Linearisation of ω - Proofs: A New Method of Generalisation Within Automated Deduction

Siani Pearson
Model-Based Systems Department
HP Laboratories Bristol
HPL-95-93
August, 1995

generalisation, proof transformation, automated theorem proving

Generalisation is a major 'open' problem in theorem proving which must often be addressed when attempting automation of proofs involving mathematical induction. This paper proposes a new, uniform method of generalisation, involving the transformation of proofs, which encompasses many different types of generalisation and which may succeed when other methods fail.

Internal Accession Date Only

Linearisation of ω-Proofs: A New Method of Generalisation within Automated Deduction

Siani L. Pearson*

Hewlett Packard Research Laboratories, Bristol BS12 6QZ
e-mail: siani@hplb.hpl.hp.com.uk
tel: +44-117-9228438

Abstract. Generalisation is a major 'open' problem in theorem-proving which must often be addressed when attempting automation of proofs involving mathematical induction. This paper proposes a new, uniform method of generalisation, involving the transformation of proofs, which encompasses many different types of generalisation and which may succeed when other methods fail.

1 Introduction

This paper presents a new method of generalisation, useful within the field of automated deduction of mathematical proofs. The following section discusses the problem of generalisation within this area. Section 3 presents ω-proofs, from which it is possible to read off appropriate cut formulae, and the semi-formal system of arithmetic in which they are derived. Section 4 discusses how an explanation-based generalisation method may be applied to these proofs in order to suggest cut formulae. Section 5 considers how linearisation and inductive proofs are related to ω-proofs. Moreover, a theorem is proven by which it is demonstrated that linearisation corresponds to inductive proof. In Section 6 a linearisation method of generalisation is proposed, with reference to examples. Section 7 describes a proof environment which implements many of these ideas, and Section 8 presents a comparison with related work. Finally, conclusions are given.

2 The Problem of Generalisation

Generalisation is a powerful tool in automated theorem proving 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. Van der Waerden's account of how the proof of Baudet's conjecture was found [33] illustrates how generalisation lies at the heart of mathematical discovery, and how generalised theorems may be easier to prove than the original goal (because the induction hypothesis is also made stronger when the goal is strengthened by generalisation). If induction is blocked for an expression[2], generalisation may be used as a step to convert this expression into a new, more general, expression which may be proven by induction. Thus in order to verify a goal, one may instead prove its generalisation, and indeed, as discussed in Section 4, in some cases it is actually *necessary* to adopt this approach.

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 relate to 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[3]. Uniform proof search methods can be used for logical systems, in sequent calculus form, where the cut rule is not used. In

* née Baker

[2] Induction is said to be blocked if, after all available symbolic evaluation has been carried out, the induction conclusion is still not an instance of the induction hypothesis, and hence remains unprovable.

[3] There are many forms of the cut rule: common variants are included in most logic text books, such as [31]. A form of the classical cut rule is:

$$\frac{\Gamma \vdash A, \Delta \quad \Lambda, A \vdash \Pi}{\Gamma, \Lambda \vdash \Delta, \Pi}$$

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.

When discussing generalisation I have so far referred to goal generalisation, which is the proof step described above which allows the postulation of a new theorem as a substitute for the current goal, from which the latter follows easily (for example, backward application of the $\forall$-elim rule of a natural deduction calculus). Yet generalisation may be carried out on proofs as well: it has a different emphasis, namely on providing a more general proof, and although goal generalisation may involve generalisation of the proof of a theorem in order to generalise the theorem itself, this need not be the case.

Generalisation of proofs has the advantage that more information may be available than for goal generalisation. The method proposed in this paper exploits this fact by enabling goal generalisation by means of carrying out linearisation on proofs. The problem of generalisation is tackled by use of an alternative (stronger) representation of arithmetic, in which proofs may be more easily generated. The "guiding proofs" in this stronger system may succeed in producing proofs in the original system when other methods fail (cf. Table 1). This method has been automated for simple arithmetical examples and results in the suggestion of an appropriate cut formula. The type of examples considered are typically of the form Π_1^0. More specifically, in order to carry out generalisations of the form $\forall x A(x)$ to $\forall x Q(x)$, individual proof instances of $A(x)$ are generated, and then generalised to a proof of the arbitrary case $A(r)$ using some inductive inference process (which presents algorithms to obtain a general pattern from individual instances). The latter proof can be generalised to a new proof of a linear form. The formula for which this is a proof provides a suitable cut formula for $A(x)$, since it satisfies the properties of being a more general form of which A is a specific case, and such that induction may be performed upon it. Figure 1 represents the overall strategy for generalisation: a "general proof" (of $A(r)$) is provided by some means (one possible option being an automatic derivation using the ω-rule), and a suitable cut formula is suggested by inspection of this proof.

3 Automation of Proofs in PA_{ω}

In order to describe the generalisation method proposed, it is first necessary to provide a description of the 'stronger' system mentioned in the previous section. A suitable rule other than induction which might be added to Peano's axioms to form a system formalising arithmetic is the ω-rule:

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

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)[4], 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 (generating the premises of the ω-rule) 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 $A(\underline{n})$ [31]. 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 proof-tree and the function

where Γ, Π, Δ and Λ are arbitrary lists of formulae, and A is an arbitrary formula. The left-hand premise is the generalisation, and the right-hand premise is a justification that it is indeed a generalisation, i.e. that its addition to the hypotheses results in a proof of the original sequent. A is referred to as the *cut formula*, since it is a formula which is "cut in"; it is new in terms of top-down proof search.

[4] See for example [31] for a formalisation.

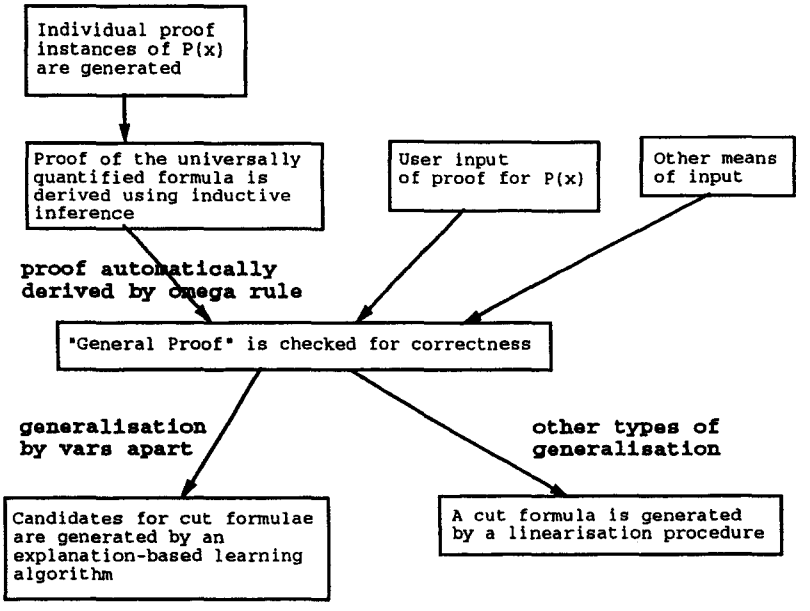

Fig. 1. Generalisation Strategy

describing the use of the different rules must be recursive) [34], which is the basis taken for implementation (as opposed to a Gödel numbering approach). The sequent calculus enriched with the constructive ω-rule (let us call it $PA_{c\omega}$) has cut elimination, and is complete [30]. Moreover, since the ω-rule implies the induction rule (cf. Figure 8), $PA_{c\omega}$ + *induction* is a conservative extension of $PA_{c\omega}$. There are many versions of a restricted ω-rule; this one has been chosen because it is suitable for automation. Note that in particular this differs from the form of the ω-rule (involving the notion of provability) considered by Rosser [26] and subsequently Feferman [11]. Implementation of a proof environment with the constructive ω-rule, which provides a basis for the implementation of the generalisation method described in this paper, is described in Section 7. 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 completeness is not claimed, some proofs that normally require generalisation can be generated more easily in $PA_{c\omega}$ than PA.

3.1 $PA_{c\omega}$: 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 proof-tree labelled at the root with the given formula. Syntactical details about this system PA_ω are given in [21, P162] (see [25, 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 [28, 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 proof-trees 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 [29, P886]. By adding the provability relation and numeric encoding, a reflection system which necessarily extends the original one may be formed [19, P163].[5] However, the necessity of using this Gödel numbering

[5] The standard approach is to use numeric encoding (using notation $\lceil\ \rceil$) and strengthen PA by adding an

approach may be avoided by following Tucker in defining primitive recursion ("effectiveness") over various data-types that are better adapted to computational purposes [32]. 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. Hence we define

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

$PA_{c\omega}$ may be defined as a (semi-)formal system by further specifying axioms and rules of inference (in this case, corresponding to those of PA: "this wholesale carry over of derived rules from predicate logic is one of the special virtues of cut free infinite proofs" [19, P166]).

Thus the objects of interest in $PA_{c\omega}$ are recursive (possibly infinite) proof-trees (in the sense of López-Escobar [21]), labelled with formulae (namely, the sequents to be proven at each point) and rules. 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 proof-tree labelled at the root with the given formula. More specifically, by an effective proof-tree we understand a function which returns for each potential position in the tree either a pair representing the sequent and rule associated with that position, or a token, $\check{\in}$, indicating that the position is outside the tree. Positions are given by a list of positive integers referring to the path through the tree from the root.

Definition 1 Effective prooftree for $PA_{c\omega}$. f is an effective prooftree for $PA_{c\omega}$ if and only if f is an effective function $f : \textit{nat list} \rightarrow \textit{seq} \times \textit{rule}$ such that f is well-founded and correct.

Definition 2 Effective function. The effective functions over a datatype $\mathcal{D}$ are the smallest set closed with regard to the defined basic functions, composition and recursion over that datatype.

The basic functions are the projection functions (defined standardly [31]), the constructor function of $\mathcal{D}$ ($cf_{\mathcal{D}} : t_1 \times \ldots \times t_{p-1} \times \mathcal{D} \rightarrow \mathcal{D}$, where $t_1, \ldots, t_{p-1}$ range over sorts and may be omitted), and the zero function(s), which map every element of $\mathcal{D}$ to a constant of $\mathcal{D}$ ($const_{\mathcal{D}}$). For example, for *nat*, the constant is 0 and the constructor function is the successor function $s : \textit{nat} \rightarrow \textit{nat}$; for t *list*, [ ] and $\textit{cons} : t \times t\ \textit{list} \rightarrow t\ \textit{list}$ respectively; for *string*, C and string concatenation $\hat{*} : \textit{string} \times \textit{string} \rightarrow \textit{string}$ respectively, etc. Recursion and composition are defined as follows:
Composition: If $g : \textit{type}_1 \times \ldots \times \textit{type}_j \rightarrow \textit{type}$ and $h_i : \textit{type}_{S_1} \times \ldots \times \textit{type}_{S_k} \rightarrow \textit{type}_i$ for $1 \leq i \leq j$ are effective, then $f : \textit{type}_{S_1} \times \ldots \times \textit{type}_{S_k} \rightarrow \textit{type}$ is effective, where for $x_i \in \textit{type}_{S_i}$:

$$f(x_1, \ldots, x_k) = g(h(x_1, \ldots, x_k), \ldots, h_j(x_1, \ldots, x_k))$$

Recursion: If $g_i : \textit{type}_1 \times \ldots \times \textit{type}_k \rightarrow \textit{type}$ and $h : \textit{type}_1 \times \ldots \times \textit{type}_k \times \mathcal{D} \ldots \times \mathcal{D} \times \textit{type} \ldots \times \textit{type} \rightarrow \textit{type}$ are effective, then so is $f : \textit{type}_1 \times \ldots \times \textit{type}_k \times \textit{type}_i \rightarrow \textit{type}$, where the n_i may be omitted and:

$$f(n_1, \ldots, n_k, const_{\mathcal{D}_i}) = g_i(n_1, \ldots, n_k)$$

$$f(n_1, \ldots, n_k, cf_{\mathcal{D}}(x_1, \ldots, x_p)) = h(n_1, \ldots, n_k, x_1, \ldots, x_p, f(n_1, \ldots, n_k, x_1), \ldots, f(n_1, \ldots, n_k, x_p))$$

Definition 3 Order in tree. Define the relation $\leq$ on *nat list* $\times$ *nat list* by:

$$\textit{Pos1} \leq \textit{Pos2} \leftrightarrow \exists l\ \textit{Pos2} = \textit{Pos1} <> l,\ l \in \textit{nat list}$$

Definition 4 Derivation in $PA_{c\omega}$. $f : \textit{nat list} \rightarrow \textit{seq} \times \textit{rule}$ describes a derivation in $PA_{c\omega}$ for the sequent Φ, where Φ is the sequent at the top of the tree (viz. at the node [ ]) if:

1. $\{p : \textit{nat list} | f(p) \neq \check{\in}\}$ is a well-founded tree according to $\leq$.
2. If $f(p) \neq \check{\in}$, then $q_2^1(f(p))$ is a sentence associated with the node p (namely the sequent to be proved), and $q_2^2(f(p))$ is the name of a rule of $PA_{c\omega}$ used to produce its immediate successors, where q_i are projection functions such that $q_2^1(A, B) = A$ and $q_2^2(A, B) = B$.

---

arithmetic schema of the form: $(\exists \Pi\ (\text{proof-tree}(\Pi) \wedge \text{conc}(\Pi) = \lceil \Phi \rceil)) \rightarrow \Phi$.

3. If p is a bottommost node in the tree, i.e. $f(q) = \check{\in}$ for all q such that $p \leq q$, and $f(p) \neq \check{\in}$, then either $q_2^1(f(p))$ is an axiom of $PA_{c\omega}$ and $q_2^2(f(p))$ is *axiom*, or else $f(p)$ is set to incomplete to indicate that the tree is incomplete.
4. If $Pos <> [K]$ ($K \in \mathbb{N}$) is not a bottommost node, then $q_2^1(f(Pos <> [K]))$ is the Kth subgoal of $q_2^2(f(Pos <> [K]))$ applied to $q_2^1(f(Pos))$.

Definition 5 Incomplete tree. The derivation is incomplete if not all the leaves are closed i.e. if incomplete is associated with any node in the tree.

Definition 6 Prooftree. The derivation will be a prooftree if it is a complete derivation, in other words if all its leaves are axioms.

The approach described above is suitable for automation, since it generates the subgoals of the ω-rule, rather than having to check their presence.

Properties of such primitively recursively defined trees can be proven using induction principles associated with the datatypes, as seen in [4]. To check for local correctness, structural induction is used over the prooftrees: the function gives a $PA_{c\omega}$-proof if, at every node of the tree, the formulae at the sub-nodes and the formula at the node are correctly related according to the rule of inference associated with the node. These are the sorts of proofs that have been automated by [7], and the simpler proofs that arise here have been automated by this method. 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 [4]: in the representation of $PA_{c\omega}$, a derived form of primitive recursive function-defining equations as inference rules is required to allow rewriting of formulae (which takes place in ω-proof examples, and hence an analogy is needed). It is convenient to define these as single steps, as shorthand in order to avoid going through all the equivalent tedious steps of PA. A simple example of such a derived step would be the justification of $s(0+0) = s(0)$ from $s(0)+0 = s(0)$, given the rewrite rule $s(x)+y \rightarrow s(x+y)$.

It would be extremely cumbersome to build up such tree functions explicitly, and we do not do this in practice. However, the system is intended to ensure that such a tree is constructible whenever an ω-rule application is shown correct.

3.2 ω-Proofs

The constructive ω-rule may be used 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. One motivation behind use of the ω-rule is that such an approach may be seen as an attempt to provide induction systems with the sub-formula property: systems which include the ordinary induction rule lose the sub-formula property (namely, that each formula in the derivation does not necessarily have to be composed of sub-formulae in the preceding derivation) due to the introduction of arbitrary "cut formulae". Another motivation is that the resulting proofs may seem to be more intuitive than standard inductive proofs of the same theorems, in the sense of corresponding more closely to the way in which people convince themselves of the correctness of the proof: philosophical induction (from "trivial" test cases) may sometimes be the means of construction by humans of a generic proof for case n, and in addition, the generic proof itself sometimes has some psychological validity. So, automated proof in such a system might be seen as a goal in itself, but the concern of this paper is rather how it is possible to use this system as a guide to the provision of difficult proofs in more conventional systems.

As discussed above, one way in which the constructive ω-rule may be put into effect is to require that there is an enumeration of the derivations which prove the premises — for example one could code proofs by numbers, by means of a primitive recursive function which generates them. But I have not used such a traditional representation; it was sufficient for my purposes to provide (for the nth case) a description for the proof, $P(n)$, of the nth subgoal after application of the constructive ω-rule to $\forall x A(x)$, 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 description shall be called an ω-proof of A. 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.

$$\begin{array}{ll}
 & \vdash (\underline{n}+\underline{n})+\underline{n} = \underline{n}+(\underline{n}+\underline{n}) \\
\underline{n} \equiv s^n(0) & \vdash (s^n(0)+s^n(0))+s^n(0) = s^n(0)+(s^n(0)+s^n(0)) \\
USE\ (A2)\ n\ TIMES\ ON\ LEFT\ ([1,1]) & \vdash s^n(0+s^n(0))+s^n(0) = s^n(0)+(s^n(0)+s^n(0)) \\
USE\ (A1)\ ON\ LEFT\ ([2,2,1,1]) & \vdash s^n(s^n(0))+s^n(0) = s^n(0)+(s^n(0)+s^n(0)) \\
USE\ (A2)\ n\ TIMES\ ON\ RIGHT\ ([2]) & \vdash s^n(s^n(0))+s^n(0) = s^n(0+(s^n(0)+s^n(0))) \\
USE\ (A1)\ ON\ RIGHT\ ([2,2,2]) & \vdash s^n(s^n(0))+s^n(0) = s^n(s^n(0)+s^n(0)) \\
USE\ (A2)\ n\ TIMES\ ON\ LEFT\ ([1]) & \vdash s^n(\underbrace{s^n(0)}_{\lambda}+\underbrace{s^n(0)}_{\lambda}) = s^n(\underbrace{s^n(0)}_{\lambda}+\underbrace{s^n(0)}_{\lambda})
\end{array}$$

Fig. 2. An ω-Proof of $\forall x\ (x+x)+x = x+(x+x)$

Definition 7 ω-proof. The ω-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 ω-proof for $\forall x\ (x+x)+x = x+(x+x)$ takes the form given in Figure 2 (although it may be represented in a variety of ways) presuming that, within the particular formalisation of arithmetic chosen, one is given the axioms $0+y = y$ $(A1)$ and $s(x)+y = s(x+y)$ $(A2)$.

It is customary to use recursion equations as rewrite rules in order to evaluate expressions. Since we have a substitution rule, we can show that such applications to quantifier-free formulae are sound. For example, the axiom (A2) gives us the rewrite $s(X)+Y \Rightarrow s(X+Y)$. The rewrite rule of inference, given a goal and a specified sub-term that matches the left hand side of the rewrite, yields as the single subgoal the rewritten goal. Note that any use of this derived rule could be expanded into a small proof tree of fixed shape in our original theory. In reasoning about ω-proofs, we will chiefly use this derived rule.

By $s^n(0)$ is meant the numeral $\underline{n}$, i.e. 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 ω-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, where positions are lists of integers representing tree co-ordinates in the syntax-tree of the expression, but in reverse order. The ω-proof represents, and highlights, blocks of rewrite rules which are being applied. 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 (A2) above.

The processes of generation of a (recursive) ω-proof from individual proof instances, and the (meta-alevel) checking that this is indeed the correct proof have been automated (see [2]). Any appropriate inductive inference algorithm[6], such as Plotkin's least general generalisation [24], or that of Rouveirol, who has tackled the problem of controlling the hypothesis generation process to get only the most relevant candidates [27], could be used to guess the ω-proof from the individual proof instances. In the CORE implementation, the theorem under consideration is proved automatically for a few cases, and then learning induction is used to guess the ω-proof from these cases; next it is automatically verified that the guessed ω-proof is correct by means of verifying each of the rewrite steps and checking the leaf as an axiom. (Verifying that the guessed ω-proof works for all n involves mathematical induction at the meta-level; however, the induction required for this differs from that required to prove the original theorem directly,

[6] Explanation-based learning permits learning from a single example, whereas inductive learning usually requires many examples to learn a concept.

and is usually simpler, so there is no circularity.) 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 [2, 4]. Note that both the rewrite rules used in the ω-proof and their application positions are automatically generated.

In general, the complexity of the algorithm needed to guess an ω-proof from non-uniformly generated examples is exponential, whereas the stages of checking the ω-proof and suggesting a cut formula are less 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 ω-proof directly. Meta-induction is used to ensure that the proposed general rule applications do indeed give a proper proof when applied to the general case of the sequent to be proven. Note that such inductive inference algorithms for generating generalisations produce a proof for an arbitrary instance: it is suggested below how linearisation can be applied using this information to find the proper induction formulae for inductive theorem provers. The problems involved in the inductive inference process differ from those involved in (goal) generalisation because inductive inference generalises a proof of $A(k)$ where k is some number to a proof of $A(n)$, whereas generalisation involves finding a formula C such that $C \vdash A$ and in addition such that C may be proven by induction. Inductive inference may be carried out by a relatively simple algorithm (such as updating a guess). However, there is no such known algorithm for goal generalisation.

Such ω-proofs are not simply disguised PA-proofs, since the system $PA_{c\omega}$ is a logically stronger system than that of PA [30]. The next section discusses how ω-proofs may be generalised.

4 Explanation-Based Generalisation Applied to ω-Proofs

There is a class of proofs which are provable in PA only using the cut rule but which are provable in $PA_{c\omega}$[7] [2, 20]. I shall consider whether the proof in $PA_{c\omega}$ suggests a proof in PA, and in particular, what the cut formula would be in a proof in Peano Arithmetic.

To illustrate the general principle, consider the simple example for $\Psi \equiv \forall x\ (x+x)+x = x+(x+x)$. The problem is to find Φ such that Φ should be a more general version of the goal Ψ, in order to prove $\Phi \vdash \Psi$, but on the other hand Φ must be just general enough to provide a proof by induction whilst not being so general that it is not a theorem.

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

Ordinary induction does not work on Ψ, 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 so fertilisation (substitution using the induction hypothesis) cannot occur. Hence it is necessary to use the cut rule.

In order to suggest a cut formula from an ω-proof, one method 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 [5]. So, for the example in Figure 2, the variable corresponding to λ would be rewritten as y. In this case, this would give

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

Φ 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 [5]).

Generalisation using Explanation-Based Generalisation. A new development of this solution, which produces a more general generalisation, is to look at the rules of the ω-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, and is the approach of explanation-based generalisation (denoted 'EBG' as

[7] Thus, without having to use the cut or induction rules.

an abbreviation). EBG is a technique for formulating general concepts on the basis of specific training examples, first described in [23]. 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.

rules of ω-proof	generalised ω proof	instantiations
(A2) *n times at* [1,1]	$\vdash fn0([s^n(X)+Y\|K]) = W$	*original*
(A1) *once at* [2,2,1,1]	$\vdash fn0([s^n(X+Y)\|K]) = W$	
(A2) *n times at* [2]	$\vdash fn0([s^n(Y)\|K]) = W$	$X = 0$
(A1) *once at* [2,2,2]	$\vdash fn0([s^n(Y)\|K]) = s^n(P+Q)$	$W = s^n(P)+Q$
(A2) *n times at* [1]	$\vdash s^n(Y+K) = s^n(Q)$	$P = 0,\ fn0 = +$
	$\vdash s^n(Y+K) = s^n(Y+K)$	$Q = Y+K$

Fig. 3. Illustration of Explanation-Based Generalisation on Rules of ω-Proof

The EBG method is applied in this instance to a new domain, namely that of ω-proofs. In order to produce a generalised proof of an ω-proof $\mathcal{A}$, the procedure is to apply the rules of $\mathcal{A}$ to the sequent $\vdash U = W$, and use higher-order unification to fill in the rest of the structure. As an illustration of the method, let us apply explanation-based generalisation to Figure 2, to give the process shown in Figure 3. The right hand column is the instantiations of variables, which are finally to be filtered back up into the original expression using constraint back-propagation. Essentially what is happening is that each application of the rewrite rules of the original ω-proof is matched with the latest line of the new ω-proof to see the necessary form of a generalised ω-proof. If (A2) is applied m times, this will match with the form $s^m(X) + Y \Rightarrow s^m(X + Y)$. Nothing more is supposed about the original form of the ω-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 ω-proof given in Figure 2. The procedure is to form the most general ω-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 ω-proof. In so doing the structure of U and W is revealed. For instance, the fact that rule (A2) 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 ω-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 ω-proof, such as that presented in the final column of Figure 3. 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 $\forall x\, \forall y\, \forall z\ (x+y)+z = x+(y+z)$. By means of this method, the generalisations are found by recognising patterns, and a uniform approach for generalisation is provided.

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, in which the previous "ω-rule" environment was extended to include generalisation tactics, follows the unification process described above, and thus subsumes the implementation of the heuristic method. Section 10 provides details of the implementational environment, which will automatically suggest cut formulae for examples such as the arithmetical examples of Table 2. Most of these examples involve generalisation of variables 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. The method may also be applied to complicated examples containing nested quantifiers, etc., for the ω-rule applies to arbitrary sequents. $\forall x \forall y\ (x + y) + x = x + (y + x)$ provides an instance of nested use of the ω-rule, which carries through directly. In some cases a cut formula could possibly be extracted by a user

from the form of the ω-proof; however, in other cases where an ω-proof may be provided, it is not clear what the cut formula might be.

This section has presented various new methods of generalisation. The following section considers how conventional inductive proofs are related to ω-proofs in the context of explanation-based generalisation and linearisation. It shall be shown later (in Subsection 5.3) that EBG linearises ω-proofs (and therefore, the initial formula of the generalised ω-proof may be proven by induction).

5 Linearity and Induction

5.1 Linearisation

The general form of a linear proof involving a single use of the constructive ω-rule is given in Figure 4,[8] with the proviso that there may be additional branches if the $P(i)$ consist of branching proof rules. Any proof which is of the form of Figure 4 (with additional leaves in the case of inductive proofs with branching in the step proof) shall be called *linear*, with the constraints that $A(\underline{k})$ is reduced to $A(\underline{k-1})$ in a uniform way for each k, and that the proof is cut-free. In the case of the constructive ω-rule operating on $\forall x\ A(x)$, the $A(i)$ are uniformly generated, and there will be a relationship between their proofs — otherwise, this may not be the case. An ω-proof is a parametrised version of an arbitrary subtree of a proof in $PA_{c\omega}$ of a universal statement; thus, a "linear" ω-proof will correspond to the kth subtree of Figure 4. In the simpler examples of the use of the ω-rule, there is only one application of that rule, and by using the rewrite inference rule we can regard each subsequent branch of the proof as linear, with no further branching of the tree.

$$\dfrac{\begin{array}{ccccccc} & & & & \Gamma \vdash A(0) & & \\ & & & & \vdots\ \textit{kth subtree} & & \\ & \Gamma \vdash A(0) & & & \Gamma \vdash A(\underline{k-1}) & & \\ \vdots\ P(0) & \vdots\ P(1) & & & \vdots\ P(k) & & \\ \Gamma \vdash A(0) & \Gamma \vdash A(\underline{1}) & \ldots & \Gamma \vdash A(\underline{k}) & & \ldots & \end{array}}{\Gamma \vdash \forall x\ A(x)}\ \textit{constructive } \omega \textit{ rule}$$

Fig. 4. Form of Linear Proof in $PA_{c\omega}$

Definition 8 Linear proof in $PA_{c\omega}$:. A proof of $A(x)$ in $PA_{c\omega}$ is *linear* if there exist rules $R_1, \ldots, R_p$ which are rules or derived rules of $PA_{c\omega}$ such that there is a cut-free derivation in $PA_{c\omega}$ of

$$\begin{array}{c} \Gamma \vdash A(\underline{k}) \\ \vdots \\ \Gamma \vdash A(s(\underline{k})) \end{array}$$

for arbitrary k (all other branches closing with axioms) and the rules applied in the proof are $B_1, \ldots, B_m$ where $B_i = R_j$, or $B_i = R_j^{f(k)}$, and $1 \leq i \leq m$, $1 \leq j \leq p$, $f : \mathbb{N} \rightarrow \mathbb{N}$ is a primitive recursive function and R^n denotes the rule R being applied n times.

It shall be shown that ω-proofs may be turned into a linear form, which will suggest that induction can be done on the original formula (that is to say that this formula is a suitable cut formula). Hence it is necessary to recognise Q such that an original ω-proof for $P(x)$ may be transformed into a "linearised" ω-proof of form $Q(s(n))$ reducing in a uniform manner (via $P(s(n))$) to $Q(n)$ to $Q(n-1)$ down to $Q(0)$, which reduces to an equality using $P(0)$. The required proof of $\forall x\ P(x)$ in PA is:

[8] Use of two inductions should be compared with P. Madden's transformations which linearise proofs defined by functions with two inductive variables [22].

$$\dfrac{\forall x\, Q(x) \vdash \forall x P(x) \qquad \dfrac{\begin{array}{cc} \textit{CLOSES} & \textit{CLOSES} \\ \vdots\, P(0) & \vdots\, P(s(r)) \\ \vdash Q(0) & Q(r) \vdash Q(s(r)) \end{array}}{\vdash \forall x\, Q(x)}\, ind(x)}{\vdash \forall x\, P(x)}\, cut$$

If the ω-proof is of a linear form, it is the case that $\forall x\, P(x)$ may be proven in this manner. The fact that a proof of $\Gamma, A(\underline{k}) \vdash A(s(\underline{k}))$ exists provided there is a derivation of $\Gamma \vdash A(s(\underline{k}))$ from $\Gamma \vdash A(\underline{k})$ (in the sense of there being a natural deduction proof of the former from the latter, possibly using additional axioms) forms an analogue of the deduction theorem.[9]

Theorem 9 Deduction Theorem for Sequents. *The sequent $\Gamma, A \vdash B$ is provable in $PA_{c\omega}$ if and only if there is a cut-free derivation in $PA_{c\omega}$ of*

$$\begin{array}{c} \Gamma \vdash A \\ \vdots \\ \Gamma \vdash B \end{array}$$

Proof. $\Rightarrow$*: Suppose $\Gamma, A \vdash B$ is provable in $PA_{c\omega}$. The cut rule may be used to derive $\Gamma \vdash B$ from $\Gamma \vdash A$ and $\Gamma, A \vdash B$. Hence, given $\Gamma, A \vdash B$, then there is a derivation of $\Gamma \vdash B$ from $\Gamma \vdash A$. By the cut elimination theorem, there must be a cut-free derivation (in $PA_{c\omega}$) of $\Gamma \vdash B$ from $\Gamma \vdash A$.*
$\Leftarrow$*: Suppose that there is a proof in $PA_{c\omega}$ of $\Gamma \vdash B$ from $\Gamma \vdash A$. By using the same rules, there must be a proof of $\Gamma, A \vdash B$ from $\Gamma, A \vdash A$. However, $\Gamma, A \vdash A$ is an axiom. Hence, there is a proof in $PA_{c\omega}$ of $\Gamma, A \vdash B$.* □

5.2 Induction Within $PA_{c\omega}$

In this subsection the nature of induction within $PA_{c\omega}$ will be investigated. A distinction shall be drawn between induction as a derived rule of inference and the rule of proof which shall be called meta-induction.

Induction as a Derived Rule. Induction is a derived rule in the sense that any use of induction can be captured by use of the ω-rule. The principle of induction is a consequence of the ω-rule, by the following argument: given $\vdash A(0)$ and $\vdash \forall x\, (A(x) \rightarrow A(s(x)))$, then by n-fold application of $\forall-$ and $\rightarrow -$ elimination, we may deduce, for each n, a proof of $\vdash A(\underline{n})$. Hence, by the ω-rule, $\vdash \forall x A(x)$. However, this justification makes essential use of induction (via the cut-rule) for the step concluding $\vdash A(s(\underline{n}))$ from $\vdash A(\underline{n})$ and $A(\underline{n}) \vdash A(s(\underline{n}))$: for in order to prove each $A(\underline{k})$, cut is used to obtain access to $\forall x(A(x) \rightarrow A(s(x)))$ in each branch. See Figure 5 for a graphical representation of this argument. Although this is an inessential use of cut in the sense that the proof could be formulated differently (i.e. without a cut, by the cut elimination theorem), this demonstrates that even without the explicit use of induction, the latter is being used implicitly because the proof is being specified in a primitive recursive way. The transformation of Figure 5 into a proof not involving the cut rule is carried out in an analogous manner to the cut elimination proof for $PA_{c\omega}$; the cases will differ according to the connectives in A, but cut elimination may be provided for the part of the proof from $A(s(\underline{k}))$ to the open leaf $A(\underline{k})$; thus, the proof will be of the form given in Figure 4.

Meta-induction Over Proofs. Ordinary induction over the natural numbers can be mimicked at the meta-level in $PA_{c\omega}$: the idea, as shown in Figure 6, is that meta-induction enables proof transformations of the form "proof of $A(\underline{n})$ justifies proof of $A(s(\underline{n}))$" to result in the conclusion that $\forall x A(x)$ holds. The n achieves the status of a meta-variable.

Definition 10 Meta-induction in $PA_{c\omega}$. $\forall x A(x)$ is provable in $PA_{c\omega}$ if $A(0)$ is provable in $PA_{c\omega}$ and, for each x, the proof of $A(x)$ justifies (via its extension or transformation) the proof of $A(s(x))$.

[9] If $\Gamma, B \vdash_K A$, then $\Gamma \vdash_K B \rightarrow A$, for some logical system K [10, P127].

$$\dfrac{\Gamma \vdash A(0) \quad \dfrac{\dfrac{\dfrac{}{\Gamma,\ A(s(\underline{k})) \vdash A(s(\underline{k}))}\ axiom \qquad \Gamma \vdash A(\underline{k})\ \text{similarly}}{\Gamma,\ A(\underline{k}) \to A(s(\underline{k})) \vdash A(s(\underline{k}))}\ \to l}{\Gamma,\ \forall x(A(x) \to A(s(x))) \vdash A(s(\underline{k}))}\ \forall l}{\vdash \forall x(A(x) \to A(s(x)))}\ cut \qquad \ldots}{\Gamma \vdash A(0) \ldots \qquad \Gamma \vdash A(s(\underline{k})) \qquad \ldots}\ \omega\ rule$$

$$\Gamma \vdash \forall x A(x)$$

Fig. 5. Derivation of Induction in $PA_{c\omega}$: *Given* $\Gamma \vdash_{PA_{c\omega}} A(0)$ *and* $\vdash_{PA_{c\omega}} \forall x(A(x) \to A(s(x)))$, *it is possible to prove with the use of the cut rule that* $\Gamma \vdash_{PA_{c\omega}} \forall x A(x)$

Induction over verification proofs has already been introduced in Section 3. This form of meta-induction carries out induction over the natural numbers in the theory of prooftrees and syntax which is described in [4], and this form of induction is not the same as the derived rule of induction discussed in the previous subsection. Using the Gödel encoding method however, these inductions merge, in an analogous manner to primitive recursion and effectiveness (cf. [4]).

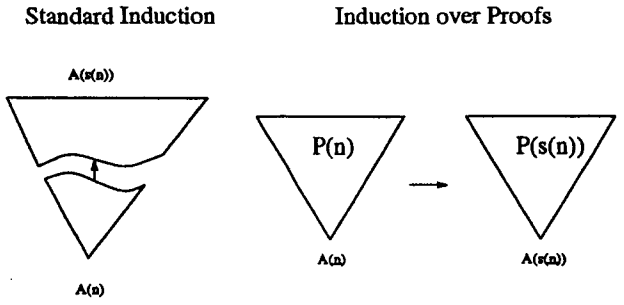

Fig. 6. Comparison of Standard Induction and Induction over Proofs

Meta-induction over ω-proofs is a more powerful form of induction than induction in PA, in the sense that there are cases (cf. Table 1) in which induction in PA is blocked (and therefore a cut is required), but meta-induction in $PA_{c\omega}$ works, making use of proof structure. Therefore, a cut in PA may be avoided by carrying out meta-induction over ω-proofs. Proofs are manipulated and not just the hypothesis, as is the case with standard induction. Not only may uniform manipulation be carried out on the nth case proof to give the $n+1$th case proof but the structure of the formal numerals themselves may be exploited to give a proof. This manipulation is not restricted to extension, as may be seen from the example of the ω-proofs: if $B_i = R_j^{f(k)}$ from Definition 8, an addition can be made to the *middle* of the proof. Meta-induction in the system $PA_{c\omega}$ can carry through when induction in PA does not, precisely because the transformation is not restricted to extension of the previous proof. This transformation is the ω-analogue of induction: manipulation of ω-proofs via meta-induction corresponds to the linearisation process, in which the same rule blocks are repeated to reduce $A(s(r))$ to $A(r)$ in the ω-proof, down to $A(0)$, and justifying $\forall x A(x)$.

To illustrate — for the example $\forall x\ (x + x) + x = x + (x + x)$, meta-induction gives the goal:

$$\underbrace{(\underline{n} + \underline{n}) + \underline{n} = \underline{n} + (\underline{n} + \underline{n})}_{A} \vdash_{PA_{c\omega}} \underbrace{(s(\underline{n}) + s(\underline{n})) + \overbrace{s(\underline{n})}^{C} = s(\underline{n}) + (s(\underline{n}) + s(\underline{n}))}_{B}.$$

A may not be substituted directly in a proof of B, because terms like C (essentially unaltered by the rules given) must be distinct from the other terms in the formula, in the sense that they must have a distinct variable. This is because one would not wish to carry out induction on these "unaltered terms"

as one would wish them to remain the same. In this case this is not so, and hence standard induction does not work. (Note the correspondence between C above and λ from Figure 2.) However, rules for proving $\mathcal{A}$ may be converted to those for proving $\mathcal{B}$ by the "meta-rule" $rule_i(n) \Rightarrow rule_i(s(n))$. Any uniform manipulation on the nth case proof (i.e. proof of $\mathcal{A}$), resulting in the $n+1$th case proof (i.e. proof of $\mathcal{B}$) will achieve the desired effect. As a result $\mathcal{B}$ may be proved without substituting $\mathcal{A}$ directly.

In conclusion, in cases of this type, induction at the meta-level may be used when induction in PA fails. (For those examples where induction in PA carries through, then a reduction of $P(s(x))$ to $P(x)$ will be possible, and by this same process, induction at the meta-level will carry through as well). Thus, the existence of the logic $PA_{c\omega}$ is justified in theorem-proving terms, for by this means a proof may be given of those propositions only provable in PA using the cut rule (the latter being considered undesirable as regards the automation of proofs), and for which induction is blocked.

5.3 Linearisation, Induction and ω-Proofs

In this subsection the relationship between linearisation of ω-proofs and induction is considered in more detail. First, an important theorem may be derived:

Theorem 11 Linearisation Theorem. *Q can be proved using a single induction if and only if there is a linear ω-proof of Q*

Proof. $\Rightarrow$: *By Theorem 9.*

$\Leftarrow$: *Given an ω-proof of the form $Q(s(k))$ reducing via rewrite steps $i(k)$ to $Q(k)$, and with such a repeated structure finally to $Q(0)$ (and equality via rewrite steps j), then $Q(s(\underline{k}))$ reduces to $Q(\underline{k})$ for all numerals such that $k \leq n$, where n was arbitrary — hence k is arbitrary. The rules $i(k)$ here form a repeated rule-block (parametrised over k): this is what has been described above as a linearisable ω-proof. By the soundness of the ω-proof representation with respect to $PA_{c\omega}$ (cf. [2]), there is a proof in $PA_{c\omega}$ of $\vdash Q(\underline{k})$ assuming $\vdash Q(s(\underline{k}))$, and by Theorem 9 there is a proof of $Q(k) \vdash Q(s(k))$, for each numeral k. By the form of the original linear proof, each proof segment $P(k)$ is obtained by taking the same shape of proof with a numerical parameter that is replaced by the appropriate value of k. So, there is a uniform correspondence between how the different instances of the numeral k are treated in each proof of $Q(k) \vdash Q(s(k))$, which demonstrates that the free variable version $Q(r) \vdash Q(s(r))$ is provable. The final lines of the ω-proof provide a proof of $Q(0)$, and therefore the prooftree in PA using induction may be completed.* □

Inductive structure at the meta-level of an ω-proof corresponds to the "linearised" form of the ω-proof: thus, an ω-proof may be "linearised" even though a direct induction proof on the original goal would be blocked.

EBG and linearisation. Theorem 11 shows that if the EBG method linearises the ω-proof, then it also suggests a suitable cut formula. It remains to be shown that the EBG method does linearise the ω-proof. Recall that the ω-proof of the generalisation is the EBG proof with the final (instantiated) result. By doing EBG, the parts of a proof which should not be altered when using induction are renamed. This puts the ω-proof into a linear form, in that (for initial line $P(n)$) $P(n)$ reduces to $P(n-1)$, whereas it did not before. More specifically, if generalisation apart (involving differentiation between the same variable) is a suitable method of generalisation, the initial theorem ($P(n)$) must be of the form $Q(\mathbf{n})$ where $\mathbf{n}$ is a sequence of n's and each occurrence of n may be generalised differently (although there might be additional free variables in Q which are not being represented in this notation). Q may have an arbitrary number of argument positions, but for clarity let us consider the simple example when there are three, and when $Q(n_1, n_2, n_3)$ reduces in the ω-proof to $Q(n_1 - 1, n_2, n_3)$. The second and third arguments will renamed by the EGB process, thus giving a linear form. A corresponding argument applies given different argument positions of the altered variable, or varied arities of Q. Thus, EBG is a way of linearising the ω-proof, and if a solution is possible by generalisation of variables apart, then this method will find it (so long as a correct ω-proof is given initially). In conclusion, when a cut formula is suggested by the EBG method proposed, it will have been shown via linearisation of the ω-proof that induction may be

successfully carried out upon it. Hence, it is not necessary to actually carry out the induction to know that the proof may be completed by using the cut rule and induction.

In conclusion, this section highlights the correspondence between linearisation and meta-induction over ω-proofs: linearity corresponds to induction over proofs. The latter can be much more powerful than standard induction in PA. Moreover, it has been shown that linearisation results in the suggestion of a correct cut formula (in PA).

6 The Linearisation Method of Generalisation

This section provides details of how the transformation of the ω-proof into a linear form is made. As discussed above, if the proof is linear induction may be carried out directly. If the proof is not obviously linear there are two cases:

- The explanation-based generalisation method may be used to generate a linear ω-proof.
- In some cases (eg. $\forall l\ len(rev(l)) = len(l)$), such a generalisation will be the same, and still not provable by induction, since what is needed is the addition of a new structure. However, this additional structure is apparent in the ω-proof (for example being explicit in the rth line of the proof), and by means of linearising the ω-proof by looking for a repeated structure while allowing some generalising, a correct cut formula may again be suggested. The procedure is to put the ω-proof into a form such that there is a repeated rule block i: thus the rth line after r uses of i is $Q(r)$, such that $Q(r)$ reduces to $Q(r-1)$. $[x_{s^r(0)}, \ldots, x_{s^{n-1}(0)}]$ is replaced by a in $Q(r)$, and rearrangements made as appropriate, to suggest a generalisation for which induction will carry through.

First though we consider how list examples and other datatypes may be represented.

6.1 Generalisation over Canonical Representations

What form would the ω-rule take for domains other than arithmetic? For the domain of arithmetic, numerals are used in the numerator: a numeral is a term which canonically represents an integer, cf. [12, P47]. For an arbitrary datatype D, so long as a canonical representation $\mathcal{K}(n)$ may be given of D, then the following ω-rule may be applied in a directly analogous way (cf. Figure 1):

$$\frac{A(\mathcal{K}(0))\ A(\mathcal{K}(1)) \ldots\ A(\mathcal{K}(n)) \ldots}{\forall x \in \mathcal{D}\ A(x)}\ \omega\ rule$$

However, for more complicated datatypes the metatheory extends the datatype, whereas this is not the case for arithmetic and lists represented using natural numbers.

Just as the representation of a natural number n may be seen so be $s^n(0)$, the general representation of a list $\mathcal{L}$ may be seen to be $[x_1, x_2, x_3, \ldots, x_m]$, where m is the length of the list, and the x_i are terms of a specified type eg. the natural numbers. These general representations may be deduced from the definitional equations of the system.

For datatypes for which the given structure and function definitions are analogous, ω-proofs will have an analogous structure. One such relevant factor is that the structural definitions should release structure in a similar manner: for example, the numerals could instead be generated by defining equations $\underline{0} = 0$ and $\underline{n+1} = s.\underline{n}$. This corresponds to lazy evaluation, as opposed to the strict evaluation used in Figure 1, and is analogous to the lazy list definitions $\mathcal{L}(0) = [\,]$ and $\mathcal{L}(s(n)) = x_{s^n(0)} :: \mathcal{L}(n)$, where $\mathcal{L}$ is a polymorphic function. The strict definitions tend to be more useful for arithmetic, because arithmetical functions often require the whole structure of a numeral, but list operations nearly always divide the list into head and tail, and hence the lazy definitions provide a more concise expression and may save unnecessary evaluation.[10] For example, the ω-proof of $\forall l\ (l <> l) <> l = l <> (l <> l)$ (where $<>$ denotes list concatenation), given the following rewrite rules:

$$nil <> X \Rightarrow X \quad (1)$$

$$(H :: T) <> X \Rightarrow H :: (T <> X) \quad (2)$$

[10] Such evaluation choices will of course affect the order of the rules used in the ω-proof. In general, one may apply $\underbrace{R_1 R_2 R_3}_{block\ one} \ldots \underbrace{R_1 R_2 R_3}_{block\ N}$, or, equivalently, $\underbrace{R_1 \ldots R_1}_{N\ times} \underbrace{R_2 \ldots R_2}_{N\ times} \underbrace{R_3 \ldots R_3}_{N\ times}$.

compares with the ω-proof given in Figure 2: just as $s^n(0)$ remains unchanged in the 2nd, 3rd, 5th and 6th places, $\mathcal{L}(n)$ remains unchanged in these positions too. The explanation-based generalisation procedure can be applied in an analogous nammer to Figure 3 to suggest an appropriate generalisation of $\forall l\ \forall m\ \forall n\ (l <> m) <> n = l <> (m <> n)$. Moreover, since lazy evaluation is used, the appropriate linearisation procedure on the ω-proof is made more apparent: the generalisation of $Q(n) \equiv (\mathcal{L}(n) <> \mathcal{L}(m)) <> \mathcal{L}(m) = \mathcal{L}(n) <> (\mathcal{L}(m) <> \mathcal{L}(m))$ may be read directly from the ω-proof: from the fact that $Q(r)$ reduces to $Q(r-1)$ in the ω-proof for $r \leq n$, and also that $Q(0)$ is provable, the general proof is in a linear form, and hence Q will be a suitable candidate for induction.

6.2 Generalisation by Adding Accumulators

Within the recursive definition of a function, an accumulator is an argument used to accumulate the value of the function. The following examples demonstrate the need for introduction of an accumulator in order to generalise.

$\forall l\ \mathit{rotate}(\mathit{len}(l), l) = l$. In order to prove $\forall l\ \mathit{rotate}(\mathit{len}(l), l) = l$ using the following rewrite rules:

$$\mathit{rotate}(0, L) \Rightarrow L \tag{3}$$
$$\mathit{rotate}(s(n), H :: T) \Rightarrow \mathit{rotate}(n, T <> (H :: \mathit{nil})) \tag{4}$$
$$\mathit{len}(\mathit{nil}) \Rightarrow 0 \tag{5}$$
$$\mathit{len}(H :: T) \Rightarrow s(\mathit{len}(T)) \tag{6}$$

the ω-proof is as follows:

$$\begin{array}{rl}
\vdash \mathit{rotate}(s(\mathit{len}(\mathcal{L}(n-1))), \mathcal{L}(n)) = \mathcal{L}(n) & \\
\vdash \mathit{rotate}(\mathit{len}(\mathcal{L}(n-1)), x_{s^{n-1}(0)} :: \mathcal{L}(n-1)) = \mathcal{L}(n) & \textit{by len, } \mathcal{L} \textit{ defs} \\
\vdash \mathit{rotate}(\mathit{len}(\mathcal{L}(n-1)), \mathcal{L}(n-1) <> [x_{s^{n-1}(0)}]) = x_{s^{n-1}(0)} :: \mathcal{L}(n-1) & \textit{by (4)} \\
\vdash \mathit{rotate}(\mathit{len}(\mathcal{L}(n-1)), (x_{s^{n-2}(0)} :: \mathcal{L}(n-2)) <> [x_{s^{n-1}(0)}]) = x_{s^{n-1}(0)} :: \mathcal{L}(n-1) & \textit{by } \mathcal{L} \textit{ def} \\
\vdash \mathit{rotate}(s(\mathit{len}(\mathcal{L}(n-2))), x_{s^{n-2}(0)} :: (\mathcal{L}(n-2) <> [x_{s^{n-1}(0)}]) = x_{s^{n-1}(0)} :: \mathcal{L}(n-1) & \textit{by (2)} \\
\vdash \mathit{rotate}(\mathit{len}(\mathcal{L}(n-2)), (\mathcal{L}(n-2) <> [x_{s^{n-1}(0)}] <> [x_{s^{n-2}(0)}]) = x_{s^{n-1}(0)} :: x_{s^{n-2}(0)} :: \mathcal{L}(n-2) & \textit{by (4), } \mathcal{L} \textit{ def} \\
\vdots \quad \textit{similarly, } n \textit{ times} & \\
\vdash \mathit{rotate}(\mathit{len}(\mathcal{L}(0)), (\mathcal{L}(0) <> [x_{s^{n-1}(0)}] <> \ldots <> [x_0]) = [x_{s^{n-1}(0)} :: \ldots :: x_0] & \\
\vdash [x_{s^{n-1}(0)}] <> \ldots <> [x_0] = [x_{s^{n-1}(0)} \cdots x_0] &
\end{array}$$

For the mth case of the above proof, such that $m < n$, the definition of Q as follows captures the linear form of the ω-proof:
$Q(m) \equiv \mathit{rotate}(\mathit{len}(\mathcal{L}(m)), (\ldots(\mathcal{L}(m) <> x_{s^n(0)}) <> \ldots) <> [x_{s^m(0)}]) = x_{s^n(0)} :: \ldots :: x_{s^m(0)} :: \mathit{nil} <> \mathcal{L}(m)$
i.e. $Q(m) \equiv \mathit{rotate}(\mathit{len}(\mathcal{L}(m)), \mathcal{L}(m) <> (x_{s^n(0)} :: \ldots :: x_{s^m(0)} :: \mathit{nil})) = x_{s^n(0)} :: \ldots :: x_{s^m(0)} :: \mathcal{L}(m)$
Let $a_m = x_{s^n(0)} :: \ldots :: x_{s^m(0)} :: \mathit{nil}$. Then $\mathit{rotate}(\mathit{len}(\mathcal{L}(m)), \mathcal{L}(m) <> a_m) = a_m <> \mathcal{L}(m)$ captures the linearisation process, the right hand side of which is produced by setting $F(x_{s^n(0)} :: \ldots :: x_{s^m(0)} :: \mathit{nil}, \mathcal{L}(m)) = x_{s^n(0)} :: \ldots :: x_{s^m(0)} :: \mathcal{L}(m)$ and solving for F. Thus the generalisation suggested is $\forall l\ \mathit{rotate}(\mathit{len}(l), l <> a) = a <> l$.

The Final Stage of Obtaining the Cut Formula. Note that a_m on the left hand side of $Q(m)$ caused the goal of writing the right hand side as some $F(a_m)$. However, the step of getting from $x :: \mathit{list}$ to $(x :: \mathit{nil}) <> \mathit{list}$ is a key step which is not obvious. Nevertheless, it can be seen from the proof that what is needed is to rewrite $x :: \mathit{list}$ into some form involving $x :: \mathit{nil}$, in order to rewrite the same terms on either side. The provision of any expression which was equivalent to $x :: \mathit{list}$ involving $x :: \mathit{nil}$ would suffice, and this stage can be carried out mechanically, by using various stored rewrite rules, together with pattern matching.

$\forall l\ \mathit{rev2}(l, \mathit{nil}) = \mathit{rev}(l)$. A similar approach works when generalising terms with a constant in an accumulator position, eg. $\forall l\ \mathit{rev2}(l, \mathit{nil}) = \mathit{rev}(l)$, where $\mathit{rev2}(\mathit{nil}, l) = l$; $\mathit{rev2}(h :: t, l) = \mathit{rev2}(t, h :: l)$. It is easily shown that induction is blocked for the original formula but not for the suggested cut formula. Defining $Q(k) = \mathit{rev2}(\mathcal{L}(k), a_k) = \mathit{rev}(\mathcal{L}(k)) <> a_k$, where a_k is $x_{s^r(0)} :: x_{s^{r-1}(0)} :: \ldots :: \mathit{nil}$, gives a linear

ω-proof from $Q(n)$ to $Q(0)$. Although a_k is dependent on k, this does not matter, since the generalisation suggested is universally quantified over a. The generalisation suggested is $\forall l\ \forall a\ rev2(l, a) = rev(l) <> a$, which is a correct generalisation, and this method has enabled the generalisation of *nil* (a constant) to a variable.

Use of the Associativity of Append. The associativity of append is needed for all of the generalised list proofs. This is because there is a definitional position in all of the examples in which the appropriate rewrite rule cannot be used until the structure of the term in this position has been broken down, or else it is necessary to use the definition of append n times in the final rule block (cf. $P(0)$). If a rule is used n times in this block, that means that the structure cannot be distributed earlier in the proof as normal, in other words into each $P(n)$, one rule each time. So if this rule is used n times in the final rule block, it means that it is necessary to use an associativity rewrite rule whenever the generalisation stage is carried out. In other words, if an additional lemma is used in a ω-proof, there will have to be some forms of rewriting at the end of the corresponding linearised ω-proof in which such lemmata will also be needed.

6.3 Partial Linearisation

For examples where it is not obvious how to apply the linearisation method, it may still be appropriate to examine the ω-proof to see if some new structure has emerged.

For the example $\forall x\ even(x + x)$, given the axioms and ω-proof as follows:

$$even(0) = true \tag{7}$$

$$even(s(0)) = false \tag{8}$$

$$\forall n\ even(s(s(n))) = even(n) \tag{9}$$

ω-Proof:

$$\begin{array}{ll} EXPAND\ n \rightarrow s^n(0) & \vdash even(n + n) = true \\ USE\ (A2)\ n\ times & \vdash even(s^n(0) + s^n(0)) = true \\ USE\ (A1) & \vdash even(s^n(0 + s^n(0))) = true \\ USE\ (9)\ n\ times & \vdash even(s^n(s^n(0))) = true^* \\ USE\ (7) & \vdash even(0) = true^{**} \end{array}$$

Obvious choices for the cut formula here would result in circularity or a non-theorem, but the ω-proof gives some clues as to what is happening, and does not suggest an incorrect cut formula. Indeed, a cut formula which is suggested is that of $\forall y\ even(2.y)$, because of the way a successor function from each of the original variables pairs off together in the ω-proof. Notice that the sub-part of the ω-proof from $*$ to $**$ is linear: $even(s^n(s^n(0)))$ reduces to $even(s^{n-1}(s^{n-1}(0)))$, and similarly by (9) eventually to $even(0)$.

This linear structure from $even(s^n(s^n(0)))$ suggests that this is a suitable candidate for generalisation. What is happening in the ω-proof is that $even(s^k(s^k(0)))$ is being proved at the meta-level (the linearisation process corresponding to induction on k, as shown in Section 5). The generalisation suggested will therefore be $\forall k\ even(z(k))$, where $\underline{z}(k) = s^k(s^k(0))$, i.e. $z(k) = 2.k$. Thus, a linear subproof of the original ω-proof may suggest a cut formula, even though the ω-proof itself cannot easily be linearised.

Hence, if ω-proofs do not have an obviously linear form, the method is still found useful in suggesting a cut formula within the framework of a co-operative environment.

6.4 Summary and Conclusions

The linearisation method has been suggested for the provision of a cut formula via the ω-proof; using this method, in many differing types of cases a correct cut formula may be found. The final stages of generalisation, with rearranging to achieve equality, were achieved by guessing (since there are only a small number of possibilities, and pre-stored information could be used, since the same or similar justifications arise each time), and then checking this result. There are only a small number of possibilities, and these may be found, and checked, mechanically. Automation of linearisation is possible, since there is literal repetition of rules (with parameters denoting where and the number of times they are applied), and

from these rule blocks may be formed, if necessary by rearranging the rules. The rth case provides the generalisation, by means of the collection and rearrangement of terms, as described above.

In conclusion, a general and uniform approach to generalisation is that of linearising the ω-proof. If accumulators are needed, or in some complicated cases, the linearisation approach suggested is able to provide a suitable cut formula (cf. Table 2). Moreover, when a cut formula is suggested, it will have been shown via linearisation of the ω-proof that induction may be successfully carried out upon it. Hence, it is not necessary to actually carry out the induction to know that the proof may be completed by using the cut rule and induction.

7 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, namely Oyster, which is a reimplementation of NuPRL [8]. This embodies a higher-order, typed constructive logic in sequent-calculus form. Within the Oyster framework, the object-level logic is replaced with Peano and Heyting arithmetic and the rules that can be applied are those of the sequent calculus axiomatisation of first order logic given in [10, P133], together with mathematical induction. The search for a proof must be guided either by a human user or by a proof tactic. Each proof is built up in the form of a tree, and every stage of the tree may be displayed on the screen with information as to the hypotheses, goals, position in the tree and whether the subtree is proved below it.

Theorem	Cut Formula
$\forall x\ (x+x)+x = x+(x+x)$	$\forall x \forall y \forall z\ (x+y)+z = x+(y+z)$
$\forall x\ x+s(x) = s(x)+x$	$\forall x \forall y\ x+s(y) = s(x)+y$
$\forall x\ x.(x+x) = x.x+x.x$	$\forall x \forall y \forall z\ x.(y+z) = x.y+x.z$
$\forall x\ (x+x).x = x.x+x.x$	$\forall x \forall y \forall z\ (x+y).z = y.z+x.z$
$\forall x\ x+x = 0 \rightarrow x = 0$	$\forall x \forall y\ x+y = 0 \rightarrow x = 0$
$\forall x\ x \neq 0 \rightarrow p(x)+s(s(x)) = s(x)+x$	$\forall x \forall y\ x \neq 0 \rightarrow p(x)+s(s(y)) = s(x)+y$
$\forall x\ even(x+x)$	$\forall x\ even(2.x)$
$\forall x\ even(x) \rightarrow even(x.x)$	$\forall x \forall y\ even(x) \rightarrow even(x.y)$
$\forall x\ half\_int(x+x) = x$	$\forall x\ half\_int(2.x) = x$
$\forall l\ len(rev(l)) = len(l)$	$\forall l\ len(rev(l) <> a) = len(rev(a) <> l)$
$\forall l\ rotate(len(l), l) = l$	$\forall l\ rotate(len(l), l <> a) = a <> l$
$\forall l\ rev(rev(l) <> y :: nil) = y :: rev(rev(l))$	$\forall l\ rev(a <> y :: nil) = y :: rev(a)$
$\forall l\ rev2(l, nil) = rev(l)$	$\forall l\ rev2(l, a) = rev(l) <> a$
$\forall l\ (l <> l) <> l = l <> (l <> l)$	$\forall l \forall p \forall q\ (l <> p) <> q = l <> (p <> q)$
$\forall x(\forall y(y \leq x \rightarrow P(y)) \rightarrow P(x)) \vdash P(z)$	$\forall x(\forall w(w \leq x \rightarrow P(w)) \rightarrow P(x)) \vdash \forall y(y \leq z \rightarrow P(y))$

Table 1. Cut Formulae Suggested by Guiding Method for Various Examples

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 initial generalisation and then works by updating this ω-proof using the other individual proofs, until the ω-proof seems to have reached a stable form. This ω-proof is then automatically checked to see if it is indeed the correct one, as described above. 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 ω-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 proof-tree. 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 ω-proof), with a further option to complete the tree as far as possible (using standard theorem-proving techniques). The ω-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 ω-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 ω-proof, and is able to display individual proofs in addition to the general case. [3] provides details of the interactive system, and how to use it.

8 Comparison with Related Work

The methods of generalisation proposed in this paper involving manipulation of the ω-proof should be compared with current generalisation methods. Of these, perhaps the most famous is that implemented by Boyer and Moore in their theorem-prover NQTHM [6]. The main heuristic for (goal) 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 [1]. 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. Castaing has proposed an alternative approach that proceeds by unfolding a function definition [9]. The induction variable is chosen, all function definitions are unfolded once and then a new variable introduced for each constant and variable, excluding those argument positions that correspond to the position of the induction variable. In general, the result of this step will be an invalid lemma, but the relationships between the new variables may be examined by means of appropriate instantiations of the induction variable. However, no suggestion for how the induction variable could be identified is given, and in general the problem of obtaining and solving the equations is a difficult one, making automation of this method very difficult. When this method fails, Castaing suggests instead that the definitions be unfolded once, cross-fertilisation used[11] to produce an equation of one function (the choice of function to be left to the user), and this equation generalised as in the previous method. This method normally requires two inductive proofs once the generalisation has been obtained. These methods are used as the basis for generalisation in many different theorem-proving systems. However, Hesketh's approach of directing generalisation by the failure of heuristics involved in proof search has proved more successful [16], although very often several alternative solutions have to be tested rather than a single solution being given in a uniform manner. Related work in proof generalisation has been carried out by Masami Hagiya, who has considered generalization of proofs in type theories. Hagiya approaches the problem of proof generalisation by synthesising a general proof from a concrete example proof by higher-order unification in a type theory with a recursion operator [15]. Rather than the ω-rule, he uses ordinary recursion terms for representing inductive proofs: in order to make recursion terms more expressive, he has extended the calculus with implicit arithmetical inferences [14]. However, Hagiya has not carried out any implementation, nor addressed the issue of ensuring that the proofs from which one generalises are in the form required to produce a suitable result. Therefore, no generalisation tactic is produced which automatically suggests cut formulae upon input of theorems.

Another related topic concerns the utility of reusing a proof in inductive theorem proving by means of generalising the proof and checking whether this generalised proof could provide a method for proving other (specific) proofs. Kolbe and Walther have developed this approach [17] and the same motivation of reusing proofs could also be applied with reference to the approach proposed in this paper.

[11] using an equality in the hypothesis to remove a term from a goal

The linearisation method proposed in this paper provides a uniform approach and does not have to check extra criteria, nor work through individual examples. Moreover, it is not possible to over-generalise to a non-theorem (the method is sound but not complete — it does not always provide a solution, nor necessarily the best solution possible). Table 2 provides a comparison between cut formulae suggested by the various methods, and demonstrates that the linearisation (and more specifically, the EBG guiding) method works well on a range of arithmetical examples. In this way, correct cut formulae are suggested for many difficult examples [2]: in particular, one generalisation provided by this method of $\forall al\ len(append(rev(l), a)) = len(append(rev(a), l))$ (from $len(rev(l)) = len(l)$) is a better result (since it only requires one induction) than the only alternative suggestion provided for this example of $\forall al\ len(append(rev(l), a)) = len(append(a, l))$[12] (requiring two inductions).

Methods	Selection of Types of Example	
	$x.(x+x) = x.x + x.x$	$(x+x)+x = x+(x+x)$
Boyer & Moore	*fail*	$y+x = x+y$
Castaing	*fail*	*fail*
Aubin	$x.(y+y) = x.y + x.y$ 1.	$(x+y)+y = x+(y+y)$
Hesketh	$x.(y+y) = x.y + x.y$ 2.	$(x+y)+y = x+(y+y)$
Pearson	$x.(y+z) = x.y + x.z$	$(x+y)+z = x+(y+z)$
	$x + s(x) = s(x) + x$	$even(x+x)$
Boyer & Moore	$x+y = y+x$	*fail*
Castaing	*fail*	*fail*
Aubin	*fail*	*fail*
Hesketh	$x + s(y) = s(x) + y$ 3.	*fail*
Pearson	$x + s(y) = s(x) + y$	$even(2.x)$ 4.
	$x \neq 0 \rightarrow p(x) + s(s(y)) = s(x) + y$	$rotate(len(l), l) = l$
Boyer & Moore	*fail*	*fail*
Castaing	*fail*	*fail*
Aubin	*fail*	*fail*
Hesketh	$x \neq 0 \rightarrow p(x) + s(s(y)) = s(x) + y$ 3.	$rotate(len(l), l <> a) = a <> l$
Pearson	$x \neq 0 \rightarrow p(x) + s(s(y)) = s(x) + y$	$rotate(len(l), l <> a) = a <> l$
	$rev2(l, nil) = rev(l)$	$x.(y.0) = y.(y.0)$
Boyer & Moore	*fail*	*fail*
Castaing	$rev2(l, a) = rev(l) <> a$	*fail*
Aubin	$rev2(l, a) = rev(l) <> a$	*fail*
Hesketh	$rev2(l, a) = rev(l) <> a$	*fail*
Pearson	$rev2(l, a) = rev(l) <> a$	*ω-proof only* 5.

1. Solution achieved by testing various possibilities after initial failure of method.
2. Solution achieved by pruning search space.
3. Probable result (this example was not considered in [16]).
4. Generalisation algorithm fails, but this cut formula is 'suggested' from the ω-proof.
5. Generalisation algorithm fails, but user might be able to suggest a generalisation from the ω-proof.

Table 2. Cut Formulae Suggested Using Different Generalisation Methods

By the EBG and linearisation methods, the most general generalisation of an expression is achieved, in a uniform manner. This is a new and desirable achievement: Hesketh writes [16, p.271] that for her generalisation method "The generalisation from $\forall x.x + (x+x) = (x+x) + x$ will be found as $\forall x \forall y.x + (y+y) = (x+y) + y$ not $\forall x \forall y \forall z.x + (y+z) = (x+y) + z$ but I know of no theorem prover that can find this last generalisation in a principled way, i.e. other than with just trial and error." Note also how if an ω-proof is provided, even without a generalisation being suggested, something has still been achieved, in the sense that a pattern might still emerge for the user. Thus the method may still be useful within a co-operative environment if it breaks down (cf. note 5., Table 1). This contrasts with alternative methods

[12] cf. [16].

of generalisation, which do not provide much information if they fail. Moreover, because the suggested method explicitly exploits general patterns, it has a higher-level structure and thus greater potential for extension than other more special-purpose approaches.

The EBG method proposed will succeed in the sense that there does exist some ω-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 if generalisation apart is not appropriate for the example under consideration. On the other hand, the linearisation method is applicable to a wide variety of types of generalisation. So long as the ω-proof can be linearised, then an inductive proof will have been found.

Note that it is possible to carry out explanation-based generalisation or linearisation on proofs in other systems such as Peano arithmetic, in order to give a more general expression. However, if inductive proof is not possible in these systems (because induction is blocked) and hence application of the cut rule is needed in order to obtain a proof, then such a method will not work. However, it is possible that an ω-proof may be returned, and in this case explanation-based generalisation or linearisation may be applied to the latter in order to obtain a solution.

The generalisation method proposed in this paper can be used in a more general context for lemma generation, regardless of the way an ω-proof was obtained, so long as one can represent in the particular system of interest the notions of nth-successor, parametrised applications of rewrite rules and also the correctness check. Hence, although the learning device for cut formulae is not reliant upon the given proof system involving use of the ω-rule, in practice suitable proof environments such as HOL [13] would require extension, and moreover the user would have to input the proof directly unless some other method of generation could be provided. Hence, if the ω-proof were represented directly in higher-order logic, one would still be faced with the problems of application of rewrite rules a parametrised number of times, and a method of provision of the ω-proof (solved by inductive inference in the case of $PA_{c\omega}$).

9 Conclusions

In summary, an ω-proof is constructed using some inductive inference process (the individual proofs being easily generated using basic tactics), or obtained by some other means. A uniform approach to generalisation is suggested in this paper of transforming the ω-proof into a proof of a generalisation of the original expression, which is linear in form, and hence suggests an appropriate cut formula. The approach also applies generally to other data-types. Not only is it the case that certain new structural patterns may be seen in the ω-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, $[x_1, x_2, \ldots x_m]$ 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.

The new method for generalisation which has been proposed 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. 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.

Acknowledgements. This paper is based on studies conducted at both the Mathematical Reasoning Group in Edinburgh University and the Computer Laboratory in Cambridge University.

References

1. Aubin, R.: Mechanising structural induction II: Strategies. *Theoretical Computer Science* 9 (1979) 347–361
2. Baker, S.: *Aspects of the Constructive Omega Rule within Automated Deduction.* PhD thesis, University of Edinburgh (1992)
3. Baker, S.: CORE Manual. Technical Paper 10, University of Edinburgh DAI (1992)
4. Baker, S.: A proof environment for arithmetic with the omega rule. In J. Campbell, editor, *Proceedings of AISMC-2*, Lecture Notes in Computer Science, Springer-Verlag (1994)

5. Baker, S., Ireland, A., Smaill, A.: 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, Springer-Verlag **624** (1992) 214–225
6. Boyer, R.S., Moore, J.S.: *A Computational Logic* Academic Press (1979)
7. Bundy, A., Stevens, A., van Harmelen, F., Ireland, A., Smaill, A.: Rippling: A heuristic for guiding inductive proofs. *Artificial Intelligence* **62** (1993) 185–253
8. Bundy, A., van Harmelen, F., Horn, C., Smaill, A.: The Oyster-Clam system. In M.E. Stickel, editor, *CADE-10*, Lecture Notes in Artificial Intelligence, Springer-Verlag **449** (1990) 647–648
9. Castaing, J.: How to facilitate the proof of theorems by using induction-matching, and by generalisation. In A. Joshi, editor, *Proceedings of the ninth IJCAI* (1985) 1208–1213
10. Dummett, M.: *Elements of Intuitionism.* Oxford Logic Guides. Oxford Univ. Press, Oxford (1977)
11. Feferman, S.: Transfinite recursive progressions of axiomatic theories. Journal of Symbolic Logic **27** (1962) 259–316
12. Girard, J.-Y.: *Proof Theory and Logical Complexity* Bibliopolis **1** (1987)
13. Gordon, M: HOL: A proof generating system for higher-order logic. In G. Birtwistle and P.A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis* Kluwer (1988)
14. Hagiya, M.: A Typed λ-Calculus for Proving-by-Example and Bottom-Up Generalisation Procedure. *Algorithmic Learning Theory 93* Lecture Notes in Artificial Intelligence **744** (1993)
15. Hagiya, M.: Programming by example and proving by example using higher-order unification. *10th Conference on Automated Deduction* Lecture Notes in Artificial Intelligence **448** (1990) 588–602
16. Hesketh, J.T.: *Using Middle-Out Reasoning to Guide Inductive Theorem Proving.* PhD thesis, University of Edinburgh (1991)
17. Kolbe, T., Walther, C.: Reusing proofs. In A. Cohn, editor, *11th European Conference on Artificial Intelligence*, John Wiley & Sons Ltd (1994)
18. Kreisel, G.: On the Interpretation of Non-Finitist Proofs. Journal of Symbolic Logic **17** (1952) 43–58
19. Kreisel, G.: Mathematical logic. In T.L. Saaty, editor, *Lectures on Modern Mathematics* John Wiley & Sons **III** (1965) 95–195
20. Kreisel, G.: Proof theory: Some personal recollections. In G. Takeuti, editor, *Proof Theory* North Holland (1987) 395–405
21. Lõpez-Escobar, E.G.K.: On an extremely restricted ω-rule. *Fundamenta Mathematicae* **90** (1976) 159–72
22. Madden, P.: The specialization and transformation of constructive existence proofs. In N.S. Sridharan, editor, *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, Morgan Kaufmann (1989)
23. Mitchell, T.M.: Toward combining empirical and analytical methods for inferring heuristics. Technical Report LCSR-TR-27, Laboratory for Computer Science Research, Rutgers University (1982)
24. Plotkin, G.: A note on inductive generalization. In D. Michie and B. Meltzer, editors, *Machine Intelligence* **5** Edinburgh University Press (1969) 153–164
25. Prawitz, D.: 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* North Holland **63** (1971) 235–307
26. Rosser, B.: Gödel-theorems for non-constructive logics. JSL **2** (1937) 129–137
27. Rouveirol, C.: Saturation: Postponing choices when inverting resolution. In *Proceedings of ECAI-90* (1990) 557–562
28. Schütte, K.: *Proof Theory.* Springer-Verlag (1977)
29. Schwichtenberg, H.: Proof theory: Some applications of cut-elimination. In J. Barwise, editor, *Handbook of Mathematical Logic* North Holland (1977) 867–896
30. Shoenfield, J.R.: On a restricted ω-rule. Bull. Acad. Sc. Polon. Sci., Ser. des sc. math., astr. et phys. **7** (1959) 405–7
31. Takeuti, G.: *Proof theory.* North-Holland, 2 edition (1987)
32. Tucker, J.V., Wainer, S.S., Zucker, J.I.: Provable computable functions on abstract-data-types. In M.S. Paterson, editor, *Automata, Languages and Programming* Lecture Notes in Computer Science, Springer-Verlag **443** (1990) 660–673
33. Van der Waerden, B.L.: How the proof of Baudet's conjecture was found. In L. Mirsky, editor, *Papers presented to Richard Rado on the occasion of his sixty-fifth birthday*, Academic Press, London-New York (1971) 252–260
34. Yoccoz, S.:. Constructive aspects of the omega-rule: Application to proof systems in computer science and algorithmic logic. Lecture Notes in Computer Science **379** (1989) 553–565

This article was processed using the LaTeX macro package with LLNCS style