Using Middle-Out Reasoning to Control the Synthesis of Tail-Recursive Programs

Jane Hesketh, Alan Bundy and Alan Smaill

DAI Research Paper No.

October 28, 1992

Submitted to CADE-92

Department of Artificial Intelligence
University of Edinburgh
80 South Bridge
Edinburgh EH1 1HN
Scotland

Using Middle-Out Reasoning to Control the Synthesis of Tail-Recursive Programs[1]

Jane Hesketh, Alan Bundy and Alan Smaill

Address:	Dept of Artificial Intelligence, University of Edinburgh,
	80, South Bridge, Edinburgh, Scotland.
Telephone:	+44 31 650 {2718,2716,2710}
Email:	{hesketh, bundy, a.smaill}%ed.ac.uk@nsfnet-relay.ac.uk

Abstract

We describe a novel technique for the automatic synthesis of tail-recursive programs. The technique is to specify the required program using the standard equations and then synthesise the tail-recursive program using the proofs as programs technique. This requires the specification to be proved realisable in a constructive logic. Restrictions on the form of the proof ensure that the synthesised program is tail-recursive.

The automatic search for a synthesis proof is controlled by proof plans, which are descriptions of the high-level structure of proofs of this kind. We have extended the known proof plans for inductive proofs by adding a new form of generalisation and by making greater use of middle-out reasoning. In middle-out reasoning we postpone decisions in the early part of the proof by the use of meta-variables which are instantiated, by unification, during later parts of the proof. Higher order unification is required, since these meta-variables can represent higher order objects.

The program synthesised is automatically verified to ensure that it satisfies its specification. This type of verification is contrasted with template-based transformation approaches which require proofs that the general transformations described by the templates preserve equivalence.

The technique described is more general than template-based approaches, since it is not tied to program patterns which must be specified in advance. Detailed information about proof structure enables it to use a wider repertoire of rewritings in a more goal-directed way than comparable transformational techniques.

1 Introduction

Consider the following two definitions[2] of procedures for reversing lists.

$$\begin{aligned} rev_n(nil) &= nil \\ rev_n(h :: t) &= append(rev_n(t), h :: nil) \end{aligned}$$

$$\begin{aligned} rev_t(l) &= rev_2(l, nil) \\ rev_2(nil, a) &= a \\ rev_2(h :: t, a) &= rev_2(t, h :: a) \end{aligned}$$

[1] The research reported in this paper was supported by SERC grant GR/E/71799, and an SERC Senior Fellowship to the second author. We are grateful for feedback on this paper from Andrew Ireland and Toby Walsh, and for conversations with Dale Miller and other members of the Mathematical Reasoning Group at Edinburgh.

[2] Notice that taking such definitions as equations, as we do, expresses more than the corresponding functional program, which uses the equations' computational content reading from left to right only. This distinction is significant in that the equations constitute a statement about the program at the specification level, not just the program itself. Although currently we rely on the recursive definitions to guide the initial formulation of the problem for the planner, in principle our system could work with a specification which was not completely and purely equationally specified.

where :: is infix *cons*.

The auxiliary procedure, rev_2, is *tail recursive*. That is, recursive calls to rev_2 occur only as the outermost function of the procedure body. The *accumulator* argument, a, is used to build up the output as the recursion is entered, so that nothing remains to be done as the recursion exits. This has important consequences for the efficiency of rev_2 and, hence, rev_t. It is not necessary to maintain a stack of recursive calls during its implementation, which cuts down considerably on the space requirements of a procedure call. Thus, tail recursive procedures can be compiled into iterative ones. In contrast, we will call rev_n a *naïve* procedure. It is the obvious definition of list reversal, and so easy for programmers to discover, but is much less space efficient.

In many cases it is possible to transform naïve procedures into equivalent tail recursive ones. For instance, any linearly recursive procedure can be transformed into a tail-recursive one [Wikström 87], although the general process for doing this creates rather convoluted and sub-optimal tail-recursive procedures. Various attempts have been made to automate tail recursive transformation, *e.g.* [Darlington 81], thus freeing the programmer from the burden of discovering the more efficient, but more complex, definition. Another approach, described in [Huet & Lang 78], uses general second-order templates of transformations, but is inevitably limited to such cases as have been anticipated and proved to preserve equivalence. In this paper we consider a novel technique for automating the synthesis of tail recursive procedures based on the *proofs as programs* technique.

Proofs as programs is a technique for synthesising computer programs from proofs that their specifications are realisable. Suppose $spec(inputs, output)$ is a logical relation between the inputs and outputs of a program. We prove the *specification theorem*:

$$\forall inputs, \exists output.\ spec(inputs, output)$$

in a constructive logic. Intuitively, this proof must show how given any combination of inputs, an output can be constructed that meets the specification. This construction can be extracted from the proof and expressed as the required program. The use of a constructive logic excludes 'pure existence' proofs, where the existence of an output is proved without a suitable construction being exhibited. By using an automatic theorem prover to prove the specification theorem we can automate the process of program synthesis.

Different proofs of the same specification theorem yield different procedures meeting the same specification. For instance, from a specification of list reversal we can synthesise either the naïve or the tail recursive procedure, according to the proof we find. We have identified a characterisation of proofs that synthesise tail recursive procedures, following [Wainer 89]. The essence is that the witnesses of the two existential quantifiers, one in the induction hypothesis and one in the induction conclusion, should be identical. The *witness* of an existential quantifier is the object which is the evidence of the existence asserted. If we restrict our theorem prover to proofs of this form then we can guarantee to synthesise only tail recursive functions. These witnesses give the value of the function. Equality between them means that the function does not change value as the recursive call is exited.

A key idea is that we can use the naïve equational definition as a specification of the tail recursive program. If $f_n(inputs)$ is the naïve definition, then $output = f_n(inputs)$ can serve as $spec(inputs, output)$. We can prevent the trivial solution to:

$$\forall inputs, \exists output.\ output = f_n(inputs)$$

in which $f_n(inputs)$ is substituted for *output* by insisting on a proof where f_n does not appear as part of the witness.

Achieving proofs with the desired form automatically, requires some insight into the proof process. The synthesis of our equivalent to rev_2 involves a generalised version of the specification theorem which is then be shown to be imply the original. This step guarantees that the synthesised function satisfies the original specification. The exact nature of the generalised theorem is not known initially, but its overall form can be described. Middle-out reasoning is used to identify its components and these define the synthesised function. This reasoning is guided to satisfy the criterion that the existential witnesses for the output are identical in induction hypothesis and conclusion All this is implemented using a proof planning system.

2 Tail-Recursive Reverse Example

To illustrate this process, consider the synthesis of tail recursive reverse. Using the naïve reverse to specify the required procedure gives the specification theorem:

$$\vdash \forall x, \exists z.\ z = rev_n(x)$$

Firstly, we generalise this theorem to:

$$\vdash \forall x, \forall a, \exists z.\ z = append(rev_n(x), a) \tag{1}$$

which leaves us with the obligation to show that the generalised theorem entails the original one:

$$\forall x, \forall a, \exists z.\ z = append(rev_n(x), a) \quad \vdash \quad \forall x, \exists z.\ z = rev_n(x)$$

This generalisation is an example of an *eureka step*. It seems to come 'out of the blue' with no apparent motivation. A major problem of automation of tail recursive transformation is to find techniques for calculating such eureka steps.

To prove (1), the generalised theorem, we use simple list induction on x.

2.1 The Step Case

The step case of this induction is:

$$\forall a, \exists z.\ z = append(rev_n(t), a) \quad \vdash \quad \forall a, \exists z.\ z = append(rev_n(h :: t), a)$$

We rewrite the induction conclusion using first the definition of rev_n:

$$\forall a, \exists z.\ z = append(rev_n(t), a) \quad \vdash \quad \forall a, \exists z.\ z = append(append(rev_n(t), h :: nil), a)$$

and then the associativity of *append*:

$$\forall a, \exists z.\ z = append(rev_n(t), a) \quad \vdash \quad \forall a, \exists z.\ z = append(rev_n(t), append(h :: nil, a))$$

which simplifies using the definition of *append* to:

$$\forall a, \exists z.\ z = append(rev_n(t), a) \quad \vdash \quad \forall a, \exists z.\ z = append(rev_n(t), h :: a)$$

The step case can now be finished off by stripping off the universal quantifiers and instantiating the a in the induction hypothesis to $h :: a$ in the induction conclusion. The induction hypothesis and induction conclusion are then identical. Note that the existentially quantified variables, z, in the induction hypothesis and the induction conclusion are equal. This equality ensures that a tail recursive procedure will be synthesised by the proof.

2.2 The Base Case

The base case is:

$$\vdash \quad \forall a, \exists z.\ z = append(rev_n(nil), a)$$

The definitions of rev_n and *append* simplify this to

$$\vdash \quad \forall a, \exists z.\ z = a$$

which can be proved by instantiating z to a.

2.3 The Justification

It only remains to prove the original theorem from the generalised one.

$$\forall x, \forall a, \exists z.\ z = append(rev_n(x), a) \quad \vdash \quad \forall x, \exists z.\ z = rev_n(x)$$

Stripping the two $\forall x$ quantifiers while identifying the xs, gives:

$$\forall a, \exists z.\ z = append(rev_n(x), a) \quad \vdash \quad \exists z.\ z = rev_n(x)$$

Instantiating a to nil gives:

$$\exists z.\ z = append(rev_n(x), nil) \quad \vdash \quad \exists z.\ z = rev_n(x)$$

The hypothesis now simplifies to the conclusion.

The proof is now complete. Analysis of the proof gives the required tail recursive definition of rev_t.

$$\begin{aligned} rev_t(l) &= rev_2(l, nil) \\ rev_2(nil, a) &= a \\ rev_2(h :: t, a) &= rev_2(t, h :: a) \end{aligned}$$

3 Proofs as Programs

We have implemented the proofs as programs technique in the OYSTER system, [Bundy *et al* 90a]. OYSTER is a interactive theorem prover for intuitionalist type theory, a higher order, constructive, typed logic based on Martin Löf Type Theory. It is a Prolog reimplementation of the Nuprl system. One advantage of using this logic for program synthesis is that every rule of inference of the logic has an associated program construction rule, so that the synthesised program is built as a side effect of constructing the proof, and no post-analysis of the proof is required to extract it.

The technical basis of this program construction process is that every statement of the logic has the form $a \in A$, with three possible readings:

1. a is an object of type A;

2. a is a proof of proposition A;

3. a is a program meeting the specification A. (a is also called the *extract term*.)

The first two readings were discovered by Curry and Howard, and are called the 'Curry-Howard isomorphism' and the 'propositions as types principle'. The third reading extends this to the 'proofs as programs principle'

Our synthesis technique exploits the ambiguity provided by these three readings. We treat the input and output variables as objects belonging to some type, *e.g.* lists of natural numbers, and the specification as a proposition over these objects. We prove the specification suppressing the details of its inhabiting extract term. When the proof is complete this extract term is revealed and interpreted as a functional program.

Following Nuprl, OYSTER provides the facility for writing tactics. These are programs (in OYSTER's case Prolog programs) which apply rules of inference of the logic. Tactics can embody some heuristic ideas about how the proof should proceed, and thus be used to guide the theorem prover. The rest of this paper is concerned with the nature of the heuristics we have developed to guide inductive proofs, in particular tail recursive transformations, with how these heuristics are embodied in the OYSTER system, and with how they can automate the discovery of eureka steps.

4 Proof Plans

The recent work of the Mathematical Reasoning Group at Edinburgh derives from the Boyer-Moore, [Boyer & Moore 79], characterisation of proofs as a combination of special purpose components, such as symbolic evaluation and induction, which act in concert. We have reconstructed and extended the Boyer-Moore components and implemented them as OYSTER tactics. Moreover, we use an AI plan formation program, CLAM, to link these components together, [Bundy *et al* 91]. Each tactic is specified by a *method* using a meta-logic. The pair of tactic and method is called a *proof plan*. CLAM operates on a meta-level representation of the proof tree, and uses meta-level reasoning on the methods to select a combination of tactics customised to the current theorem. This combination can then be executed in the object-level logic to produce a proof.

In particular, the meta-level representation of the proof can contain meta-variables. These meta-variables act as place holders for expressions to whose precise identity we are not yet ready to commit. They are introduced during the meta-level application of rules of inference which at the object-level require some commitment, and are later instantiated by unification during the planning process. Meta-variables can be of higher-order type, so higher order unification is required to instantiate them. Meta-variables can be introduced, for instance, during the stripping of an existential quantifier on the right hand side of a sequent or a universal quantifier on the left hand side of a sequent, or during the generalisation of a theorem. At the object-level these proof steps require a commitment to some particular expression, *e.g.* as the witness of a quantifier. We call this use of meta-variables *middle-out reasoning*, because it allows us to turn the search space inside out — doing the middle of the proof first and the beginning later.

The proof plans technique is easily adapted to deal with a variety of types of proof using a body of general-purpose proof plans. Methods describe the preconditions under which their tactics are applicable and the effect of their tactics on given input sequents. A simple, iterative-deepening search strategy is used by the planner. No combinatorial explosion is encountered due to the small size of the search space defined by the proof plans. The object-level search space, on the other hand, is huge, *cf.* [Bundy *et al* 91].

Some standard proof plans are:

induction : the selection and application of an appropriate induction rule;

symbolic evaluation : the simplification of expressions using the unfolding of definitions;

rippling : the rewriting of the induction conclusion to make it resemble the induction hypothesis;

tautology : propositional tautology checking plus simple equality manipulation;

existential : use of middle-out reasoning to postpone the selection of an existential witness;

fertilization use of the induction hypothesis to prove the induction conclusion.

By using these proof plans in appropriate combinations, CLAM is capable of proving a sizeable body of theorems from the Boyer-Moore corpus. As in the Boyer-Moore theorem-prover, theorems which have already been proved may be made available as lemmas for later proofs, but this increases the size of the search space.

The key proof plan is rippling. It is also the only one that is not self-explanatory. We outline it below.

In the step case of an induction proof we prove the induction conclusion from the induction hypothesis. These two formulae are very similar. We call their points of difference, *wave fronts*. Wave fronts are expressions with holes in them. We indicate them by putting them in boxes, *e.g.*

$$\forall a, \exists z.\ z = append(rev_n(t), a) \quad \vdash \quad \forall a, \exists z.\ z = append(rev_n(\boxed{h ::\ }t), a)$$

The role of rippling is to move these wave fronts from their innermost position around the induction variable to somewhere where they will not block the matching of induction hypothesis to induction conclusion. There are two such places: (1) surrounding the entire induction conclusion, and (2) surrounding an universally quantified variable, *e.g.* an accumulator. This movement is effected by rewrite rules called *wave rules*. There are two kinds corresponding to the two directions of movement: longitudinal wave rules move the wave fronts outwards, and transverse wave rules move the wave fronts sideways. To move wave fronts to target (1) involves purely longitudinal wave rules, but to move them to target (2) involves longitudinal wave rules followed by transverse wave rules, and sometimes followed by longitudinal wave rules applied backwards.

Examples of longitudinal wave rules are:

$$\begin{aligned} append(X, \boxed{append(}Y\boxed{,Z)}) &\Rightarrow \boxed{append(}append(X,Y)\boxed{,Z)} && (2)\\ rev_n(\boxed{X ::}Y) &\Rightarrow \boxed{append(}rev_n(Y)\boxed{,X :: nil)} && (3)\\ append(\boxed{X ::}Y, Z) &\Rightarrow \boxed{X ::}append(Y,Z) \end{aligned}$$

and examples of transverse wave rules are:

$$\begin{aligned} append(\boxed{append(}X\boxed{,Y)}, Z) &\Rightarrow append(X, \boxed{append(Y,}Z\boxed{)}) && (4)\\ \boxed{s(}X\boxed{)} + Y &\Rightarrow X + \boxed{s(}Y\boxed{)} \end{aligned}$$

X, Y and Z in these rules are *meta-variables*, *i.e.* these rewrite rules are really rule schemata. We adopt the convention that upper case letters represent meta-level variables and lower case letters represent object-level variables. Using meta-variables facilitates the use of unification at the meta-level during middle-out reasoning.

The rewriting of:

$$\forall a, \exists z.\ z = append(rev_n(\boxed{h ::}t), a)$$

to:

$$\forall a, \exists z.\ z = append(\boxed{append(}rev_n(t)\boxed{,h :: nil)}, a)$$

using longitudinal wave rule (3) is an example of rippling outwards, and the application of transverse wave rule (4) to this to produce:

$$\forall a, \exists z.\ z = append(rev_n(t), \boxed{append(h :: nil,}a\boxed{)})$$

is an example of rippling sideways. For more details see [Bundy *et al* 90b].

5 Proof Plans for Tail-Recursive Synthesis

For tail-recursive synthesis, the existing proof plans were adapted, mostly in minor ways to inhibit inadvertent application in the presence of meta-variables. The rippling proof plan was significantly changed, and a new generalisation proof plan was added to create and introduce the generalised theorem. These will now be described in detail.

5.1 Tail-Recursive Generalisation

We add a new proof plan for tail-recursive generalisation. Our overall strategy is to generalise in order to be able to construct a proof with the special characteristic of not altering the existential witnesses in the induction step case. In particular, the generalisation introduces an accumulator. Suppose $f_n(inputs)$ is the naïve procedure and $f_t(inputs)$ is the tail-recursive procedure. We will synthesise a procedure $f_2(inputs, accumulator)$ and define $f_t(inputs) = f_2(inputs, a_0)$, for some particular value, a_0, of the accumulator. We, therefore, want to generalise the specification, $outputs = f_n(inputs)$, of f_t to a specification, $outputs = g(f_n(inputs), accumulator)$, of f_2. and recover the value a_0 from the justification branch of this generalised proof. Since we don't know what value of g to use to wrap $f_t(inputs)$ and $accumulator$ together, we will use a meta-variable, which will be instantiated by later stages of the proof, *i.e.* the new specification will be: $outputs = G(f_n(inputs), accumulator)$.

5.1.1 Preconditions

The proof plan applies when the goal is of the form:

$$\forall x, \forall \underline{y}, \exists z.\ z = f_n(x, \underline{y})$$

where $\underline{y}$ represents zero or more additional parameters.

This is a universally quantified expression consisting of an equality between an existentially quantified variable and an arbitrary term containing any of the universally quantified variables but not the existentially quantified one.

If middle-out reasoning is already taking place on the goal, initiating further middle-out reasoning is likely to be unwieldy or explosive, so a check is made that it does not currently contain any meta-variables. Specifically, since the generalised goal has the same form as the ungeneralised one, the lack of such a check could result in generalisations of generalisations, etc.

5.1.2 Effects and Tactic

The first output sequent of the tactic is the generalisation of the original specification theorem, *i.e.*

$$\vdash \quad \forall x, \forall \underline{y}, \forall a, \exists z.\ z = G(f_n(x, \underline{y}), a)$$

where G is a meta-variable. The second output sequent is the justification proof branch: *i.e.* that the generalised theorem implies the original one.

$$\forall x, \forall \underline{y}, \forall a, \exists z.\ z = G(f_n(x, \underline{y}), a) \quad \vdash \quad \forall x, \forall \underline{y}, \exists z.\ z = f_n(x, \underline{y})$$

5.2 Longitudinal Wave Proof Plan

Rippling is applied during the step case of inductive proofs, after wave fronts have been inserted by the induction method. It consists of repeated, but selective, rewriting with longitudinal and transverse wave rules. Rippling works by calling two sub-proof-plans: longitudinal wave and transverse wave. Our versions of these are identical to the standard one, except that they take account of the possibility that the input sequent could contain meta-variables.

The longitudinal wave proof plan takes as input a sequent such as the generalisation:

$$\vdash \forall x, \forall \underline{y}, \forall a, \exists z.\ z = G(f_n(\boxed{c(\boxed{x})}, \underline{y}), a)$$

5.2.1 Preconditions

Assume that the longitudinal wave rule applied to this is[3]:

$$\phi(\boxed{\kappa(\boxed{X})}, \underline{Y}) \Rightarrow \boxed{\kappa'(\boxed{\phi(X, \underline{Y})}, \underline{Y})}$$

Wave rules like this are selected and tested in turn. In standard CLAM the left-hand-side of the rule would be tested to see if it unified as a whole with some subterm of the sequent. Expressions are annotated with marks to indicate wave fronts. The standard CLAM unification algorithm aligns these marks and hence the wave fronts in goal and rule. In our middle-out reasoning system, separate unifications are performed successively on each of the following expression pairs, and progressive instantiation takes place.

[3]This is a simplification, see [Bundy *et al* 90b] for general form of wave rules.

- the terms contained by the respective wave fronts, x and X
- the smallest terms containing the wave fronts, $c(x)$ and $\kappa(x)$
- the term to be rewritten and the whole left-hand side of the rule, $f_n(c(x), \underline{y})$ and $\phi(\kappa(x), \underline{Y})$.

The unifications are performed with the wave front marks removed. Wave fronts in goal and rule are still aligned by the first two stages of our three stage unification process.

5.2.2 Effects and Tactic

These are exactly as in the standard proof plan, the application of the lemma or definition is computed and planning continues. The tactic records what is to be applied, in which direction, and the position of the relevant subterm.

5.3 Transverse Wave Proof Plan

This incorporates changes to permit meta-variables as described for the longitudinal wave proof plan.

It takes as input a sequent conclusion such as:

$$\vdash \quad \forall \underline{y}, \forall a, \exists z.\ z = G(\boxed{c'(}\, f_n(x, \underline{y}) \boxed{, \underline{y})}, a)$$

5.3.1 Preconditions

Assume a transverse wave rule:

$$\phi(\boxed{\kappa'(}\, X \boxed{, \underline{Y})}, A) \rightarrow \phi(X, \boxed{\kappa''(\underline{Y},}\, A \boxed{)})$$

The positions of the wave fronts before and *after* the use of a lemma are used to ensure that the effect of the lemma is a ripple sideways, as intended.

Essentially everything proceeds as for the longitudinal wave proof plan, except that there is an extra unification stage. We insist that the term at the position around which the wave front is moved is unified with the last universally quantified variable in the induction conclusion, *i.e.* the accumulator added by the generalisation.

5.3.2 Effects and Tactic

Again, these are just the same as in standard CLAM.

6 Tail-Recursive Reverse Example Revisited

We now repeat the example of §2, but this time with annotations to explain how our proof plans are able to find this proof with very little search. Note particularly how meta-variables are used to postpone the commitment to existential witnesses and generalisations.

As before, the specification goal is:

$$\vdash \quad \forall x, \exists z.\ z = rev_n(x)$$

The only proof plans whose preconditions may be satisfied are induction and tail-recursive generalisation. After some search induction fails the tail-recursive restriction about existential witnesses. The result of generalisation is a new sequent:

$$\vdash \quad \forall x, \forall a, \exists z.\ z = G(rev_n(x), a)$$

and a further justification sub-goal to show that this entails the original theorem.

6.1 Proving the Generalisation Using Induction

The only proof plans which might conceivably apply to this are existential, tail-recursive generalisation and induction. The first two are barred since the meta-sequent already contains a meta-variable. Induction is unaffected by the meta-variable. The preconditions of the induction strategy proof plan look ahead to try to find an induction rule that will allow rippling to proceed ([Bundy *et al* 89] explains this look-ahead process). This look-ahead suggests a simple list induction on x.

6.1.1 The Step Case

$$\forall a, \exists z.\ z = G(rev_n(t), a) \quad \vdash \quad \forall a, \exists z.\ z = G(rev_n(\boxed{h ::\,}t), a)$$

The only applicable proof plan for this meta-sequent is rippling. Again the existential and tail-recursive generalisation's preconditions would fail due to the presence of the meta-variable.

The rippling proof plan could select several wave rules, depending on which were available. It is always debatable, when discussing wave rules, which ones should be assumed to be available. There are broadly three options:

- Minimal - Only necessary definitions, anything else to be created.
- Average - Definitions, along with some collection of lemmas. Specifically *not* just such lemmas as will make for the proof in hand easy.
- Maximal - everything, no matter how trivial, which is true for the theory.

The last of these is impossible. The first is an interesting but not particularly realistic case. What mathematician or automatic theorem prover would we expect to derive everything from first principles all the time? So we have taken the middle way and provided an average collection of rules: 20 longitudinal rules and 12 transverse ones. Below we explain what choices they present to the system reasoning middle-out. From this it should be clear that they do not make the task artificially simple.

In our example the only possible match, which obeys the constraints introduced by rippling, is, (3), the definition of rev_n. This applies, producing a new sequent:

$$\forall a, \exists z.\ z = G(rev_n(t), a) \quad \vdash \quad \forall a, \exists z.\ z = G(\boxed{append(\boxed{rev_n(t)}, h :: nil)}, a)$$

This is submitted afresh to the planner, and exactly the same proof plans apply as for its predecessor — namely only rippling. Each wave-rule is considered in turn. For each, as described before, there is progressive unification of the terms within the wave front, the wave front term, and the whole left-hand-side of the rule with whatever subterm of the conclusion it will match. This is vital not only to achieve the effect we want, but also to control the higher order unification process. At any point, one of these unifications may fail, and the planner will backtrack to get the next wave-rule.

Here either (2) or (4) could apply. Consider the branch of the search space in which (4) is applied. This is a sideways ripple towards the accumulator a. The unification instantiates G to $\lambda u \lambda v.append(u, v)$, so, after β-reduction, the result of the rule application is:

$$\forall a, \exists z.\ z = append(rev_n(t), a) \quad \vdash \quad \forall a, \exists z.\ z = append(rev_n(t), \boxed{append(h :: nil, \boxed{a})})$$

Rippling finishes by applying symbolic evaluation to the wave front, which gives:

$$\forall a, \exists z.\ z = append(rev_n(t), a) \quad \vdash \quad \forall a, \exists z.\ z = append(rev_n(t), \boxed{h ::\,}a)$$

Now fertilisation can apply, matching the induction hypothesis and induction conclusion and finishing the step case. a in the induction hypothesis is instantiated to $h :: a$ from the induction conclusion.

6.1.2 The Base Case

The step case work has instantiated G, so the meta-sequent is

$$\vdash \quad \forall a, \exists z.\ z = append(rev_n(nil), a)$$

The existential proof plan could apply, and would lead to the introduction of $append(rev_n(nil), a)$ for z. This would be cumbersome, but not wrong. Symbolic evaluation applies twice, using the base definitions of rev_n and $append$:

$$\vdash \quad \forall a, \exists z.\ z = a$$

Now, the existential proof plan is the only one which can apply, it introduces a and a meta-variable Z for z. The tautology proof plan now applies, instantiating Z to a, and completing the base case.

6.2 The Justification

Now that G has been instantiated, the justification sub-goal is to prove:

$$\forall x, \forall a, \exists z.\ z = append(rev_n(x), a) \quad \vdash \quad \forall x, \exists z.\ z = rev_n(x)$$

First, we assume that any universally quantified variables in the conclusion should be identified with their counterparts in the generalisation hypothesis. So each of these is introduced, and echoed in the hypothesis.

$$\forall a, \exists z.\ z = append(rev_n(x), a) \quad \vdash \quad \exists z.\ z = rev_n(x)$$

An instantiation, a_0, of a is chosen such that the z in the hypothesis is equal to the z in the conclusion, *i.e.* $append(rev_n(x), a_0) = rev_n(x)$.

We can postpone the choice of a_0 by middle-out reasoning. This requires the universal proof plan, a dual of the existential proof plan, which works on universal quantifiers in the hypothesis. A meta-variable, A, is inserted for a, and the symbolic evaluation proof plan is applied to that hypothesis alone, *i.e.* to the expression

$$\exists z.\ z = append(rev_n(x), A)$$

This instantiates A to nil and reduces the hypothesis to:

$$\exists z.\ z = rev_n(x)$$

to which fertilization applies, finishing the justification step, and, hence, the whole proof.

7 Using Higher Order Unification

As noted above, higher-order unification is required to implement middle-out reasoning in our higher-order logic. Unfortunately, higher-order unification is undecidable; it may fail to terminate when two expressions are not unifiable, and it may produce an unlimited number of unifiers when they are. If we only test for unifiability we can obtain better behaviour. Higher order unifiability is semi-decidable. Therefore, we have implemented the higher order unifiability algorithm invented by Huet, [Huet 75], which yields unifiers if there are any. This algorithm uses a simply typed lambda calculus, which has proved adequate in practice. However, since our logic includes dependent types, we intend to extend our algorithm to Pym's version for dependent types, [Pym 90], in the near future.

Higher order unification may provide more than one unification, possibly infinitely many. Not all of these will be sensible for our problem. Below we explain how a choice is made.

7.1 Interfacing Middle-Out Reasoning and Higher Order Unification

All meta-variables are treated as variables in the higher order unification. Additionally, any universally quantified variables in lemmas or hypotheses are treated as variables. Anything else is deemed a constant. Some pre-processing is necessary:

1. Copies are made of both the terms to be compared and their contexts, which are used for type-checking. The context of a term to be unified is the sequent or theorem from which it originates.
2. The types of the two terms to be compared are tested to ensure they are the same. If not, the algorithm fails immediately.
3. The copies of the terms are normalised by β-reduction.
4. The types of all the symbols in the ground copies of the terms are guessed and recorded.
5. The expressions to be compared are further normalised by η-conversion.
6. All the arguments throughout the term are checked to ensure that they have the correct types to suit the functions they are in.

In practice, it is usually possible to reduce the problem to a sequence of higher-order matches rather than unification. Matching is much more controllable.

7.2 Inadmissible Unifications

Unification suggests instantiations for meta-variables, some of which may be legal but not sensible.

Suppose, for example, we have the following sequent:

$$\forall a, \exists z.\ z = G(rev_n(t), a) \quad \vdash \quad \forall a, \exists z.\ z = G(append(rev_n(t), h :: nil), a)$$

where G is a meta-variable to be instantiated through the demands of the subsequent proof. We expect to instantiate G by considering unifications of

$$append(append(rev_n(t), h :: nil), a)$$

with

$$G(append(rev_n(t), h :: nil), a)$$

Higher order unification suggests four unifiers for G:

- $\lambda u \lambda v.append(u, v)$
- $\lambda u \lambda v.append(u, a)$
- $\lambda u \lambda v.append(append(rev_n(t), h :: nil), v)$
- $\lambda u \lambda v.append(append(rev_n(t), h :: nil), a)$

At this point in the proof, each of them would be a valid object to use for G in terms of OYSTER's logic. The last three are not sensible choices, because they build universally quantified or free variables into the identity of the function, G. As G is a function whose identity must be describable externally to the proof, and so should not be dependent on the names of variables used during proof, such instantiations are not appropriate. Effectively, they are barred by *temporal scoping* - a notion suggested by Dale Miller. For this reason, we discard them.

8 Comparison with Related Techniques

Other research has also tackled this problem, and it is related to work on program transformation. There are two major approaches. One is to develop general templates which have an input part, an output part and conditions as in [Huet & Lang 78], all described in terms of shared variable components. If a program matches the input part and the conditions on the components can be satisfied, the instantiation of the output part yields a tail-recursive version. The other approach, as taken in [Darlington 81], is to apply a sequence of transformations to the initial program, for example using definitions and known properties of functions. Although this has an outwardly similar appearance to the work reported here, and comparable search problems, there are some notable differences, as will be explained below.

Template-based approaches are static, in the sense that they must declare the form of their transformation in advance. This reduces their potential to handle new problems. Dynamic techniques, such as the one described here and in [Darlington 81] are more flexible, and able to use whatever information is available at the time.

Darlington's system does not ensure that optimisation has actually occurred during the transformation. In the proofs as programs characterisation, tail-recursiveness becomes the goal which drives the process. The requirement that the proof structure satisfy the criterion regarding the identity of the existential witnesses guides the strategy which controls the search for a synthesis proof.

The work described in this paper is completely automated. In contrast, the account given in [Huet & Lang 78] is theoretical. Their work concentrates on the preservation of equivalence by the technique without detailed examination of the search problems arising from the need to identify the values of higher-order variables present in the output program, but not the input. This identification is non-trivial in general. Darlington's approach is automated to a considerable degree, but full automation was never a goal of that system. Lacking the tools of the CLAM approach, such as the monitoring of wavefronts, it has less ability to characterise the types of operation, and so needs more guidance. Explicit instructions to unfold using definitions and then use an associativity lemma provide such guidance. The various kinds of wave rules we have described are defined syntactically according to their effect on the conclusion of an inductive proof. Longitudinal waves are not only the recursive steps of definitions, but anything which can achieve the same effect of moving a wavefront upwards in the term structure. Transverse waves are not just derived from associative functions, they too are syntactically defined to include any lemmas which enable a wavefront to be moved sideways. In this sense the proof plans system is more general. It is, however, worth pointing out that Darlington's system is capable of other forms of optimisation than tail-recursiveness, which have not so far been implemented with proof plans.

A significant difference between the work described in this paper and the transformational approaches is that here we prove that the synthesised program satisfies the original specification. Each of these other techniques carries the burden of guaranteeing that their processes of transformation are equivalence-preserving. A disadvantage for our approach is that time must be spent proving this satisfaction on each occasion, but this is compensated for by the fact that any constraints required are those particular to this proof. We do not have to prove that the general process is equivalence preserving, as must be done by computational induction for each higher order template in [Huet & Lang 78].

9 Results and Conclusions

We have described a novel approach to the transformation of naïve procedures into tail-recursive ones. We specify the tail-recursive procedure using the naïve definition and synthesise it using the proofs as programs technique embodied in the OYSTER proof development system. Restrictions on the form of the proof ensure that the resulting extract term is bound to be tail-recursive. To control the search involved in this proof we have used the proof plans technique as embodied in the CLAM proof planner. This reduces a huge object-level search space to a small meta-level search space, which can be successfully searched by a weak general-purpose strategy: iterative-deepening.

We have introduced a new form of generalisation, which is required for tail-recursive transformation, and extended the middle-out reasoning capabilities of CLAM by allowing the use of meta-variables during generalisation. This has necessitated the improvement of some of the existing CLAM proof plans so that they can cope with meta-variables more successfully.

The approach has been successfully tested on the synthesis of procedures for the summation of an arbitrary function from 0 to an arbitrary value, *times*, *total*, *greatest* and *length*. In each case a simple and natural tail-recursive procedure is synthesised. We plan to continue testing. Initial results suggest that this is a powerful new technique which extends existing ones.

Further work needs to be done to specify all the restrictions which are appropriate to the choice of unification given our understanding of the aims of the task. Work is in progress to improve the flexibility of the unification process so that composite functions can be identified, and to extend Huet's algorithm to dependent types.

References

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

[Bundy *et al* 89] A. Bundy, F. van Harmelen, J. Hesketh, A. Smaill, and A. Stevens. A rational reconstruction and extension of recursion analysis. In N.S. Sridharan, editor, *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 359–365. Morgan Kaufmann, 1989. Also available from Edinburgh as DAI Research Paper 419.

[Bundy *et al* 90a] 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.

[Bundy *et al* 90b] A. Bundy, F. van Harmelen, A. Smaill, and A. Ireland. Extensions to the rippling-out tactic for guiding inductive proofs. In M.E. Stickel, editor, *10th International Conference on Automated Deduction*, pages 132–146. Springer-Verlag, 1990. Lecture Notes in Artificial Intelligence No. 449. Also available from Edinburgh as DAI Research Paper 459.

[Bundy *et al* 91] A. Bundy, F. van Harmelen, J. Hesketh, and A. Smaill. Experiments with proof plans for induction. *Journal of Automated Reasoning*, 7:303–324, 1991. Earlier version available from Edinburgh as DAI Research Paper No 413.

[Darlington 81] J. Darlington. An experimental program transformation and synthesis system. *Artificial Intelligence*, 16(3):1–46, August 1981.

[Huet & Lang 78] G. Huet and B. Lang. Proving and applying program transformation expressed with second order patterns. *Acta Informatica*, 11:31–55, 1978.

[Huet 75] G. Huet. A unification algorithm for lambda calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[Pym 90] D.J. Pym. *Proofs, search and computation in general logic.* Unpublished PhD thesis, University of Edinburgh, 1990. Available as LFCS report ECS-LFCS-90-125.

[Wainer 89] S.S. Wainer. Programs from proofs. Seminar given at the Department of Artificial Intelligence, Edinburgh, 1989.

[Wikström 87] ÅWikström. *Functional Programming Using Standard ML*. Prentice Hall, 1987.