will match with the

[6] See [Schwichtenberg 77], for example.

rules	form of general	proof	instantiations
(2) *n times at* [1, 1]	$fn0([s^n(X)+Y\|K])$	$= W$	*original*
(1) *once at* [2, 2, 1, 1]	$fn0([s^n(X+Y)\|K])$	$= W$	
(2) *n times at* [2]	$fn0([s^n(Y)\|K])$	$= W$	$X = 0$
(1) *once at* [2, 2, 2]	$fn0([s^n(Y)\|K])$	$= s^n(P+Q)$	$W = s^n(P)+Q$
(2) *n times at* [1]	$s^n(Y+K)$	$= s^n(Q)$	$P = 0,\ fn0 = +$
EQUALITY	$s^n(Y+K)$	$= s^n(Y+K)$	$Q = Y+K$

Figure 2: Illustration of Explanation-Based Generalisation on Rules of General Proof

form

$$s^m(X)+Y \Rightarrow s^m(X+Y)$$

Nothing more is supposed about the original form of the general proof than that it is of the form $U = W$. The rule application blocks on the left hand side of this figure are identical with those of the general proof given in Figure 1. The procedure is to form the most general general proof which could use those same rules to achieve equality. Hence, these same rewrite rules are applied at the specified subpositions to give a new general proof. In so doing the structure of U and W is revealed. For instance, the fact that rule (2) may be applied n times at subposition [1,1] of $U = W$ reveals that U must be of the form $fn0([s^n(X)+Y|K])$ (which represents some functor $fn0$ of as yet unknown arity with initial argument $s^n(X)+Y$ and additional arguments K) before the rule application, and of the form $fn0([s^n(X+Y)|K])$ afterwards. This process is repeated until all the given rules are exhausted. Finally, the left-hand side and the right-hand side of the general proof are unified (since the original proof resulted in equality). Throughout this process, information will have been obtained regarding the structure of some of the postulated variables in this new general proof, such as that presented in the final column of Figure 2.

Feeding such variable instantiation information back to the original expression $U = W$ shows that it must be of the form:

$$(\underline{n}+Y)+K = \underline{n}+(Y+K)$$

This gives the most general generalisation as being

$$\forall x\ \forall y\ \forall k\ (x+y)+k = x+(y+k)$$

The whole process is really just a term-matching exercise, and has been successfully automated.

The EBG method proposed in this section will succeed in the sense that there does exist some general proof such that a correct cut formula could be found by EBG (so long as inductive proof by generalisation apart, that is, generalisation by means of renaming some occurrences of the same variable in an expression, is possible). However, it will not necessarily work with any given general proof, nor if generalisation apart is not appropriate for the example under consideration.

Although the heuristic of replacing unaltered terms is suitable for implementation (and was successfully implemented), the method of explanation-based generalisation extends this idea to provide a uniform algorithm based on the underlying structure of the proof. The implementation of EBG described in [Baker 92a] follows the unification process described above, and thus subsumes the implementation of the heuristic method.

A third, more general, method of generalisation which subsumes the previous suggestions is provided by linearisation of the general proof, which suggests cut formulae in more complex cases (see [Baker 92a] since there is not the space to give further details in this abstract). In the natural number examples given above, the general proof is linear in the sense that the proof of $P(s(n))$ reduces to that of $P(n)$. However, in many examples involving lists, this is not so, and a new method for providing a cut formula is needed. The linearisation method suggests correct cut formulae for examples (12)–(14) of Table 1 (where <> denotes list concatenation). In particular, the generalisation of (12) is given as $\forall a\ \forall l\ len(rev(l) <> a) = len(rev(a) <> l)$, which is a better result (since it only requires one induction) than that more commonly suggested by other methods of $\forall a\ \forall l\ len(rev(l) <> a) = len(a <> l)$ (requiring two inductions).

A selection of the theorems proved by these methods is listed in Table 1.[7] Note that examples (3)–(9) and example (11) involve generalisation apart, or else generalisation of common subexpressions. In these cases both the heuristic of replacing unaltered variables and the explanation-based generalisation method work fairly straightforwardly. Although the examples listed in the table are of a similar simple form, these methods may also be applied to complicated examples containing nested quantifiers, etc., for the ω-rule applies to arbitrary sequents. Example (8) provides an instance of nested use of the ω-rule, which carries through directly. For example (10), the cut formula of $even(2.x)$ could possibly be extracted by a user from the form of the general proof, which is an improvement over other generalisation methods. However, in some cases where an ω-proof may be provided, it is not clear

[7] Definitions of the predicates involved may be found in [Baker 92a].

$\forall x\ x + s(x)$	$=$	$s(x + x)$	(3)
$\forall x\ (x + x) + x$	$=$	$x + (x + x)$	(4)
$\forall x\ x + s(x)$	$=$	$s(x) + x$	(5)
$\forall x\ x.(x + x)$	$=$	$x.x + x.x$	(6)
$\forall x\ (2 + x) + x$	$=$	$2 + (x + x)$	(7)
$\forall x\ \forall y\ (x + y) + x$	$=$	$x + (y + x)$	(8)
$\forall x\ x \neq 0 \rightarrow p(x) + s(s(x))$	$=$	$s(x) + x$	(9)
$\forall x\ even(x + x)$	$=$	$true$	(10)
$\forall l\ (l <> l) <> l$	$=$	$l <> (l <> l)$	(11)
$\forall l\ len(rev(l))$	$=$	$len(l)$	(12)
$\forall l\ rotate(len(l), l)$	$=$	l	(13)
$\forall l\ rev2(l, nil)$	$=$	$rev(l)$	(14)

Note that induction is blocked for the above expressions, but they may all be proved by the method proposed (namely by using the constructive ω-rule) and a correct cut formula produced as appropriate.

Table 1: Some Examples of Theorems Proved

what the cut formula might be.

These methods involving manipulation of the general proof should be compared with current generalisation methods. Of these, perhaps the most famous if that implemented by Boyer and Moore in their theorem-prover NQTHM [Boyer & Moore 79]. The main heuristic for generalisation is that identical terms occurring on both the left and right side of the equation are picked for rewriting as a new variable (with certain restrictions). This may be a quick method if it happens to work, but may also entail the proofs of many lemmas, which might need to be stored in advance in anticipation of such an event in order to be more efficient. The problems inherent in Boyer and Moore's approach have led Raymond Aubin to extend their work in this field [Aubin 75]. Aubin's method is to "guess" a generalisation by generalising occurrences in the definitional argument position, and then to work through a number of individual cases to see if the guess seems to work. If it does work, he will look for a proof. If it does not, then he will "guess" a different generalisation. However, Aubin's solution does not work in all cases. In particular, if a constructor such as a successor function appears in an original goal, together with individual variables, Aubin's method may result in over-generalisation or indeed no solution at all. The proposed guiding methods provide a uniform approach and do not have to check extra criteria, nor work through individual examples. Moreover, it is not possible to overgeneralise to a non-theorem (the method is sound but not complete — it does not always provide a solution, nor necessarily the best solution possible).

The suggested approaches also apply more generally to other data-types. Not only is it the case that certain new structural patterns may be seen in the general proof which may guide generalisation, but also that the general representation of an arbitrary object of that type (eg. $s^n(0)$ for natural numbers, $x_1 :: x_2 :: \ldots x_m :: nil$ for lists, etc.) enables the structure of that particular data-type to be exploited, in the sense that rewrite rules may be used which would not otherwise be applicable.

Hence a new method for generalisation has been proposed which is robust enough to capture in many cases what the alternative methods can do (in some cases with less work), plus it works on examples on which they fail.

6 A Proof Environment for the Constructive Omega Rule

This section describes the Constructive Omega Rule Environment (CORE), which is a proof development environment in which a (constructive) version of the ω-rule may be used as a rule of inference, and a system in which ω-proofs may be displayed and investigated. The implementation allows both the automatic or incremental construction of ω-proofs, and the validations of descriptions of ω-proofs. It is carried out within the framework of an interactive theorem-prover with Prolog as the tactic language.

Within CORE, any finitely large number of individual instances of proofs of a proposition may be generated automatically by the use of various tactics. The general representation of the proofs is provided by an inductive inference algorithm, which starts with an ini-

tial generalisation and then works by updating this general proof using the other individual proofs, until the general proof seems to have reached a stable form. This general proof is then automatically checked to see if it is indeed the correct one. There are two options which are allowable from a goal $\Gamma \vdash \forall x P(x)$. One is to ask to use the constructive ω-rule, whereby the system will check to see whether it can find a correct general proof, and then return to the former system and close the branch, or else report failure. The user may then continue to investigate other positions in the prooftree. The other option is to ask for an appropriate cut to be carried out in PA (the cut being worked out by the system from the general proof), with a further option to complete the tree as far as possible (using standard theorem-proving techniques). The general proof may be provided automatically, but there is an option in each case to switch temporarily to another system which will allow for the description, manipulation and display of the general proof. The user may specify the proof incrementally, in terms of applications in positions in the tree, plus induction over a distinguished parameter, or all at once — and this is checked. The system builds up a recursive function description of the general proof, and is able to display individual proofs in addition to the general case. A cut formula is automatically suggested from general proofs using an implementation based on the method of explanation-based generalisation, which is a technique for formulating general concepts on the basis of specific training examples, first described in [Mitchell 82] (see [Baker 92a]). [Baker 92b] provides details of the proof development systems upon which the implementation is based; representation of the ω-rule and its subgoals; generation of individual proofs; the application of rewrite rules; provision of a general proof; correctness checking of the general proof (using meta-induction); generalisation, and finally, the interactive system, and how to use it.

Current work involves reimplementation of such a system of arithmetic with the ω-rule, plus the generalisation methods, within Isabelle. The latter is a generic proof development environment [Paulson 86] which provides a sophisticated environment for development of these strategies across many different datatypes.

7 Conclusions

In conclusion, implementation of a system of arithmetic with the ω-rule has been carried out within the framework of an interactive theorem-prover with Prolog as the tactic language. This approach works for theories other than arithmetic and logics other than a sequent version of the predicate calculus, and may rather be regarded as suggesting a general framework. So long as a procedure for constructing a proof for each individual of a sort is specified, universal statements about objects of the sort could be proved. Thus it appears that the approach described in this paper may be an aid to automated deduction, and in addition provides a mechanism for guiding proofs in more conventional systems.

Acknowledgements

I would like to acknowledge the Science and Engineering Research Council for funding the research reported in this paper, the help of Alan Smaill, and many others, from the Mathematical Reasoning group in Edinburgh University, plus the Isabelle group in the Cambridge Computer Laboratory.

References

[Aubin 75] R. Aubin. Some generalization heuristics in proofs by induction. In G. Huet and G. Kahn, editors, *Actes du Colloque Construction: Amélioration et vérification de Programmes.* Institut de recherche d'informatique et d'automatique, 1975.

[Baker 92a] S. Baker. *Aspects of the Constructive Omega Rule within Automated Deduction.* Unpublished PhD thesis, University of Edinburgh, 1992.

[Baker 92b] S. Baker. CORE manual. Technical Paper 10, Dept. of Artificial Intelligence, Edinburgh, 1992.

[Baker *et al* 92] S. Baker, A. Ireland, and A. Smaill. On the use of the constructive omega rule within automated deduction. In A. Voronkov, editor, *International Conference on Logic Programming and Automated Reasoning - LPAR 92, St. Petersburg*, Lecture Notes in Artificial Intelligence No. 624, pages 214–225. Springer-Verlag, 1992.

[Boyer & Moore 79] R.S. Boyer and J.S. Moore. *A Computational Logic.* Academic Press, 1979. ACM monograph series.

[Bundy *et al* 91] A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill. Rippling: A heuristic for guiding inductive proofs. Research Paper 567, Dept. of Artificial Intelligence, Edinburgh,

1991. To appear in The Journal of Artificial Intelligence.

[Donat & Wallen 88] M.R. Donat and L.A. Wallen. Learning and applying generalised solutions using higher order resolution. In E. Lusk and R. Overbeek, editors, *Lecture Notes in Computer Science*, volume 310, pages 41–60. Springer-Verlag, 1988.

[Feferman 62] S. Feferman. Transfinite recursive progressions of axiomatic theories. *Journal of Symbolic Logic*, 27:259–316, 1962.

[Kreisel 65] G. Kreisel. Mathematical logic. In T.L. Saaty, editor, *Lectures on Modern Mathematics*, volume III, pages 95–195. John Wiley and Sons, 1965.

[López-Escobar 76] E.G.K. López-Escobar. On an extremely restricted ω-rule. *Fundamenta Mathematicae*, 90:159–72, 1976.

[Mitchell 82] T.M. Mitchell. Toward combining empirical and analytical methods for inferring heuristics. Technical Report LCSR-TR-27, Laboratory for Computer Science Research, Rutgers University, 1982.

[Nelson 71] G.C. Nelson. A further restricted ω-rule. *Colloquium Mathematicum*, 23, 1971.

[Paulson 86] L. Paulson. Natural deduction as higher order resolution. *Journal of Logic Programming*, 3:237–258, 1986.

[Plotkin 69] G. Plotkin. A note on inductive generalization. In D Michie and B Meltzer, editors, *Machine Intelligence 5*, pages 153–164. Edinburgh University Press, 1969.

[Prawitz 71] D. Prawitz. Ideas and results in proof theory. In J.E. Fenstad, editor, *Studies in Logic and the Foundations of Mathematics: Proceedings of the Second Scandinavian Logic Symposium*, volume 63, pages 235–307. North Holland, 1971.

[Rosser 37] B. Rosser. Gödel-theorems for non-constructive logics. *JSL*, 2(3):129–137, September 1937.

[Rouveirol 90] C. Rouveirol. Saturation: Postponing choices when inverting resolution. In *Proceedings of ECAI-90*, pages 557–562, Stockholm, August 1990.

[Schütte 77] K. Schütte. *Proof Theory*. Springer-Verlag, 1977.

[Schwichtenberg 77] H. Schwichtenberg. Proof theory: Some applications of cut-elimination. In Barwise, editor, *Handbook of Mathematical Logic*, pages 867–896. North-Holland, 1977.

[Shoenfield 59] J.R. Shoenfield. On a restricted ω-rule. *Bull. Acad. Sc. Polon. Sci., Ser. des sc. math., astr. et phys.*, 7:405–7, 1959.

[Takeuti 87] G. Takeuti. *Proof theory*. North-Holland, 2 edition, 1987.

[Tucker *et al* 90] J.V. Tucker, S.S. Wainer, and J.I. Zucker. Provable computable functions on abstract-data-types. *Lecture Notes in Computer Science*, 443:660–673, 1990.

[van Harmelen 89] F. van Harmelen. The CLAM proof planner, user manual and programmer manual: version 1.4. Technical Paper TP-4, DAI, 1989.

[Yoccoz 89] S. Yoccoz. Constructive aspects of the omega-rule: Application to proof systems in computer science and algorithmic logic. *Lecture Notes in Computer Science*, 379:553–565, 1989.