MAX-PLANCK-INSTITUT
FÜR
INFORMATIK

Formal Methods for Automated Program Improvement

Peter Madden

MPI–I–94–238

August 1994

INFORMATIK

Im Stadtwald
66123 Saarbrücken
Germany

Formal Methods for Automated Program Improvement

Peter Madden

MPI–I–94–238 August 1994

Authors' Addresses

Peter Madden
Max-Planck-Institut für Informatik
Im Stadtwald
D-66123 Saarbrücken
Germany
`madden@mpi-sb.mpg.de`

Publication Notes

To appear in the Proceedings of the 18th German Annual Conference on Artificial Intelligence

Acknowledgements

Some of the research reported in this paper was carried out when the author was an SERC Post-Doctoral Research Fellow within the *Mathematical Reasoning Group* at the Department of Artificial Intelligence, Edinburgh University. Acknowledgements are due to Ian Green, Jane Hesketh, Alan Smaill and, in particular, to Alan Bundy.

Abstract

Systems supporting the manipulation of non-trivial program code are complex and are at best semi-automatic. However, formal methods, and in particular theorem proving, are providing a growing foundation of techniques for automatic program development (synthesis, improvement, transformation and verification). In this paper we report on novel research concerning: (1) the exploitation of synthesis proofs for the purposes of automatic program optimization by the transformation of proofs, and; (2) the automatic synthesis of efficient programs from standard equational definitions. A fundamental theme exhibited by our research is that mechanical program construction, whether by direct synthesis or transformation, is tantamount to program verification plus higher-order reasoning. The exploitation of the proofs-as-programs paradigm lends our approach numerous advantages over more traditional approaches to program improvement. For example, we are able to automate the identification of efficient recursive data-types which usually correspond to *eureka* steps in "pure" transformational techniques such as unfold/fold. Furthermore, all transformed, and synthesized, programs are guaranteed correct with respect to their specifications.

1 Introduction

There is a growing interest in the use of formal methods, and in particular automatic theorem proving, for program development (for example, synthesis, improvement, transformation and verification). Systems supporting the manipulation of non-trivial programs are, however, complex and are at best semi-automatic. In this paper we report on novel research concerning the exploitation of synthesis proofs for the purposes of automatic program improvement. The research takes two different, but related, approaches: program improvement by transforming formal synthesis proofs, and; program improvement by synthesizing efficient programs from equational definitions that correspond to less efficient programs. This research has direct applications regarding the improvement of the quality of software produced through automatic programming. Our approach has numerous advantages over more traditional approaches to program optimization which we shall address in the subsequent sections.

Although some of the research has been documented in previous publications, [Mad92, MB93], we have since reconstructed and considerably extended the systems reported there in. This paper represents an up to date account of our research. New features of our research documented within this paper include the use of higher-order variables to delay choices concerning the identification of recursive data-types for optimized programs, and the systemization of meta-level control strategies. We also highlight a common theme that links the transformation and synthesis aspects of the research: namely, that (automatic) program generation can be viewed as a higher-order verification process. Moreover, this is the first time we have covered the broad scope of this research in one paper. Thus, through necessity, we shall describe the research at a fairly high level of abstraction. Whenever appropriate, however, we direct the reader toward various references for such things as extensive examples, implementational details and low level discussions of the proof theoretic properties that apply.

1.1 Formal Methods and Automated Reasoning

A dilemma in the field of computer science is that demands for quality and complexity of software are outstripping the tools currently available. As computer programs play an increasingly important role in all our lives so we must depend more and more on techniques, preferably automatic, for ensuring the high quality (*efficiency* and *reliability*) of computer programs. By *efficient* we mean that a program is designed to compute a task with minimum overhead and with maximum space and time efficiency. By *reliable* we mean that a program is ensured, or guaranteed in some sense, to compute the desired, or specified, task.

The most promising technique being developed for the automatic development of high quality software are *formal methods.* Applications of formal methods in software engineering depend critically on the use of automated theorem provers to provide improved support for the development of safety critical systems. Potentially catastrophic consequences can derive from the failure of computerized systems upon which human lives rely such as medical diagnostic systems, air traffic control systems and defence systems. The failure last year of the computerized system controlling the London Ambulance Service provides an example of how serious software failure can be. Formal methods are used to provide programs with, or prove that programs have, certain properties: a program may be proved to *terminate*; two programs may be proved equivalent; an inefficient program may be *transformed* into an equivalent efficient program; a program may be *verified* to satisfy some specification (i.e. a program is proved to compute the specified function/relation); and a program may be *synthesized* that satisfies some specification.

The research described herein addresses both the reliability and efficiency, as well as the automatability, aspects of developing high quality software using formal methods. We describe novel theorem proving techniques for both automatic program optimization and automatic program synthesis. In both cases the *target* program is a significant improvement on the *source* (the efficiency criteria), and is guaranteed to satisfy the desired program specification (the reliability criteria).[1]

In the remainder of this section we provide some background to the proofs as programs paradigm. §2 concerns the application of formal methods – specifically theorem proving – for automatic, and correctness preserving, program improvement. We first, in §2.1, describe the automatic source to target *transformation* of synthesis proofs yielding inefficient programs into synthesis proofs that yield efficient programs. Secondly, in §2.2, we describe program improvement through the automatic *synthesis* of efficient programs from standard equational definitions

1.2 Background: *Proofs as Programs Paradigm*

Exploiting the *Proofs as Programs Paradigm* for the purposes of program development has already been addressed within the AI community [HS90, CAB+86]. Constructive logic allows us to correlate computation with logical inference. This is because proofs of propositions in such a logic require us to construct objects, such as functions and sets, in a similar way that programs require that actual objects are constructed in the course of computing a procedure. Historically, this correlation is accounted for by the *Curry-Howard isomorphism* which draws a duality between the inference rules and the functional terms of the λ-calculus [CF58, How80].

Such considerations allow us to correlate each proof of a proposition with a specific λ-term, λ-terms with programs, and the proposition with a specification of the program. Hence the task of generating a program is treated as the task of proving a theorem: by performing a proof of a formal specification expressed in constructive logic, stating the *input-output* conditions of the desired program, an algorithm can be routinely extracted from the proof. A program specification can be schematically represented thus:

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

Existential proofs of such specifications must establish (constructively) how, for any input vector, an output can be constructed that satisfies the specification.[2] Thus any synthesized program is guaranteed correct with respect to the specification. Different constructive proofs of the same proposition correspond to different ways of computing that output. By placing certain restrictions on the nature of a synthesis proof we are able to control the efficiency of the target procedure. Thus by controlling the form of the proof we can control the efficiency with which the constructed program computes the specified goal. Here in lies the key to both synthesizing efficient programs, and to transforming proofs that yield inefficient programs into proofs that yield efficient programs.

2 Program Improvement by Formal Methods

This section addresses program improvement by:

[1] The terms *source* and *target* are used throughout this paper and refer to the input and output of the automatic improvement process under discussion. To each source and target proof there corresponds a source and target program. The source may be either manually or automatically constructed, where as the target is always obtained through automatic transformation of the source proof.

[2] Thus constructive logic *excludes* pure existence proofs where the existence of *output* is proved but not identified.

1. the optimization of programs through the transformation of synthesis proofs (§2.1), and;

2. the synthesis of efficient programs, from standard equational definitions, using meta-level proof planning strategies called *proof-plans* (§2.2).

In both case the program improvement is completely automatic. Regarding 1. a source proof, together with any source lemmas, form the input to the system. Regarding 2. the source equational definitions form the input to the system. The output in both cases corresponds to a complete target proof from which an improved program can be routinely extracted. Moreover, the resulting program is guaranteed to satisfy the operational criteria specified in the root node (goal) of the proofs.

In both cases meta-variables are employed to circumvent difficult procedural choices during the program construction process. The two approaches differ in the application of the meta-variables since how they are employed depends on characteristics of the kind of optimization required of the target program. The approaches also differ in the means by which the meta-variables are instantiated. In 1. a source proof is used to create explicit target definitions and then meta-variables are used in the construction of recursive definitions. The instantiation of the meta-variables is aided by further analyses of the induction steps in the source proof. In 2. we use meta-variables to actually formulate both our explicit definition and recursive definitions. The automatic proof planning technique is used to instantiate the meta-variables through higher-order unification.

2.1 Program Optimization by Proof Transformation

The Proof Transformation System, henceforth PTS, has the desirable properties of *automatability*, *correctness* and mechanisms for *reducing the transformation search space*, and various *control mechanisms* for guiding search through that space. As far a the author is aware, the PTS is the only working system that accomplishes automatic program optimization through proof transformation. We summarize the benefits of this approach below.[3]

Knowledge of theorem proving, and in particular automatic proof guidance techniques, can be brought to bear on the transformation task. The proof transformations allow the human synthesizer to produce an elegant source proof, without clouding the theorem proving process with efficiency issues, and then to transform this into an opaque proof that yields an efficient target program.

The proofs are in a sequent calculus and proved within the OYSTER proof refinement system [BvHHS90].[4] OYSTER is a theorem prover for intuitionalist type theory, a higher order, constructive, typed logic based on Martin Löf Type Theory [ML84]. The main benefit of using such a logic is that, recalling §1.2, it combines typing properties with the properties of constructivism, such that we can both correlate the propositions of the λ-calculus with specifications of programs and correlate the proofs of the propositions with how the specification is computed. The main benefit of using a sequent calculus notation, as opposed to that of any of the numerous natural deduction systems, is that at any stage (node) during a proof development, all the dependencies (assumptions and hypotheses) required to complete that proof stage are explicitly presented. This provides an analysis of the calling structure

[3]The only other working system which exploits the proofs as programs paradigm for the purposes of program transformation is a program specialization system developed by C.A.Goad [Goa80]. However, specialization is not optimization but rather adapts a general purpose program according to constraints placed on it's input parameters. The author has reconstructed and extended Goad's system. This work is discussed in [Mad89].

[4]OYSTER is the Edinburgh Prolog implementation, and extension, of NuPRL; version "nu" of the *Proof Refinement Logic* system originally developed at Cornell [CAB+86].

of the programs synthesized. Such analyses are not present within normal program code. To exploit such information usually requires additional, and expensive, mechanisms such as the production and analyses of (symbolic) dependency graphs [Pet84, Chi90].

Synthesis proofs differ from straightforward programs in that more information is formalized in the proof than in the program: a description, or *specification*, of the task being performed; a *verification* of the method; and an account of the *dependencies* between facts involved in the computation. Thus, synthesis proofs represent a *program design record* because they encapsulate the reasoning behind the program design by making explicit the procedural commitments and decisions made by the synthesizer. This extra information means that proofs lend themselves better to *transformation* than programs since one expects that the data relevant to the transformation of algorithms will be different and more extensive than the data needed for simple execution.

A key feature of our approach consists in the transformation of the various induction schemas employed in OYSTER synthesis proofs. Of particular importance to inducing recursion in the extracted algorithm is the employment of *mathematical induction* in the synthesis proofs: to each form of induction employed in the proof there corresponds a dual form of recursion [BM79]. Such dualities offer the user a handle on the type, and efficiency, of recursive behaviour exhibited by the extracted algorithm.

By having a specification present, the PTS ensures that all transformed proofs yield programs that are correct with respect to that specification. Traditional program transformation systems have no such formal specification and this this means there is no immediate means of checking that the target program meets the desired operational criteria. Because of the induction-recursion duality, we can also guarantee that the target will be an optimization of the source program. Thus target programs are guaranteed to compute the input-output relation specified originally for the source, and guaranteed to do so more efficiently.

There are two applications of the PTS corresponding to the way in which inductive proofs are transformed in order to optimize recursive programs: firstly, recursive programs are improved by transforming the induction schemas employed in the source proofs into *logically equivalent* schemas that yield more efficient recursion behaviour[5]. Secondly, whilst retaining the dominant induction, the PTS can improve a program by transforming, or removing, sub-proofs at the corresponding induction cases. We shall consider each application in turn.

- **Transformation of induction schemas:**

source to target transformations of the first kind transform the recursion schemata of source programs. Although the individual syntheses have much in common, in particular the general shape exhibited by the majority of inductive proofs, the main difference between the source and target proofs are the induction schemata employed, and the existential instantiations (witnesses) employed at the induction proof cases. The PTS exploits the induction-recursion duality by transforming a source proof induction schema into a target schema that yields a more efficient recursion schema. To illustrate the process we shall consider the optimization of a program, f, for computing the Fibonacci numbers as a simple example. We consider three types of induction rule that can be employed in proofs of the following

[5]By logically equivalent induction schemas we mean that the associated induction theorems are inter-derivable. This guarantees that any two proofs satisfying the same complete specification but differing only in which of the two schemas employed are *functionally equivalent*.

specification, $\mathcal{S}$, for Fibonacci:[6]

$$\mathcal{S}\colon\ \forall input,\ \exists output.\ f(input) = output$$

In TABLE 1 we show the (uninstantiated) induction schema corresponding to the induction rule employed, where P is some property on natural numbers and all variables are universally quantified (s is the successor, or $+1$, function). Note that such rules are presented upside down, with the goal sequent appearing at the bottom. This reflects the goal-directed nature of the sequent calculus. Also shown are the complexity and recursive data-type of the λ-function constructed through the inductive proof. Finally, we show the left hand side of the function's definition (i.e. the data-type used for computing the Fibonacci numbers) and the right hand side (i.e. the recursion and terminating branches of the definition). The standard definition for Fibonacci appears in the first, course-of-values, column.

induction schema	**course-of-values** $\dfrac{((y<z)\rightarrow P(y))\vdash P(z)}{\vdash P(x)}.$	**stepwise (+1)** $\dfrac{\vdash P(0)\quad P(y)\vdash P(s(y))}{\vdash P(x)}.$	**divide-and-conquer** $\dfrac{\vdash P(0)\quad P(y)\vdash P(y+y)\quad P(z)\vdash P(s(z+z))}{\vdash P(x)}.$
complexity	exponential	linear	logarithmic
data-type	natural number	tuple	matrix
def. LHS	f_n	$\langle f_n, f_{n-1}\rangle$	$\begin{bmatrix} f_{n+1} & f_n \\ f_n & f_{n-1}\end{bmatrix}^n$ (*abbrv. to* $\mathcal{M}$)
def. RHS	$\begin{array}{ll} 1 & n=0; \\ 1 & n=1; \\ f_{n-1}+f_{n-2} & n\geq 2.\end{array}$	$\begin{array}{ll} \langle 1,1\rangle & n=0; \\ \langle f_{n-1}+f_n, f_n\rangle & n\geq 1.\end{array}$	$\begin{array}{ll} 1 & n=0; \\ \mathcal{M}^{n+2}\times\mathcal{M}^{n+2} & even(n); \\ \mathcal{M}^{n+2}\times\mathcal{M}^{n+2}\times\mathcal{M} & odd(n).\end{array}$

TABLE 1: RELATION BETWEEN SOURCE AND TARGET PROOFS AND FUNCTIONS

Program optimization through proof transformation consists in transforming a source induction proof to a target proof whose induction schema has a more efficient associated complexity. The pre- and post-conditions of the transformation correspond to the induction schema, and the recursive data-type, of the source and target proofs. Thus, the post-conditions of the exponential to linear transformation are precisely the pre-conditions for the linear to logarithmic transformation. This illustrates how, by "dove-tailing" each of the source to target transformations, depicted in TABLE 1, the passage from an exponential procedure to a logarithmic one, with linearization as an intermediary optimization, is performed automatically through proof transformation (and with the correctness guarantee afforded by the specification language).

Although uniform in strategy, individual inductive proofs will usually differ regarding the following procedural commitments made during the synthesis component of a proof: (**i**) the choice of induction schema employed; (**ii**) the type of object introduced at the induction step (e.g. natural number, list, tuple), and; (**iii**) the witness (existential instantiation) of the object. These commitments are responsible for constructing the recursion schemata and the recursive data types of the target procedures. By incorporating general rules that associate (**i**) and (**ii**) with the kind of recursive behaviour desired of the target algorithm, and analysing the definitions of, and dependency information in, the source proofs to identify (**iii**), the PTS is able to construct the target proofs automatically.

We shall use the *linearization* of the Fibonacci function for the purposes of explanation (i.e. the transformation of course of values induction to stepwise induction). The PTS linearization procedure is, in fact, our adaptation to the proofs as programs paradigm of the unfold/fold *tupling* technique for "merging" repeated (sub)computations, [BD77, Chi90].

[6] The Fibonacci function, f is initially defined through lemmata corresponding to the course-of-values definition given in TABLE 1, column 1:

base lemmas: $fib(0) = s(0)$; $fib(s(0)) = s(0)$;

step lemma: $\forall x, \exists y, \exists z.\big((x \neq 0) \wedge (x \neq s(0)) \wedge fib(s(x)) = y \wedge fib(x) = z\big) \rightarrow fib(s(s(x))) = y + z.$

The most natural way to synthesize a procedure for computing the Fibonacci numbers is to employ the course-of-values induction to $\mathcal{S}$. This is because it directly mirrors the course-of-values recursion exhibited by the standard Fibonacci definition. The corresponding schema, TABLE 1, will be instantiated as follows:

$$\frac{H:\ (\forall z, \forall y.((y < z) \to \exists n'.f(y) = n') \vdash \exists n''.f(z) = n''}{C: \vdash \forall x, \exists n.f(x) = n}.$$

The proof of the induction conclusion, C, requires identifying a witness for n. This is obtained by:

- eliminating on the induction hypothesis, H, twice:[7] first with a value for y of $x-1$, and subsequently with a value of $x-2$. The resulting constructs for f_{x-1} and f_{x-2} appear as two new hypotheses; and
- adding the new hypotheses to obtain a witness for n.
  i.e. $\vdash \forall x, f(x) = f(x-1) + f(x-2)$

By employing course-of-values induction, and eliminating on the hypothesis twice, we obtain a program such that in order to calculate $fib(n)$ one must first calculate $fib(n-1)$ and $fib(n-2)$. Each of these sub-goals leads to another two recursive calls on fib and so on. In short the computational tree is exponential where the number of recursive calls on fib approaches 2^n. The automatic linearization of such procedures involves constructing a target tuple (as shown in TABLE 1) whose elements act as accumulating parameters. The accumulators are used to build up the output as the recursion is entered, so that nothing remains to be done as the recursion exits. Thus 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.

In order to identify a target (tuple) definition, the PTS observes how many times the induction conclusion C appeals to the hypothesis H, and how many applications, namely 2, of the induction constructor/destructor function the proof employs when eliminating on the induction hypothesis in order to synthesize constructs for the induction witnesses. This completely identifies an *explicit* definition, $\mathcal{G}$, for the auxiliary recursive procedure through which Fibonacci can be defined:[8]

$$\mathcal{G}:\quad \forall n, \exists u, \exists v.g(n) \;=\; \langle u, v \rangle, \quad \text{where} \quad \langle u, v \rangle \;=\; \langle f(n+1), f(n) \rangle.$$

The provision of such explicit definitions, where the target is defined in terms of the source, generally constitute the well known *eureka* step in unfold/fold transformations, and are notoriously difficult to automate [BD77]. The unfold/fold strategy is motivated by the observation that significant optimization of a (declarative) program generally implies the use of a new recursion schema. This process usually depends on the *user* providing the requisite explicit target definition (in our example: $\mathcal{G}$). The strategy then proceeds to evaluate the recursive branches of the target definition, primarily through unfolding with the source definitions, until a fold (match) can be found with the explicit definition.

Within the context of proof transformation, the PTS exploit the source proof to automatically form such definitions. Every new hypothesis formed as result of eliminating on previous hypotheses is recorded in the sequent calculus proof notation. This provides the kind of information usually associated with symbolic dependency

[7] A feature of the goal-directed proofs is that elimination rules have the effect of introducing an existential instantiation in the hypotheses of sequents.

[8] In practice, tuples are represented as conjunctions within the OYSTER system. So a tuple $\langle A, B, C \rangle$ is represented as $A \wedge B \wedge C$. Hence we avoid the charge that (program) tupling techniques rely heavily on any ad hoc requirements to introduce tuples (memo tables or similar objects).

graphs and used, for example, in the semi-automatic construction of unfold/fold tuple definitions [Chi90, Pet84]. The PTS constructs the explicit definitions completely automatically without appealing to the user, or requiring the considerable over-head required in the formation and analyses of dependency graphs.

The explicit definition, $\mathcal{G}$, is automatically applied as a sub-goal of $\mathcal{S}$. This will produce, in addition to $\mathcal{G}$, a trivial justification sub-goal that the function specified by $\mathcal{S}$ can be constructed from that specified by the sub-goal: the variable *output* in $\mathcal{S}$ is witnessed by v from the body of the $\mathcal{G}$. In effect, the justification sub-proof provides us with a definition for Fibonacci in terms of the auxiliary function g:

$$f(n) = v \quad \text{where} \quad g_n = \langle \_, v \rangle$$

Thus, an advantage of using specification proofs is that at the target proofs completion the PTS ensures that the auxiliary program, corresponding to the sub-proof of $\mathcal{G}$, computes the function specified by the $\mathcal{G}$, *and* by performing the justification goal we ensure that the complete program construction, corresponding to the whole proof, computes $\mathcal{S}$.

To synthesize the auxiliary function, g, the PTS applies *stepwise*, or $+1$, induction to $\mathcal{G}$ so as to construct the dual stepwise recursion. In other words, the PTS constructs a recursive definition through an inductive proof of the explicit definition $\mathcal{G}$. The base case, $g(0)$, evaluates to $\langle 1, 1 \rangle$ by using symbolic evaluation with the base definitions for Fibonacci. At the step case of the induction the PTS is required to provide a definition for the recursive step in terms of the hypothesis (i.e. $g(n+1)$ in terms of $g(n)$). A characterizing feature of such tupling proofs is that the recursive definition will consist of some, as of yet unknown, function(s) applied to the tuple components, u and v, of the induction hypothesis. Hence , using upper-case to represent meta-variables, at the induction step of the target proof, the PTS formulates a partially identified definition for the recursive step of g in terms of the hypothesis (we omit the quantifiers for the remainder of this section):

$$g(n+1) \; = \; \langle M_1(u,v), M_2(u,v) \rangle, \;\; \textit{where} \;\; \langle u,v \rangle \; = \; g(n).$$

The induction step proof then proceeds as in FIG 1 until, by a process of unfolding with the source and target definitions, all references to the source function, f, have been removed from the developing target recursive branch (essential if we wish to eliminate the source inefficiency). Once this stage has been reached, the PTS *could* use higher-order unification to instantiate M_1 to $\lambda u, v. u+v$, and M_2 to $\lambda u, v \,.\, u$.

$$\begin{array}{rcl}
g(n+1) & = & \langle M_1(u,v), M_2(u,v) \rangle, \;\; \textit{where} \;\; \underbrace{\langle u,v \rangle \; = \; g(n)}_{\textbf{source} \mapsto \textbf{target}}; \\
\textbf{unfold } g & \text{and} & \textbf{unfold } g; \\
\langle f(n+2), f(n+1) \rangle & = & \langle M_1(u,v), M_2(u,v) \rangle, \;\; \textit{where} \;\; \underbrace{\langle u,v \rangle \; = \; \langle f(n+1), f(n) \rangle}_{\textbf{source} \mapsto \textbf{target}}; \\
\textbf{unfold } f & & \\
\langle (f(n+1) + f(n)), f(n+1) \rangle & = & \langle M_1(u,v), M_2(u,v) \rangle, \;\; \textit{where} \;\; \langle u,v \rangle \; = \; \langle f(n+1), f(n) \rangle; \\
\textbf{fertilize } (u/f(n+1), \; v/f(n)) & & \\
\langle u+v, u \rangle & = & \langle M_1(u,v), M_2(u,v) \rangle; \\
\textbf{instantiation} & & M_1 \; = \; \lambda u, v. u + v \;\text{ and }\; M_2 \; = \; \lambda u, v \,.\, u.
\end{array}$$

$\underbrace{rhs = lhs}_{\text{source} \mapsto \text{target}}$ **signifies that $rhs = lhs$ is obtained by analysis of source to identify: (1) the tuple size, and; (2) the constituent data structures.**

FIG 1: SYNTHESIS COMPONENT OF TARGET PROOF CONSTRUCTION (INDUCTION STEP ONLY)

However, the PTS avoids any need for higher-order matching, and the associated control problems, by providing general *mapping mechanisms* which abstract information from the induction branches of the source proof. This information is then used to instantiate the target meta-variables. The general mapping mechanisms are described in detail in [Mad91].

Regarding our current example, the procedure is quite simple: the first component of the r.h.s tuple (corresponding to the induction conclusion) results from *substituting* the *target* induction hypothesis tuple components, u and v, for those in the *source* induction step. Hence the first component is $u + v$ (i.e. M_1 is instantiated as $\lambda u, v.u + v$). No higher-order matching, or unification, procedures are required since the dominant function of the first tuple component will always be that employed at the induction step of the source (where the number of tuple elements corresponds to the number of source proof eliminations on the induction hypothesis). The second component results from a direct one on one mapping of the first component, u, of the *target* induction hypothesis (i.e. $M_2 = \lambda u, v \,.\, u$).

The λ program construction extracted from the target proof is shown below (we have necessarily simplified the notation).

$$\lambda x.(\lambda tuple.(sub(\langle u, v\rangle, [\sim, x, x])).(step_{+1}(x, \langle s(0), s(0)\rangle, [\bar{x}, \langle u, v\rangle, \langle u+v, u\rangle])))$$

The solution for *Fibonacci* corresponds to v in the extract (i.e., the second argument of the first tuple component). The *substitution* function, *sub*, substitutes the second element of the tuple (the desired output v) for x in the root node specification. The $step_{+1}$ function, corresponding to the application of stepwise induction, will automatically build the dual recursion schema into the extract term being synthesized. The application of the $step_{+1}$ induction constructs a triple where the first member, x, names the induction candidate: the argument over which the recursion is defined. The second member, $\langle s(0), s(0)\rangle$, corresponds to the construction of the base case output. The third member is a further triple where $\bar{x}$ denotes the induction variable, and $\langle u, v\rangle$ denotes the constructive evidence for the induction hypothesis. The induction conclusion, $\langle u + v, u\rangle$, is composed from the elements, u and v, of the hypothesis.

The form, or shape, of refinement proofs means that *folding* is not a necessary requirement in order to introduce a recursion into the developing equations. This is because the proof synthesis is driven by the heuristic requirement of matching induction hypothesis with induction conclusion, i.e., *fertilization* (we shall say more concerning fertilization in §2.2). This can be achieved purely by unfolding both sides of the induction step until both head and body match (*cf.* FIG 1). By unfolding terms on *both* sides of the induction conclusion we gradually remove the induction term from the conclusion. This bi-directional rewriting has advantages over the more traditional program derivations, such as [BD77, Chi90], wherein re-writing is restricted to the body of the equations: most notably, we avoid the control problems of directing sequences of unfolds toward a fold. The bi-directional search toward the fertilization step significantly reduces the search space.

The verification component of the proof will mirror the synthesis component:

$$\begin{array}{rcl}
g(n+1) & = & \langle u+v, u\rangle, \ \ \textit{where}\ \langle u, v\rangle = g(n); \\
\textbf{unfold}\, g & \text{and} & \textbf{unfold}\, g; \\
\langle f(n+2), f(n+1)\rangle & = & \langle u+v, u\rangle, \ \ \textit{where}\ \langle u, v\rangle = \langle f(n+1), f(n)\rangle; \\
\textbf{unfold}\, f & \text{and} & \textbf{fertilize}\ (u/f(n+1), v/f(n)); \\
\langle (f(n+1)+f(n)), f(n+1)\rangle & = & \langle (f(n+1)+f(n)), f(n+1)\rangle.
\end{array}$$

FIG 2: VERIFICATION COMPONENT OF TARGET PROOF CONSTRUCTION

An observation is that the essential difference between the synthesis (FIG 1) and verification (FIG 2) components of the target construction is that the former uses meta-variables (simply compare the first lines of each figure). A common theme of our work within the proofs as programs paradigm is that program synthesis/transformation is tantamount to program verification plus meta-variables. That is, we recast first-order synthesis proofs as higher-order verification proofs, and in doing so circumvent eureka steps concerning the identification of recursive data-types. We see a further illustration of this theme in §2.2.

The verification strategy of induction proofs invariably follows the same procedure of applying refinement rules that consist primarily of unfolding the recursive branches with the equational definitions that define the function computed by the extract program. Indeed, the induction strategy is uniform enough to be systemized as a meta-level proof plan, with pre- and post-conditions. So once the PTS has automatically made the target proof procedural commitments corresponding to (i) - (iii), by an analysis of the source proof, then verification is automatically performed through the use of an induction proof plan. We shall return to proof plans in more detail in §2.2.

A much more in-depth description of the processes involved, and of the implementation, are provided in [Mad91].

As indicated in TABLE 1, the PTS can automatically optimize linear procedures to logarithmic procedures through proof transformation. This is done using the method of *matrix multiplication* and replacing the *stepwise* induction employed in the source proof by a target *divide-and-conquer* induction. Full details are available in [Mad94].

- **Transformation of induction cases:**

transformations on induction cases correspond to transforming the sub-proofs of the base and /or step case sub-goals *without* altering the particular schema for which the sub-goals are cases. Different recursive behaviour can be induced in algorithms, satisfying the same specification, by refining the step and base cases of the *same* schema in different ways. One important class of source to target proof transformation on induction cases with which we are concerned is the transformation of *nested* inductions. Nested inductions are often employed when synthesizing *auxiliary recursive functions*, that is, functions which in computing a self-recursive call must appeal to some other function, either directly or indirectly.[9]

A nested induction may lead to inefficiency since for each of the recursive passes induced by the outermost induction, the program will have to fully recurse on the innermost recursive schema induced by the innermost induction. Thus the average time efficiency of such programs will be a multiple of the time efficiencys associated with the two inductions. So for example, the recursive definition of the following schematic function f_1:

$$f_1(n) \quad = \quad f_2(f_3(n), f_1(n-1))$$

contains both an auxiliary function call, $f_3(n)$, and a self-recursive call, $f_1(n-1)$. Each time a recursive call is made on f_1, the program must fully recurse down the schema associated with f_3. Proofs wherein a nested (stepwise) induction is applied at the step case of the outermost induction may, for example, yield a λ program construction of the following (schematic) form:

$$\lambda x.\ step_{+1}(x, \phi_0, [\bar{x}, \phi_{H_{ind}}, \lambda\bar{x}.\ step_{+1}(\bar{x}, \phi_0, [\bar{\bar{x}}', \phi'_{H_{ind}}, \phi'_C])]),$$

where in order to evaluate the step case of the outermost +1 stepwise induction on

[9]The nesting may be indirect since the structure introduced for the step case of an induction may well correspond to the application of an extract term from another proof which itself employs stepwise/list induction.

x, with induction parameter $\bar{x}$, the program must evaluate a nested induction on $\bar{x}$. The terms $\phi_0, \phi_{H_{ind}}, \phi_C$ denote, respectively, the induction base, hypothesis and conclusion constructs. A prime, $'$, demarcates the nested induction constructs from those of the outer induction. Optimizations on such extract terms are performed through "merging" the innermost induction with the outermost induction. This is achieved by removing the innermost induction and introducing a tuple structure at the cases of the remaining induction. This yields a target construction of the following schematic form:

$$\lambda x.\lambda tuple.\ step_{+1}(x, \langle \phi_0^1, ..., \phi_0^n \rangle, [\bar{x}, \langle \phi_{H_{ind}}^1, ..., \phi_{H_{ind}}^n \rangle, \langle \phi_C^1, ..., \phi_C^n \rangle])$$

where there is a single stepwise induction, on the same variable x. In this case, the induction schema cases are satisfied through the evaluation of a tuple, of fixed size n, at the base and step branches. In effect the remaining induction tabulates the computation associated with the innermost induction removed from the source.

As with application 1, the PTS analyses the dependency information in the source proofs to obtain explicit definitions and to instantiate meta-variables [Mad91]. For example, the auxiliary recursive *factlist* function, defined in FIG 3, is synthesized through a nested inductive proof, at the step case of the outer most induction, in order to to synthesize the auxiliary function *fact*:

factlist **definition**	*fact* **definition**
$factlist_n = \begin{cases} nil & n = 0; \\ fact_n :: factlist_{n-1} & n \geq 1. \end{cases}$	$fact_n = \begin{cases} 1 & n = 0; \\ n \times fact_{n-1} & n \geq 1. \end{cases}$

FIG 3: DEFINITIONS OF *factlist* AND AUXILIARY *fact*

Here redundancy does not occur directly due to any self-recursive call but rather among the auxiliary recursive *fact* calls: for each recursive pass corresponding to the outermost schema, the function must fully recurse on the innermost schema.

The corresponding target definition is defined through a tuple function, g', which merges the auxiliary *fact* recursion schema with that of the outermost schema (details of this process are provided in [Mad91]). As with the previous example, the PTS avoids any eureka requirements by automatically forming the explicit target definition, $\mathcal{G}'$, through an analysis of the source proof (in this case, by analysing the nested induction construct).

$$\mathcal{G}' : \ \forall n, \exists u, \exists v. g(n) \ = \ \langle u, v \rangle, \quad where \quad \langle u, v \rangle \ = \ \langle fact(n), factlist(n-1) \rangle.$$

Stepwise induction is applied to $\mathcal{G}'$ and, as we show in FIG 4, the proof proceeds in a similar fashion to the previous example: the PTS uses meta-variables to specify that the recursive case is some function of the induction hypothesis. The induction conclusion is then refined, by unfolding with $\mathcal{G}'$ and the source definitions, until instantiations abstracted from the source proof enable the two sides of the conclusion to match:

$$\begin{array}{rcl}
g'(n+1) & = & \langle M_1(u,v), M_2(u,v) \rangle \ where \ \langle u,v \rangle = g'(n); \\
\textbf{unfold } g' & \text{and} & \textbf{unfold } g'; \\
\langle fc(n+1), fcl(n) \rangle & = & \langle M_1(u\ ,\ v), M_2(u\ ,\ v) \rangle, \ where \ \langle u, v \rangle = \langle fc(n), fcl(n-1) \rangle; \\
\textbf{unfold } fc \ \ \textbf{unfold } fcl & & \\
\langle (n+1) \times fc(n), fcl(n-1) :: fc(n) \rangle & = & \langle M_1(u\ ,\ v), M_2(u\ ,\ v) \rangle, \ where \ \langle u, v \rangle = \langle fc(n), fcl(n-1) \rangle; \\
\textbf{fertilize } (u/fc(n),\ v/fcl(n)) & & \\
\langle (n+1) \times u, u :: v \rangle & = & \langle M_1(u\ ,\ v), M_2(u\ ,\ v) \rangle; \\
\textbf{instantiation} & & M_1 \ = \ \lambda n, u.(n+1) \times u \ \text{ and } M_2 \ = \ \lambda u, v\ .\ u :: v
\end{array}$$

FIG 4: SYNTHESIS COMPONENT OF TARGET PROOF CONSTRUCTION (INDUCTION STEP ONLY)

Analysis of the proof provides the target recursive definition:

$$g'_n = \begin{cases} \langle nil, nil\rangle & n = 0; \\ \langle (n+1)\times u, u :: v\rangle & n \geq 1. \end{cases} \quad \text{and;} \quad factlist_n = m, \quad where \quad \langle \_, m\rangle = g'_n.$$

The PTS is capable of combining the different kinds of induction transformation such that, for example, course-of-values (exponential) definitions that employ auxiliary functions can be optimized to stepwise (linear) definitions with a single induction (single recursion schema). The PTS is also capable of performing more esoteric induction transformations involving schemas such as divide-and-conquer induction (*cf.* TABLE 1), two step induction, and induction based on the construction of numbers as products of primes [Mad91].

2.2 Program Optimization by Proof Synthesis

The Mathematical Reasoning Group, at the Edinburgh University Department of AI, has achieved considerable success regarding the automation of inductive theorem proving using a meta-level control paradigm called 'proof planning' [Bun88]. A proof planning system, CLAM, is able to prove a large number of inductive theorems automatically [BvHS91]. Proof plans are formal outlines of constructive proofs and provide a means for expressing, in a meta-language, the common patterns that define a family of proofs [BSvH$^+$91, MHGB93]. A tactic expresses the structure of a proof strategy at the level of the inference rules of the object-level logic. Proof plans are constructed from the tactic specifications called *methods*. Using a meta-logic, a method captures explicitly the preconditions under which a tactic is applicable.

This section reports on the most recent application of CLAM: the use of proof plans to control the (automatic) synthesis of efficient functional programs, specified in a standard equational form, $\mathcal{E}$, by using the proofs as programs principle [MG94]. The goal is that the program extracted from a constructive proof of the specification is an optimization of that defined solely by $\mathcal{E}$. Thus the theorem proving process is a form of program optimization allowing for the construction of an efficient, *target*, program from the definition of an inefficient, *source*, program. Our main concern lies with optimizing the recursive behaviour of programs through the use of proof plans for inductive proofs. Thus again we exploit the duality between induction and recursion (which forms one aspect of the *Curry-Howard isomorphism*).

The main difference from the proof transformation approach to program improvement, §2.1, is that there is no source proof to guide the construction of the target (only source definitional equations). Hence this is a process of synthesis rather than transformation. We again employ meta-variables, except in this case a proof planning technique called *middle-out reasoning* is used to instantiate them through higher-order reasoning. In fact, we view program synthesis as the combination of verification and middle-out reasoning [Mad93]. Middle-out reasoning is a technique that allows us to solve the typical eureka problems arising during the synthesis of efficient programs by allowing the planning to proceed even though certain object-level objects are unknown (e.g. identification of induction schema, recursive types etc.) Subsequent planning then provides the necessary information which, together with the original definitional equations, allows for the instantiation of such meta-variables through higher-order unification procedures.[10]

[10] *λ-Prolog*, [MN88], is used for the higher-order unifications and has been interfaced with the CLAM system. An indefinite number of unifications may be produced by such an algorithm if no means of selecting suitable choices is present. Details of such selection criteria, and of the algorithm itself, are provided in [Hes91].

A further property of program improvement through proof planning is that the nature of the optimization is controlled by placing certain restrictions on the proof. We illustrate one such restriction in the following example.

A Simple Example

To illustrate program improvement by proof planning we use a simple example of the synthesis of tail-recursive programs from naïve definitions using the *tail-recursive proof plan* (TRPP) [MHGB93].[11]

Consider the following example of a *naïve*, $length_n$, and a *tail recursive*, $length_t$, definition for the *length* function, where $length_t$ is defined in terms of the auxiliary accumulator function $length_2$:

naïve definition	**tail recursive definition**
$length_n(nil) = 0;$ $length_n(h :: t) = 1 + length_n(t).$	$length_t(l) = length_2(l, 0)$ $length_2(nil, a) = a$ $length_2(h :: t, a) = length_2(t, 1 + a)$

TABLE 2: NAÏVE DEFINITION AND TAIL RECURSIVE DEFINITIONS FOR *length*

The key initial step to the TRPP is to provide an *explicit* target definition by specifying the *tail recursive* algorithm in terms of the *naïve* algorithm. This is depicted on the left hand side of TABLE 3 where we show the schematic form of such explicit definitions, together with the particular instance, (1), for our current example: the specification goal for synthesizing $length_t$. There is no eureka involved with forming such explicit definitions: for any function, f, under consideration all that is stated is that for any input, x, and any additional vectors, $\vec{y}$, that there exists some output, z, equal to $\lambda x.f(n)$.[12]

	preconditions	**postconditions**
schematic	$\forall x, \forall \vec{y}, \exists z.\ z = f_n(x, \vec{y})$	$\vdash \forall x, \forall \vec{y}, \forall a, \exists z.\ z = G(f_n(x, \vec{y}), a)$
instance	$\vdash \forall x, \exists z.\ z = length_n(x) \quad (1)$	$\vdash \forall x, \forall a, \exists z.\ z = G(length_n(x), a) \quad (2)$

TABLE 3: PRE- AND POST- CONDITIONS OF ACCUMULATOR GENERALIZATION

The next stage is for the TRPP to introduce an accumulator into the function being synthesized through a generalization procedure. This is done by making a call to a (sub)proof plan for *accumulator generalization*. The pre-condition of accumulator generalization is satisfied by a schematic equational form of which (1) is an instance. The post-condition of the generalization is shown on the right hand side: the pre-conditional form is *generalized* through the introduction of a meta-variable M to produce the post-conditional form. In the majority of transformation systems the identification of M is a *eureka* step. The tail-recursive generalization strategy removes the *eureka* step and identifies M automatically through *middle-out* reasoning. In the case of $length_t$ the post-condition is (2) and we shall show how M is automatically instantiated as $\lambda u \lambda v.append(u, v)$ through the proof of (2).

Having applied generalization, the proof of (1) is now cashed out in terms of:

- a *synthesis* goal, proving (2), and;

[11] The examplified application of the TRPP is adequate to illustrate the middle-out reasoning methodology. However, since it's original implementation, documented in [MHGB93], it has been extended to cover a broader range of optimizations such as deforestation and fusion [MG94].

[12] The trivial instantiation of z, i.e. $f(x)$, is prevented by placing a restriction on the proof that f is not a term of the witness for z.

- a *justification* goal, proving that (2) $\vdash$ (1). (i.e. that the new (tuple) goal entails the original theorem.)

The justification goal is, in fact, a second post-condition of the TRPP and establishes that the new (tuple) goal entails the original theorem.[13] So the justification goal will, in effect, provide us with a definition of the tail-recursive function, $length_t$, in terms of the auxiliary accumulator function, $length_2$, synthesized via the synthesis goal.

A characterizing feature of the TRPP is the restriction that the witnesses of the two existential quantifiers, one in the induction hypothesis and one in the induction conclusion, should be identical. This restriction ensures that we force the value of the function before the recursion is entered (determined by the induction hypothesis) to be the same as the value as the recursive call is exited (determined by the induction conclusion) — i.e. the function synthesized is tail recursive [Wai89].

To continue the illustration we must explain a bare minimum of technical terminology. As mentioned in §2.1, the verification stages of an inductive proof invariably involve a process whereby formulae are "unpacked" - or *unfolded* - by replacing terms by suitably instantiated definitions. The proliferation of this process such that recursive terms are gradually removed from the recursive branches — by the repeated unfolding of induction terms — is part of the (heuristic) process known as *rippling* (following [Aub75]). The goal of the rippling proof-plan is to reduce the induction step case to terms which can be unified with those in the induction hypothesis. As mentioned in §2.1, this unification is called *fertilization*. Fertilization is facilitated by the fact that the induction conclusion is structurally very similar to the induction hypothesis except for those function symbols which surround the induction variable in the conclusion. These points of difference are called *wave-fronts*. Thus, the remainder of the induction conclusion - the *skeleton* - is an exact copy of the hypothesis. Wave fronts consist of expressions with holes — *wave holes* — in them corresponding to sub-terms in the skeleton. Returning to our example, having generalized the specification goal, (1), to produce the higher-order sequent (2), the TRPP must next satisfy the synthesis goal of proving (2). This is done by applying the *induction* proof plan, again as a sub-proof plan, to produce the **step case** sequent (3). By convention, wave-fronts are annotated by placing them in boxes, and the wave-holes are underlined:[14]

$$\forall a, \exists z.\ z = M(length_n(t), a) \ \vdash\ \forall a, \exists z.\ z = M(\boxed{1 + \underline{length_n(t)}}, a) \qquad (3)$$

Rippling applies special structural rewrite rules, *wave-rules*, so as to remove the difference (wave-fronts) from the conclusion, thus leaving behind the skeleton and allowing fertilization to take place. Wave rules are schematic rewrites — hence they employ meta-variables — and are *automatically* formed, by the proof planning mechanism, from recursive definitions and semantic laws. For example, the wave rule below is formed from the recursive branch of $length_n$:

$$length_n(\boxed{X :: \underline{Y}}) \ \Rightarrow\ \boxed{1 + \underline{length_n(t)}} \qquad (4)$$

So by matching and then applying the current sequent against the available (wave) rules, the meta-variable in (3) is instantiated at the step case of the induction. Briefly, after rippling on (3), using wave-rule (4), we obtain:

$$\forall a, \exists z.\ z = M(length_n(t), a) \ \vdash\ \forall a, \exists z.\ z = M(\boxed{1 + \underline{length_n(t)}}, a) \qquad (5)$$

By using higher-order matching with the applicable wave-rules, the meta-variable M is instantiated to $\lambda x, y. x + y$ through applying the following wave-rule formed

[13]This is directly analogous to where a justification sub-goal is produced during the PTS transformation of a source proof by cutting in a new sub-goal specifying a target definition (*cf.* §2.1).

[14]There are additional annotations which further direct the rippling process. We omit these in this paper. For full details *cf.* [BSvH+91].

from the law of commutitivity of $+$:

$$(\boxed{U+\underline{V}})+W \quad\Rightarrow\quad V+\boxed{U+(\underline{W})}$$

i.e. the result of the applying wave rule (6) to (5) is the following sequent:

$$\forall a, \exists z.\ z = \mathit{length}_n(t) + a \quad \vdash \quad \forall a, \exists z.\ z = \mathit{length}_n(t) + \boxed{1+\underline{a}} \qquad (6)$$

Fertilisation can now take place: a in the induction hypothesis is instantiated to $1+a$ from the induction conclusion. Such an instantiation effects our tail-recursive restriction: that the witnesses of the two existential quantifiers should be identical. This step is perfectly legitimate since the a's on either side of the sequent are bounded by *distinct* universal quantifiers.

The **base case** sequent is as follows:

$$\vdash\ \forall a, \exists z.\ z \ = \ M(\mathit{length}_n(\mathit{nil}), a) \qquad (7)$$

M is instantiated to $\lambda x, y.x+y$, from the step case, and following symbolic evaluation, using the base definitions of $+$ and length_n, (7) is refined to:

$$\vdash\ \forall a, \exists z.\ z \ = \ a \qquad (8)$$

A further, and very simple, application of middle-out reasoning strips the universal quantifier and introduces a meta-variable for the base witness, which is subsequently instantiated to a by tautology.

Analysis of the proof so far, specifically (6) and (8), provides the recursive and terminating branches of the auxiliary length_2 function (*rhs* of TABLE 2). However, the TRPP must still satisfy the justification obligation. The justification proof will provide a definition for length_t in terms of length_2, and requires proving the subgoal:

$$\forall x, \forall a, \exists z.\ z = \mathit{length}_n(x) + a \quad \vdash \quad \forall x, \exists z.\ z = \mathit{length}_n(x) \qquad (9)$$

The solution to (9) again involves middle-out reasoning: briefly, any universal quantified variables in the conclusion are identified with counterparts in the hypothesis – in this case x. Following the introduction of the universally quantified variables, x, a meta-variable A is inserted for a into the hypothesis:

$$\forall x, \exists z.\ z \ = \ \mathit{length}_n(x) + A \quad \vdash \quad \forall x, \exists z.\ z \ = \ \mathit{length}_n(x) \qquad (10)$$

Symbolic evaluation instantiates the meta-variable to 0, and the resulting hypothesis can be matched (fertilized) with the conclusion. This completes the proof. The witness 0, for A, provides the definition of length_t in terms of length_2 (i.e. $\mathit{length}_t(l) \ = \ \mathit{length}_2(l, 0)$).

Rippling has numerous desirable properties. A high degree of control is achieved for applying the rewrites since the wave-fronts in the rule schemas must correspond to those in the instance. This leads to a very low search branching rate which, together with the further search reduction afforded by the proof restrictions, enables the automation of the synthesis process. Rippling is guaranteed to terminate since wave-front movement is always propagated in a desired direction toward some end state (a formal proof of this property is presented in [BSvH+91]).

2.2.1 Further Applications

We have found this approach to program improvement to be very promising. This is demonstrated by the success we have had in expressing, in addition to tail-recursive transformation, a wide variety of well-known, but disparate, program improvement techniques within the proof planning framework. For example, constraint-based transformation, generalization, fusion and tupling can be seen as proof planning [MHGB93]. Each of these types of optimization have characteristic features which are used to determine how higher-order meta-variables are employed in the specification goals. These applications also extend the simple tail-recursive syntheses in that the wave-rules themselves may contain second-order meta-variables as well as the sequents to which they apply.

We have also extended the work of Wadler, [Wad88] and later Chin, [Chi90], to encompass a larger class of functions that can be usefully optimized using the influential *deforestation technique* [MG94].

We believe that middle-out reasoning will play an increasingly important role in theorem proving, since it allows us to address important problems like choosing induction schemata,[15] and existential witnesses (which correlate with recursion schemata and recursive data types).

3 Conclusion

We have illustrated the fact that formal methods in general, and theorem proving in particular, provide a foundation for automated reasoning. We have described two novel implemented techniques for the automatic generation of high quality (efficient and reliable) software using the proofs as programs paradigm. Program improvement by transformation is achieved through the transformation of typed proofs in a constructive logic. The synthesis of efficient programs from standard equational definitions is achieved through the use of (meta-level) proof-planning techniques. A common theme of the research is the maxim that program construction can be automated through higher-order verification proofs.

Well known eureka steps concerning the identification of target definitions are circumvented by: in the transformational approach, using higher-order meta-variables and extracting information from source proofs in order to instantaite them, and; in the synthesis approach, using meta-variables in a technique called middle-out reasoning to delay procedural commitments until subsequent theorem proving provides the requisite instantiations. In both case the meta-variables areemployed according to characteristics of the type of optimization desired.

Both the transformation and synthesis techniques satisfy the desirable properties for automatic programming systems: correctness, generality, automatability and the means to guide search through the transformation space.

References

[Aub75] R. Aubin. Some generalization heuristics in proofs by induction. In G. Huet and G. Kahn, editors, *Actes du Colloque Construction: Amélioration et vérification de Programmes.* Institut de recherche d'informatique et d'automatique, 1975.

[BD77] R.M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the Association for Computing Machinery*, 24(1):44–67, 1977.

[BM79] R.S. Boyer and J.S. Moore. *A Computational Logic.* Academic Press, 1979. ACM monograph series.

[BSvH+91] A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill. Rippling: A heuristic for guiding inductive proofs. Research Paper

[15] We did not address this application of middle-out reasoning in this paper. Meta-variables can be used to delay the choice of induction schema as well as, or instead of, the choice of existential witnesses [KBB93].

567, Dept. of Artificial Intelligence, Edinburgh, 1991. In the Journal of Artificial Intelligence.

[Bun88] A. Bundy. The Use of Explicit Plans to Guide Inductive Proofs. In R. Luck and R. Overbeek, editors, *CADE9.* Springer-Verlag, 1988.

[BvHHS90] 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.

[BvHS91] 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.

[CAB$^+$86] R.L. Constable, S.F. Allen, H.M. Bromley, et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice Hall, 1986.

[CF58] H.B. Curry and R. Feys. *Combinatory Logic.* North-Holland, 1958.

[Chi90] W.N. Chin. *Automatic Methods for Program Transformation.* PhD thesis, Imperial College, 1990.

[Goa80] C.A Goad. Proofs as descriptions of computation. In W. Bibel and R. Kowalski, editors, *Proc. of the Fifth International Conference on Automated Deduction*, pages 39–52, Les Arcs, France, July 1980. Springer Verlag. Lecture Notes in Computer Science No. 87.

[Hes91] J.T. Hesketh. *Using Middle-Out Reasoning to Guide Inductive Theorem Proving.* PhD thesis, University of Edinburgh, 1991.

[How80] W.A. Howard. The formulae-as-types notion of construction. In J.P. Seldin and J.R. Hindley, editors, *To H.B. Curry; Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, 1980.

[HS90] C. Horn and A. Smaill. Theorem proving with Oyster. Research Paper 505, Dept. of Artificial Intelligence, Edinburgh, 1990. To appear in Procs IMA Unified Computation Laboratory, Stirling.

[KBB93] I. Kraan, D. Basin, and A. Bundy. Middle-out reasoning for program synthesis. In P. Szeredi, editor, *Proceedings of the Tenth International Conference on Logic Programming.* MIT Press, 1993. Also available as Max-Planck-Institut für Informatik Report MPI-I-93-214.

[Mad89] P. Madden. 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. Also available as DAI Research Paper No. 416, Dept. of Artificial Intelligence, Edinburgh.

[Mad91] P. Madden. *Automated Program Transformation Through Proof Transformation.* PhD thesis, University of Edinburgh, 1991.

[Mad92] P. Madden. Automatic Program Optimization Through Proof Transformation. In D. Kapur, editor, *CADE11*, pages 446–461. Springer Verlag, 1992. Lecture Notes in Computer Science No. 607. Also available as DAI Research Paper No. 604, Dept. of Artificial Intelligence, Edinburgh.

[Mad93] I. Madden, P. Green. *Optimization = Verification + Middle-Out Reasoning.* Research Paper, Dept. of Artificial Intelligence, University of Edinburgh, 1993. Extended abstract in Proceedings of *Workshop on Automated Theorem Proving*, IJCAI-93.

[Mad94] P. Madden. Linear to logarithmic optimization via proof transformation. Research paper 416, Max-Planck-Institute für Informatik, 1994. Short version submitted to JELIA-94.

[MB93] P. Madden and A. Bundy. General techniques for automatic program optimization and synthesis through theorem proving. In *The Proceedings of the EAST-WEST AI CONFERENCE: From Theory to Practice - EWAIC'93*, September 1993. Also available as DAI Research Paper No 644, Dept. of Artificial Intelligence, Edinburgh.

[MG94] P. Madden and I. Green. A general technique for automatic optimization by proof planning. In *Proceedings of Second International Conference on Artificial Intelligence and Symbolic Mathematical Computing (AISMC-2)*, King's College, Cambridge, England, August 1994. To Appear.

[MHGB93] P. Madden, J. Hesketh, I. Green, and A. Bundy. A general technique for automatically optimizing programs through the use of proof plans. Research Paper 608, Dept. of Artificial Intelligence, Edinburgh, 1993. Abstract appears in the Proceedings of LOPSTR-93.

[ML84] Per Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis, Naples, 1984. Notes by Giovanni Sambin of a series of lectures given in Padua, June 1980.

[MN88] D. Miller and G. Nadathur. An overview of λProlog. In R. Bowen, K. & Kowalski, editor, *Proceedings of the Fifth International Logic Programming Conference/ Fifth Symposium on Logic Programming.* MIT Press, 1988.

[Pet84] A. Pettorossi. A powerfull strategy for deriving programs by transformation. In *ACM Lisp and Functional Programming Conference*, pages 405–426, 1984.

[Wad88] P. Wadler. Deforestation: Transforming Programs to Eliminate Trees. In *Proceedings of European Symposium on Programming*, pages 344–358. Nancy, France, 1988.

[Wai89] S.S. Wainer. Computability - logical and recursive complexity, July 1989.