Proofs About Lists Using Ellipsis

Alan Bundy and Julian Richardson*

Institute for Representation and Reasoning
University of Edinburgh
80 South Bridge, Edinburgh EH1 1HN, Scotland.
`a.bundy@ed.ac.uk,julian.richardson@ed.ac.uk`

Abstract. In this paper we explore the use of *ellipsis* in proofs about lists. We present a higher-order formulation of elliptic formulae, and describe its implementation in the λ*Clam* proof planner. We use an unambiguous higher-order formulation of lists which is amenable to formal proofs without using induction, and to display using the familiar ... notation.

1 Introduction

A notation often used in informal mathematical proofs is ellipsis (the dots in $a_1 + ... + a_n$). Not only does the use of ellipsis make many proofs much easier to understand, but it also naturally lends itself to theories where induction has been replaced by suitable axioms.

Ellipsis can be used to abbreviate many different kinds of formulae; in this paper, we explore the use of ellipsis in proofs about lists. This allows us to address important issues in the automatic treatment of ellipsis and, while we do not extensively consider it here, can be extended by applying `fold` functions (see for example equation (2) below and §11) to reasoning about elliptic formulae in which the main connective is not list `cons`. We present a higher-order formulation of elliptic formulae, and describe its implementation in the λ*Clam* proof planner [8]. To resolve the ambiguities inherent in elliptic representation, we use an underlying unambiguous representation which is portrayed by ellipsis. We define a higher-order function $\square$ which represents a list by the length of the list and a function which takes a natural number n and returns the n^{th} member of the list.

Displaying proofs in elliptic notation poses interesting challenges. One step of a proof in the elliptic notation may require several steps in the implementation. The display mechanism can itself perform quite sophisticated rewriting in order to get a useful portrayal of a formula. The portrayal system cannot just be bolted on top of the theorem prover but must itself influence the way in which proofs are carried out; ensuring that formulae are in a form for which elliptic portrayal is effective imposes restrictions on the order in which proof steps are applied.

* The authors gratefully acknowledge the support of EPSRC grants GR/L/11724 and GR/M/45030, and the comments of their colleagues in the Mathematical Reasoning Group. We would also like to thank the referees for their insightful comments.

2 A Motivating Example

We consider two alternative definitions of a *foldl* function, one a recursive definition, the other an elliptic definition. The recursive definition is given in (1).

$$\begin{aligned} foldl(\otimes, A, []) &= A \\ foldl(\otimes, A, [H|T]) &= foldl(\otimes, A \otimes H, T) \end{aligned} \qquad (1)$$

How quickly can you spot what this function does? Compare this with an elliptic definition:

$$foldl(\otimes, A, [E_1, E_2, \ldots, E_n]) = (\ldots((A \otimes E_1) \otimes E_2) \otimes \ldots \otimes E_n) \qquad (2)$$

Do you find that easier to understand?

If you are like us, you find (2) much easier to understand than (1). In fact, one can argue that (2) is the *real* meaning of $foldl$, and (1) is merely the best way to represent this meaning in most logics. Unfortunately, (2) is not normally available because ellipsis is not usually a legal part of the syntax.

We will call formulae like (2) *schematic*, because we can think of it as a schema standing for an infinite number of formulae: one for each n. Imagine we had a logic in which schematic formulae were legal syntax and in which (2) was the *definition* of $foldl$. We will call this a *schematic logic*. Such a logic was used in [1] to represent generalised proofs. From time to time other people propose such logics, e.g. [6].

We can use definition (2) to prove the following theorem:

$$foldl(\otimes, A, [E_1, \ldots, E_{n-1}, E_n]) = foldl(\otimes, A, [E_1, \ldots, E_{n-1}]) \otimes E_n$$

This is a trivial theorem in the schematic logic. It requires just two applications of definition (2). We must only be careful to insert the condition that $1 \leq n$, so that the right hand side is meaningful. By contrast the usual inductive proof using (1) is less immediately understandable as it requires induction and choosing an appropriate instantiation for the A in the induction hypothesis.[1]

It seems that schematic definitions and proofs that use them can be easier to understand than their regular counterparts. It is often possible to avoid induction by using a generalised schematic proof, i.e. one in which ellipsis is used in the proof as well as the formulae.

There are several problems which must be solved to make this possible:

1. Ellipsis can be ambiguous. There has to be a mechanism for deciding what is elided in the $\ldots$. For example, what is meant by $[E_2, ..., E_{16}]$? Does the list have 15, 8, 4 or some other number of elements? In the preceding examples, the meaning of the ellipsis is clear, but in general it may be necessary to restrict the use of ellipsis to those cases that are unambiguous, if we can decide what those are.

[1] If A is not universally quantified in the conjecture, then an additional generalisation step is required in the inductive proof.

2. It is necessary to keep track of conditions, like $1 \leq n$ in the proof above, which are needed to ensure that schematic formulae are well formed. This can get quite hard.
3. We might want to translate the resulting schematic proof into a proof in a regular logic. Writing the tactics for this would be a challenge.

In the sections which follow, we present a representation of lists which lends itself both to formal proof and to elliptic proof and portrayal. To address (1) above, we do not consider the *input* of formulae which contain ellipsis, but aim instead merely to portray formulae using ellipsis in a predictable way which is unambiguous to the reader. We address (2) by disregarding well-formedness conditions in our initial implementation and checking manually to ensure that ill-formed formulae do not appear in the proof. The proofs we construct are proofs in a higher-order logic, so no translation is necessary to satisfy (3) above.

3 The Representation of Ellipsis

3.1 The Ambiguity of Ellipsis

The first problem in formalising ellipsis is its inherent ambiguity. The reader of a formula containing ellipsis has to induce a pattern from the expressions on either side of the dots. For instance, it is necessary to induce that $a_1 + \ldots + a_n$ means $\sum_{i=1}^{n} a_i$ and not $\sum_{i=1/2}^{n/2} a_{2.i}$, say, *i.e.* that the numbers go up in ones not twos — or threes — or in some more complicated pattern. One can try to disambiguate ellipsis by putting in more context, *e.g.* $a_1 + a_2 + \ldots + a_n$, but some ambiguity will always remain.

More importantly, it is hard to see how we can ensure that a "proof" is in fact a **proof** unless it can be expressed in an *unambiguous* internal representation.

3.2 An Unambiguous Representation

If an unambiguous internal representation is needed anyway, then why not use this instead of ellipsis? Ellipsis can be used as an external 'portray' form of this unambiguous representation. This will avoid the need for constant pattern recognition to figure out what is going on, but externally can be indistinguishable. Pattern recognition would be needed only if ellipsis is used when *inputting* formulae. This is the view we adopt here.

For n-ary sums and products we already have such an unambiguous notation in $\sum$ and $\prod$. However, we don't have such a notation for lists, sequences or other n-ary operations. The main focus of this paper is to introduce a similar notation for lists. This notation is then used for representing sequences and any other use of ellipsis. We will use the notation $\Box$ in a similar way to $\sum$ or $\prod$.

$\Box$ is a polymorphic, second order function of type:

$$\Box : (nat \rightarrow (nat \rightarrow \tau)) \rightarrow list(\tau)$$

Its first argument is the length of the list. It applies the function to each of the natural numbers 1, 2, *etc.* up to this length and returns a list of the results, *i.e.*

$$\Box(n, f) = [f(1), \dots, f(n)]$$

Note that we use function application instead of subscripts, so a subscribed term a_i is represented by a function application $a(i)$.

4 The Axiomatisation of $\Box$

$\Box$ can be defined recursively as follows (where :: and <> are infix *cons* and *append* respectively):

$$\begin{aligned} \Box(0, F) &= nil \\ \Box(s(N), F) &= \Box(N, F) <> (F(s(N)) :: nil) \end{aligned}$$

Or, alternatively, as:

$$\begin{aligned} \Box(0, F) &= nil \\ \Box(s(N), F) &= F(1) :: \Box(N, \lambda i.\ F(s(i))) \end{aligned}$$

Armed with $\Box$ we can avoid much of the need for recursion in defining new functions (*cf.* the work of Bird [2]). All we need is an axiom that says that all lists can be put in $\Box$ form, *i.e.*

$$\forall L{:}list(\tau), \exists n{:}nat, \exists f : (nat \to \tau).\ L = \Box(n, f)$$

Then we can define *len*, <> (infix *append*) and *rev* as:

$$\begin{aligned} len(\Box(N, F)) &= N \\ rev(\Box(N, F)) &= \Box(N, \lambda i.\ F(s(N) - i)) \\ \Box(M, F) <> \Box(N, G) &= \Box(M + N, comb(M, F, G)) \end{aligned}$$

where *comb* is defined by:

$$comb(M, F, G)(i) = \begin{cases} F(i) & \text{if } i \leq M \\ G(i - M) & \text{if } i > M \end{cases} \tag{3}$$

These definitions should be portrayed, in elliptic notation, as:

$$\begin{aligned} len([F(1), \dots, F(N)]) &= N \\ rev([F(1), \dots, F(N)]) &= [F(N), \dots, F(1)] \\ [F(1), \dots, F(M)] <> [G(1), \dots, G(N)] &= [F(1), \dots, F(M), G(1), \dots, G(N)] \end{aligned}$$

5 Proofs using Ellipsis

As so many of the definitions are non-recursive, the proofs can be non-inductive. In this section we present an example.

5.1 Rotate Length

Consider the classic rotate-length conjecture:

$$rot(len(L), L) = L$$

Informally, $rot(N, L)$ returns a list with the same length as the list L but with the first N elements removed from the front and appended to the end. Here is a definition of rot using ellipsis:

$$M \leq N \rightarrow rot(M, \Box(N, F)) = \Box(N - M, \lambda i \,.\, F(M + i)) <> \Box(M, F)$$

In elliptic notation, this definition translates to:

$$rot(M, [F(1), ..., F(N)]) = [F(M + 1), ..., F(N)] <> [F(1), ..., F(M)]$$

Then the $\Box$ proof is:

$$\begin{aligned} rot(len(\Box(N, F)), \Box(N, F)) &= rot(N, \Box(N, F)) \\ &= \Box(N - N, \lambda i.\ F(N + i)) <> \Box(N, F) \\ &= \Box(0, \lambda i.\ F(N + i)) <> \Box(N, F) \\ &= \Box(0 + N, comb(0, \lambda i.\ F(N + i), F)) \qquad (4) \\ &= \Box(N, F) \end{aligned}$$

or in elliptic notation:

$$\begin{aligned} & rot(len([F(1), \ldots, F(N)]), [F(1), \ldots, F(N)]) \\ &\qquad = rot(N, [F(1), \ldots, F(N)]) \\ &\qquad = [] <> [F(1), \ldots, F(N)] \\ &\qquad = [F(1), \ldots, F(N)] \end{aligned}$$

For comparison, λ*Clam* cannot prove this theorem using its standard inductive strategy. The *Clam* proof planner [4] is unable to prove this theorem without using critics [5]. Both *Clam* and λ*Clam* are able to prove the generalised theorem $rot(len(l), l <> m) = (m <> l)$.

6 Elliptic Portrayal

The key to the success of this technique is that the internal $\Box$ notation can be portrayed in an intuitively satisfying external elliptic notation. A comparison of the number of steps in the formal (four steps), versus the informal (two steps), proofs above indicates that there need not be a 1-1 correspondence between proof steps in the formal and informal proofs, and conversion between the two representations may not be entirely straightforward. Rewriting is often required to process the internal representation into a portrayable form. For example, correct portrayal of (4) above requires two rewrites: $0 + N \Rightarrow N$, and

$comb(0, \lambda i.\ F(N+i), F) \Rightarrow F$. Sometimes internal proof steps cannot be portrayed at all and must be omitted, leading to a mismatch between internal and external proof steps.

Consider, for instance, the definition of append:

$$\Box(M, F) <> \Box(N, G) = \Box(M+N, comb(M, F, G))$$

which we would like to portray as:

$$\begin{aligned}&[F(1), \ldots, F(M)] <> [G(1), \ldots, G(N)]) = \\ &\qquad\qquad [F(1), \ldots, F(M), G(1), \ldots, G(N)]\end{aligned}$$

Firstly, note that we do not want the internal function *comb* to appear at all. We want to evaluate expressions like $comb(M, F, G)(M+N)$ to $G(N)$, which requires the rewriting:

$$\begin{aligned}comb(M, F, G)(M+N) &\Rightarrow G(M+N-M)\\ &\Rightarrow G(N)\end{aligned}$$

In general, there is no limit to the amount of rewriting that might be required here. A lot of conjectures can be proved, however, by normalising arithmetic expressions when possible, and applying a few rewrite rules concerning *comb* and similar functions.

Secondly, note that which elements of the list we portray is very context sensitive. We do not always want to portray just the first and last elements, but also the elements either side of significant boundaries. In general, detecting such critical boundaries involves solving inequalities over the natural numbers modulo some domain theory. Inequality reasoning is not implemented in the current system. This limits both portrayal and proof to a small but interesting class of examples.

7 Implementation

We have implemented a system for reasoning about ellipsis in lists in the higher-order proof planner, λ*Clam* [8]. λ*Clam* provides a convenient basis for our implementation because we need to reason carefully about higher-order functions and variable scope; correct reasoning about functions and variable scope is built into λ*Clam*'s underlying meta-theory.

The implementation consists of a number of proof planning methods [3] and some code for portraying elliptic formulae.

7.1 Portrayal

The bulk of the work which is necessary during portrayal is normalisation of arithmetic expressions. For example, portraying $\Box(n-(m+(n-1))+(m+1+n), F)$ as the elliptic term $[F(1), ..., F(n-(m+(n-1))+(m+1+n))]$ is both ugly and destroys the simplicity of presentation which is the main point of the exercise. The first step in elliptic portrayal is therefore to simplify the first argument as much as possible using a procedure which normalises expressions built from positive integer constants, variables, $+$ and $-$. The above example is correctly portrayed by our implementation as $[(F\ 1), ..., (F\ (n+2))]$.

7.2 Methods

λClam was extended with a new proof planning method: `boxintro`, which converts conjectures about lists in the standard notation to conjectures about lists in the $\Box$ notation. Every universal quantifier $\forall l : list(\tau)$ is replaced by two quantifiers $\forall n : nat\ \forall f : nat \rightarrow \tau$, and the occurrences of l which are in this quantifier's scope are replaced by $\Box(n, f)$. Occurrences of nil in the conjecture are replaced by $\Box(zero, (\lambda x\, .\, x))$.

In addition, *λClam*'s symbolic evaluation (exhaustive rewriting) method has been modified to apply equations which simplify expressions involving natural numbers before other equations.[2] This is necessary in order to help the portrayal code simplify arithmetic expressions as soon as possible and thereby avoid portrayals such as $[F_1(s(len([F_1(1), ...F_1(V_0)]))), ..., F_1(V_0)]$, which was produced by an early version of the system (and in fact turned out to be $[\,]$, a fact which is only apparent after equation (length3) (see below) has been applied).

In the following sections we give some example output from the system, and discuss the issues it raises.

8 An Example: The Rotate Length Theorem

The rotate-length example of §5.1 cannot be proved by the standard version of *λClam*.[3] Using ellipsis, it is proved automatically by *λClam* using only repeated rewriting. For clarity, in the presentation below, we have removed quantifiers, written equality in infix form, and written function applications as $f(x)$ instead of $f\ x$. The elliptic parts of the presentation are however as produced by the system.

The following rewrite rules are used:

$$\begin{array}{ll}
len([F(1), ..., F(N)]) \Rightarrow N & \text{(length3)} \\
rot(M, [F(1), ..., F(N)]) \Rightarrow & \\
\qquad [F(M+1), ..., F(N)] \ <>\ [F(1), ..., F(M)] & \text{(rot1)} \\
N - N \Rightarrow 0 & \text{(minus4)} \\
[F(1), ..., F(N)] \ <>\ [G(1), ..., G(M)] \Rightarrow & \\
\qquad [F(1), ..., F(N), G(1), ..., G(M)] & \text{(box3)} \\
0 + X \Rightarrow X & \text{(pluszeroleft)} \\
\Box(N, (comb(0, F, G))) \Rightarrow \Box(N, G) & \text{(combdef2)} \\
X = X \Rightarrow trueP & \text{(idty)}
\end{array}$$

[2] *λClam* applies the equations exhaustively but does not currently try to reduce arithmetic expressions to a normal form.

[3] *Clam* can prove it but only with the aid of a critic.

λ*Clam* automatically constructs the proof below. In this presentation, we use the notation $\Downarrow$ *name* to indicate application of a rewrite rule (*name*).

$$\begin{array}{c}
\vdash rot(len([F_1(1),...F_1(V_0)]),[F_1(1),...F_1(V_0)] \;=\; [F_1(1),...F_1(V_0)]) \\
\Downarrow \; length3 \\
\vdash rot(V_0,[F_1(1),...F_1(V_0)]) \;=\; [F_1(1),...F_1(V_0)] \\
\Downarrow \; rot1 \\
\vdash [] \;<>\; [F_1(1),...F_1(V_0)] \;=\; [F_1(1),...F_1(V_0)] \\
\Downarrow \; minus4 \\
\vdash [] \;<>\; [F_1(1),...F_1(V_0)] \;=\; [F_1(1),...F_1(V_0)] \\
\Downarrow \; box3 \\
\vdash [F_1(1),...F_1(V_0)] \;=\; [F_1(1),...F_1(V_0)] \\
\Downarrow \; pluszeroleft \\
\vdash [F_1(1),...F_1(V_0)] \;=\; [F_1(1),...F_1(V_0)] \\
\Downarrow \; combdef2 \\
\vdash [F_1(1),...F_1(V_0)] \;=\; [F_1(1),...F_1(V_0)] \\
\Downarrow \; idty \\
\vdash trueP
\end{array}$$

Three proof steps — application of equations $minus4$, $pluszeroleft$, and $combdef2$ — do not change the portrayed form of the proof. They should therefore be completely suppressed, or only reported briefly.[4]

9 Results

All of the theorems about lists in the standard corpus of *Clam* were imported into λ*Clam*. Systematic tests showed that our initial implementation of ellipsis proves, without list induction, 50% of the test theorems which λ*Clam* proves with list induction. The results are tabulated in figure 1. One additional theorem (the last one in figure 1) was added to the test set; the ungeneralised form which was presented in §5.1 of the rotate-length conjecture.

The tested version of λ*Clam* was unable to prove the *member* examples because of a problem using the definition of *member* which is suitable for elliptic proofs — $member(x, \Box(n, F)) \leftrightarrow \exists i \leq n \,.\, x = F(i)$. We expect to fix this problem soon.

We plan to increase this 50% figure in three steps:

1. Fixing the problem which prevented the application of the elliptic definition of *member*.
2. Implementation of a method for normalising arithmetic expressions in λ*Clam*. Currently, the portrayal code is able to normalise arithmetic expressions but λ*Clam* is not.

[4] It is interesting to ponder to what extent there is a correspondence between these "null" proof steps and proof steps which would be considered "trivial" by a human.

3. Implementation of conditional rewriting and methods for solving simple inequalities. This third step should enable the system to prove using ellipsis all of the theorems that λ*Clam* can prove using induction, and more besides.

Conjecture	Ellipsis	List Induction
$l <> nil = l$	Y	Y
$reverse(l) <> reverse(m) = reverse(m <> l)$		Y
$l <> (m <> n) = (l <> m) <> n$		Y
$l = m \rightarrow (x <> l) = (x <> m)$		Y
$len(l <> m) = len(m <> l)$	Y	Y
$len(l) = len(reverse(l))$	Y	Y
$len(l <> m) = len(l) + len(m)$	Y	Y
$member(x, l) \rightarrow member(x, l <> m)$		Y
$member(x, m) \rightarrow member(x, l <> m)$		Y
$member(x, l) \vee member(x, m) \rightarrow member(x, l <> m)$		Y
$nth(n, nil) = nil$	Y	Y
$qrev(l, nil) = rev(l)$	Y	
$qrev(l, m) = rev(l) <> rev(m)$	Y	Y
$reverse(x :: nil) = x :: nil$	Y	Y
$reverse(l <> (x :: nil)) = x :: reverse(l)$		Y
$reverse(l) <> (x :: nil) = reverse(x :: l)$		Y
$rot(len(l), l) = l$	Y	

Fig. 1. Performance of λ*Clam* with ellipsis versus list induction proof strategies on a subset of *Clam*'s list theory. Conjectures which are proved are marked with a Y in the relevant column. For space reasons, we omit quantifiers. In the conjectures above, all free variables are universally quantified (l,m, and n are quantified over $list(nat)$ and x is quantified over nat).

Note that some of the elliptic proofs still use induction over the natural numbers. For example, the proof that $len(l <> r) = len(r) + len(l)$ uses induction over natural numbers after the use of ellipsis in order to prove the commutativity of addition.

An interesting failed proof attempt is *appreverse*:

$$\forall l, m\, list\, .\, reverse(l) <> reverse(m) = reverse(m <> l)$$

The proof attempt stumbles when it is unable to prove:

$$\begin{aligned}\Box(n+m, comb(n, (\lambda i.F(n-i+1)), (\lambda i.G(m-i+1)))) \quad &= \\ \Box(m+n, (\lambda i.(comb(m, G, F)(m+n-i+1))))&\end{aligned}$$

Proof of this goal is difficult because to make the two sides of the equality syntactically equal, we must rewrite the application term $(\lambda i.(comb(m, G, F)))\,(m+$

$n - i + 1$) to a *comb* term. This involves reasoning about inequalities to decide the values of i which cause the application term to fall into either the first or the second case of the definition of *comb* (3).

10 Discussion

As noted in §7.2, the need for a clear elliptic presentation of a proof can affect the order in which proof steps are carried out. If the set of rewrite rules which is applied during the proof is not confluent, then the changes caused by reordering the application of rewrite rules in the proof can be significant, and can lead for example to different lemmas being applied or to failure of the proof. This possibility of causing fundamental changes in the proof indicates that proof portrayal cannot be relegated to a "pretty-printing" role but must instead be considered at the time the conjecture is proved.

Some functions do not easily lend themselves to representation in the $\Box$ formulation, for example *flatten* over arbitrarily nested lists. There may be a correspondence between such difficult examples and recursive definitions which are difficult to understand.

11 Related work

We briefly mentioned in (§1) that our approach can be extended by means of the higher-order `fold` function to the representation and manipulation of formulae involving ellipsis where the main connective is not list `cons`. If the function $\otimes$ is associative, then the portrayal in equation (2) can be simplified by removing the brackets — $foldl(\otimes, a, \Box(n, F))$ is portrayed as $a \otimes F(1) \otimes \ldots \otimes F(n)$. Such an approach produces a similar formulation to the "Three Dots Language" (TDL) presented in [7], in which (following the terminology of [7]) the iteration star "$*$" is essentially a higher-order $fold$ function (compare the equations (1) defining $foldl$ above with the expansion equations in [7, p.231]), and the iteration counter "$\hat{c}$" represents lambda abstraction. For example, the elliptic formula $\forall x^* \exists d\,.\,(x_1 \cdot x_1 + \ldots + x_1 \cdot x_n \le a_{b+k} \cdot \ldots \cdot a_{d+k})$ is represented in TDL by equation (5) below (equation 13 of [7, p.237]) and in our formalism by (6).

$$\forall x^* \exists d\,.\,(((x_1 \cdot x_{\hat{c}}) \ \ +^* \ \hat{c}, 1..n) \le (a_{\hat{c}+k} \ \ \cdot^* \ \hat{c}, b..d)) \tag{5}$$

$$\begin{aligned} \forall^F x \exists d\,.\, & foldl(\lambda z \lambda w. x(1) \cdot z + w, 0, \Box(n, (\lambda\, i\,.\, x(i)))) \le \\ & \quad foldl(\lambda z \lambda w. z \cdot w, 1, \Box(d - b + 1, (\lambda\, i\,.\, a(b + k - 1 + i)))) \end{aligned} \tag{6}$$

Since TDL concentrates on defining a small mathematical language in which terms can be reduced to a normal form, it is quite restrictive. Our approach allows us to represent and manipulate quite general kinds of elliptic formulae (for example subscripts can be nonconsecutive because of the presence of a *comb* operator (which has no equivalent in TDL)). We have tested our formalism in an automated theorem proving system (λ*Clam*).

12 Conclusion

In this paper we have proposed a mechanism for allowing ellipsis in automatic proofs. The key idea is to use an internal notation in which the ambiguity inherent in elliptic notation is resolved. This uses a second-order functional $\Box$, which is similar to $\sum$ and $\prod$. Ellipsis is recovered from this notation by portray-like print routines which hide the internal notation and replace it with ellipsis.

With this notation many functions which normally require recursive definitions can be given explicit ones. As a result induction and generalisation can be eliminated from many proofs which normally require them. The result is proofs which seem closer to ordinary mathematical intuitions, in fact, we might describe these as more 'informal' proofs. Axiomatisation of lists using $\Box$ has a similar flavour to the work described in [2], but the representation and its use for proofs using ellipsis that we present are new.

A heavy burden is transferred to the portray routines. To present intuitively satisfying formulae and proofs they must carry out significant rewriting to transform the internal representation into a printable form. They must also make subtle decisions about which elements of an elliptic sequence are portrayed and which suppressed. It also may be necessary to rearrange the order in which rewrite rules are applied. This indicates that in general we need to consider how to present proofs clearly at the time the proof is constructed; we cannot leave it to a post-processing step.

References

1. S. Baker. *Aspects of the Constructive Omega Rule within Automated Deduction.* PhD thesis, Edinburgh, 1993.
2. R. S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, pages 5–42. Springer-Verlag, 1987. International Summer School. Proceedings of the NATO Advanced Study Institute, Marktoberdorf.
3. A. Bundy. The use of explicit plans to guide inductive proofs. In R. Lusk and R. Overbeek, editors, *9th International Conference on Automated Deduction*, pages 111–120. Springer-Verlag, 1988. Longer version available from Edinburgh as DAI Research Paper No. 349.
4. A. Bundy, F. van Harmelen, C. Horn, and A. Smaill. The Oyster-Clam system. In M. E. Stickel, editor, *10th International Conference on Automated Deduction*, pages 647–648. Springer-Verlag, 1990. Lecture Notes in Artificial Intelligence No. 449. Also available from Edinburgh as DAI Research Paper 507.
5. A. Ireland and A. Bundy. Productive use of failure in inductive proof. *Journal of Automated Reasoning*, 16(1–2):79–111, 1996. Also available as DAI Research Paper No 716, Dept. of Artificial Intelligence, Edinburgh.
6. E. B. Kinber and A. N. Brazma. Models of inductive synthesis. *Journal of Logic Programming*, 9:221–233, 1990.
7. Leon Łukaszewicz. Triple dots in a formal language. *Journal of Automated Reasoning*, 22(3):223–239, March 1999.

8. J.D.C Richardson, A. Smaill, and I.M. Green. System description: proof planning in higher-order logic with lambdaclam. In C. Kirchner and H. Kirchner, editors, *Proceedings of CADE-15*, volume 1421 of *Lecture Notes in Computer Science*. Springer Verlag, 1998.