MAX-PLANCK-INSTITUT FÜR INFORMATIK

A Survey of Program Transformation

With Special Reference to *Unfold/Fold* Style Program Development

Peter Madden

MPI–I–94–252 October 1994

INFORMATIK

Im Stadtwald
66123 Saarbrücken
Germany

A Survey of Program Transformation

With Special Reference to *Unfold/Fold* Style Program Development

Peter Madden

MPI–I–94–252 October 1994

Authors' Addresses

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

Publication Notes

This paper has grown from what was originally the *Related Work* chapter of my Ph.D thesis [42]. However it has overtime been substantially altered and updated. The first major overhaul was for a "hand-out" for students attending the MPI *Foundations of Program Verification* course (winter term 1993, course organizer: D. A. Basin). It has since undergone some further changes, with the intention of submitting to a suitable organ in the near future (*e.g.* JACM).

Acknowledgements

Although substantially rewritten and extended at MPI, the initial version of this paper was written when the author was an SERC Post-Doctoral Research Fellow within the *Mathematical Reasoning Group* at the Department of Artificial Intelligence, Edinburgh University. Thanks are due to Dr. Alan Smaill and, in particular, to Prof. Alan Bundy for comments regarding the initial draft.

Abstract

This paper consists of a survey of current, and past, work on *program transformation* for the purpose of optimization. We first discuss some of the general methodological frameworks for program modification, such as *analogy*, *explanation based learning*, *partial evaluation*, *proof theoretic optimization*, and the *unfold/fold* technique. These frameworks are not mutually exclusive, and the latter, unfold/fold, is certainly the most widely used technique, in various guises, for program transformation. Thus we shall often have occasion to: compare the relative merits of systems that employ the technique in some form, *and*; compare the unfold/fold systems with those that employ alternative techniques. We also include (and compare with unfold/fold) a brief survey of recent work concerning the use of *formal methods* for program transformation.

Contents

1 Introduction

The body of work, in the available literature, relating to program transformation is dauntingly extensive. In this paper we therefore limit ourselves to reviewing, albeit still extensive, a representative selection of the more influential transformation systems and techniques. We shall keep our explanations of the various systems and techniques at a fairly high-level, providing examples where it adds to clarity. We ensure that pointers to the relevant literature are provided when ever appropriate. Thus this paper should serve as a useful first base for those wishing to pursue an interest in program transformation, but not sure where to begin regarding the extensive litereature.

Common to the majority of transformation systems is the employment, albeit in different guises, of the so-called *unfold/fold* technique. Thus, the *unfold/fold strategy*, from which the basic rules derive, features in the majority of the systems reviewed. Throughout, we shall contrast and compare the various systems: we shall be primarily concerned with certain properties that are generally considered desirable criteria of a transformation system, and to what degree the systems reviewed measure up to these criteria. They are:

1(i). search and control strategies, and;

1(ii). (the degree of) automatability;

2. the correctness of the transformations;

3. the generality of the transformations; and

4. the expressiveness of the program specification language.

We couple 1.(i) and 1.(ii) since the degree of automatabilty hinges on the systemized search and control strategies. Conversely, the degree to which a system requires user intervention is a measure of to what extent that system lacks automatic control.

The strategy originates from [17], and its systemization in an interactive context is first described in [9].[1] The general idea is to transform an inefficient, source functional program into an equivalent, more efficient, target functional program through a process of unfolding and folding recursive definitions. The target program is defined in terms of the source program and then the unfold/fold process is used, as a re-writing strategy, to derive a recursive definition for it independent of the source program. The large search space associated with this process offers ample scope for the employment of various (heuristic) control strategies. Examples, together with formal definitions of *folding*, *unfolding* and further transformation rules, are provided in §2.1.

1.1 The Specification Language and Preserving Equivalence (or Ensuring Correctness)

Before embarking on the systems review, we shall provide a brief introductory commentary on the aforementioned criteria. We divide our discussion into two parts: the first, which forms the main bulk of this paper, concerns the direct transformation of executable code (programs) through the application of transformation rules; the second shorter part concerns the transformation of programs through proof transformations.

[1] To be historically accurate, Manna and Waldinger developed a different, but equivalent, version of the unfold/fold technique around the same time [53].

1.1.1 Transformation of Programs

Although a desirable property, not all program transformation systems are primarily concerned with ensuring the correctness of their transformations. Such systems are referred to as *heuristic* systems, and rely generally on heuristic re-write rules which will produce a target program without the considerable extra work required to formally establish that it is equivalent to the source program.[2] Examples of such heuristic systems include the LOPS system [3], Grant and Zhang's "list-processing" optimization system [25] and the original implementations of Darlington's unfold/fold strategy [9].[3] More recent work of Darlington's provides the unfold/fold transformations with a partial correctness guarantee [19].

- *Providing Correctness Criteria*

Establishing that source to target transformations are *totally* correctness preserving is a broader issue than establishing the correctness, or equivalence, of the various transformation re-writes employed during the transformation. *Total* correctness includes, in addition to correctness, establishing that the transformations will terminate.

In *general*, and due to undecidability factors, it is not possible to establish total correctness unless some sort of restrictions are placed either on the form of the re-write rule applications, or on the sub-set of logic within which the (executable) specifications are formalized.[4]

For example, both [35] and [63] have the restriction that there are always an equal, or greater, number of unfold steps than fold steps. In this way termination, and correctness, can be established ([63] is discussed in §2.1).

A similar approach is taken by [62] where folding is restricted to those newly defined functions wherein at least one (subsidiary) function call has been unfolded. Without such restrictions total correctness cannot be ensured.

For the most part, we shall be primarily concerned with whether or not systems have the property of correctness (since, without some kind of restrictions, non-termination will always be a fact of computational life). That is, *if* a source to target transformation terminates *then* does it terminate with a correctness guaranteed target program.

Of those systems where correctness is a primary goal there is generally one main approach, the refinement of executable specifications (viz. programs) using equivalence preserving re-write rules. The intention is that a source program is transformed by representing it within an executable subset of some logic, and then applying re-write rules that ensure that the input/output relation specified in the source program remains unchanged.

The motivation behind such systems is that each individual re-writing of the source program/specification is in itself guaranteed to preserve equivalence (given the re-write/logic sub-set restrictions). In this way correctness is ensured by the actual (target) program construction process, hence removing any need to (directly) provide lengthy equivalence proofs (of the source and target programs). In the case

[2] Such systems should not be equated with systems that employ heuristics to *select* appropriate re-write rules (i.e., to control the path through the transformation search space). Although heuristics may be responsible for selecting appropriate re-write rules, whether or not the system is correctness preserving depends on whether or not the rules themselves are equivalence preserving.

[3] The LOPS system is in fact more akin to an interactive synthesis system for developing logic programs from an executable specification. The LOPS system is compared with the theorem proving approach to synthesis (namely NUPRL synthesis) in [40].

[4] The reason for the undecidability is essentially due to the possible non-termination of a target program produced by the *arbitrary* application of folding. For example, although the identity function, $id(x) = x$, is terminating, a non-terminating target can be obtained by folding the definition against itself to produce $id(x) = id(x)$.

of recursive program transformation, this means that there is no (direct) recourse to lengthy inductive proofs to establish the correctness of the transformations.

However, this approach somewhat shifts the problem of providing correctness guarantees to the program construction process itself: that the re-write rules are in themselves correctness (equivalence) preserving needs to be established, and this will, as a general rule, require as much effort as providing an explicit proof of correctness for the source to target transformations.

For example, many of the systems that employ the *unfold/fold* strategy re-write the recursive step(s) of a source program through the application of various *equality* lemmas, each of which needs to be proved (by induction) if the source to target transformation is to preserve equivalence (Gregory, 1980; Manna & Waldinger, 1980; Tamaki & Sato, 1984) (we briefly describe these systems in §2.1).

Hogger and Clark, whose work we also address in §2.1, place the condition on their source to target transformations that the input/output relation defined by the executable specification is a sub-set of the relation computed by the target program [31, 12]. That this condition is met, again, requires lengthy inductive proofs.

Furthermore, with such systems, any extension to either the sub-set of logic within which the programs are formalized, or to the set of re-write rules (or both) will require a corresponding extension to the equivalence proof(s).

A more arduous approach is to provide termination proofs at the end of each source to target transformation: terminating programs are then deemed valid since they could only have arisen from equivalence preserving re-writing. For recursive program transformations, these termination proofs will require induction. Unfortunately, termination proofs are generally an undecidable problem, so this approach, again, only ensures partial correctness.

- *The Specification Language*

Kowalski's famous slogan, ALGORITHM = LOGIC + CONTROL, succinctly conveys the notion that, *ideally*, the logic component of a program should be a clear, and correct, statement of the problem, while the control component should be distinct from the logic component and responsible for the efficiency of the program [36].

In practice, it is not generally possible to attain a totally clear and distinct separation of the two components of an algorithm, and the degree of autonomy between the two components is generally dependent on the nature of the specification language.

In general, regarding program transformation systems, the specification language and the programming language are required to be virtually one and the same, since a source program is optimized through the direct application of re-writes to that program. A drawback of this approach is that since the specification itself is tantamount to an executable (source) program then it becomes difficult to avoid placing constraints, *within that specification*, on how the program is executed. This is particularly the case with systems that use a simple functional programming language where the procedural content of the specification/program to be refined may determine to some degree *how* the resulting target program computes the specified input/output relation. This is clearly a restriction since, ideally, how the program computes it's output should be determined by how it is constructed from it's specification, and not by the specification itself. Or, in other words, for the purposes of transformation, the specification should have a minimal effect on the dependencies between facts involved in the computation of that specification.

On the other side of the coin, there is the problem of refinements capture: how well can the specification language be used to capture the description of the task to be computed. Some researchers have found that using a specification language that

is identical to the target language is too restrictive (i.e., it becomes very difficult to specify the exact computational task at hand, and to separate it's description from control issues). Hence the development of special purpose specification languages which facilitate the problem description. For example, at Imperial College a functional language is being developed as a successor to HOPE$^+$ [19]. The language includes facilities for logic, constraint and object-oriented programming features. As such, it allows for the formalization of high-level specifications, and the functional features support the transformational development of HOPE$^+$ like programs. A further example is provided by Manna and Waldinger who have developed a "flexible" specification language for their SYNSYS system, §2.1.10, which can, to some extent, be tuned to the users requirements [54].

- *Automation and Generality*

Practically all the unfold/fold systems rely on user interaction. This applies equally to systems that are primarily concerned with synthesis, as opposed to optimization, and which use the unfold/fold technique such as Bibel and Hörnigs *Heuristic Program Construction System.* Of those systems that have achieved some success in (partially) automating source to target unfold/fold transformations, considerable reliance is often placed on some form of user-provided control program. This is the case with Sato and Tamaki's logic transformation system, [63], and Darlington's current HOPE$^+$ system [19].[5]

In general, the larger the class of transformations desired, then the greater the number of transformation rules that the system must have access to, and hence the greater the search space associated with the (legal) re-writing of the source code. So the more general the system, the more difficult is the task of automating and/or controlling the search within the transformation search space.

An interesting approach to avoiding such control problems is to make the specification language flexible such that it can be tailored to a particular function's requirements (this strategy is adopted in [54]).

Another approach is to employ some form of meta-language to specify tactics, with pre- and post-conditions, which can then control the object-level re-writing. (This strategy is adopted in [21, 26] – *cf.* §2.1.6 and §2.1.7).

Chin describes how an impressive degree of automation can be obtained for unfold/fold transformations that employ the *tupling technique* [11]: the re-writing of a source program is guided by redundancy information culled from an (automatic) construction and analysis of *dependency graphs.* In this way the dependencies between facts involved in the source computation are rendered open for inspection and modification such that repeated sub-computations are then grouped together into a single, more efficient, target tuple structure.

We shall discuss all the aforementioned references in §2.1.

1.1.2 Transformation of Refinement Proofs from (non-executable) Specifications

An alternative, and altogether different, approach to program transformation is *program* transformation through proof transformation. With this approach, if the termination condition is met – the production of a complete target proof – then the transformation is *ipsofacto* correctness guaranteed.

So although the formal proofs of correctness and existence for the source program may be rather lengthy, transformations performed on such proofs have two advantages: firstly, the correctness guarantee of the source to target transformations; and secondly, much of the extra information in such proofs, superfluous to

[5] HOPE$^+$ is an extension of NPL with built in tupling procedures [9].

the actual computation, can be exploited for the transformations (particularly with respect to automation).

One of the first systems to exploit this extra information, by performing program transformation through proof transformation, is Goad's specialization system [24, 23]. The system is not a straight forward optimization system but rather adapts programs to special situations through (pruning) transformations performed on existence proofs. In §2.3 we provide an account of the main properties of Goad's specialization system.

More recently, research has been done concerning straight forward program optimization by performing transformations on synthesis proofs (e.g. [58, 56]). The author also researched into program through proof transformation for his doctoral thesis [42]. Synopses, and extensions, to this work can be found in [41, 43, 45].

We provide an overview of the aforemention "proofs as programs" approach to program transformation in §2.3

2 Program Transformation Review

The lay-out of the review is as follows:

§3.2.1 We begin by describing the unfold/fold strategy for program transformation, within the context of Darlington's pioneering NPL program transformation system. We give formal definitions of *folding* and *unfolding* and illustrate the unfold/fold technique by a worked example.

We then provide brief surveys of systems which employ the unfold/fold transformation strategy in some guise or another.

§3.2.2 We then move on to review briefly systems that use *partial evaluation*, or *explanation based learning* techniques, in order to optimize a source program by, in some sense, observing its behaviour when run on a concrete, or abstract, example.

§3.2.3 Finally, we turn to program through proof transformation. We review Goad's work on program specialization, and briefly describe Pfenning's proposed methodology for meta-level controlled by proof transformation (notably [58]. As already mentioned, there is little to review concerning this approach (especially regarding any working implementation), and so for a detailed account the reader must wait until the descriptions of the author's OMTS system.

Such a categorization of systems is primarily for presentation purposes, and should not be taken too rigidly since, for example, most of the systems that employ partial evaluation also employ the unfold/fold technique, and Goad's proof transformations are initialized by partial evaluation.

2.1 The Unfold/Fold Strategy

The most influential application of the unfold/fold technique is within Darlington's NPL program transformation system. Many of the more recent system designs are based upon Darlington's framework for refining clear but inefficient programs (or executable specifications) into their efficient equivalents [27, 54, 63].

The unfold/fold technique is a specific kind of re-writing which involves matching, and replacing, recursive terms from the developing branches of the target program: by a process of re-writing recursive definitions, a recursive definition for the target program is derived which is independent of the source definition. Although it has been employed, in various guises, in many of the existing transformation

systems (and suggested system designs), it originated within the NPL context of developing and optimizing simple functional programs.

The unfold/fold technique is usually initiated by some form of *eureka step* where the desired target program specification is defined, via lemma introduction, in terms of the source, thus setting the scene for unfolding, followed by folding, to take place. The generation of such lemmas has proved notoriously difficult to automate, as have the control issues involved in deciding whether or not to introduce a fold, or to continue unfolding.

2.1.1 Darlington's Thesis

Developments on Darlington's PhD research resulted in the unfold/fold transformation technique. The most relevant aspect of this research was the use of *schemas*. These schemas express certain recursive forms which, if matched with an input expression, will produce an optimized equivalent. The matching of these schemas with input formulae, and the subsequent transformations, takes place during the compilation processes with the result that high level description involving recursions of arbitrary complexity are transformed into a lower level target language.

Darlington's program improvement system is automated to a large extent. It uses both knowledge concerning the structural form of the input and concerning properties of the specific functions involved. The system employs a *structure recognising* process where by schemas expressing particular recursive forms are compared with the input expression, represented as tree structures, until a match is found. Once a match is found the corresponding schematic rewriting is initiated. The properties of the functions involved in the matching also determine the final outcome of the improvement process (for example, whether the function in a particular rôle is associative or has an inverse).

2.1.2 Darlington's NPL Functional Program Transformation System

Darlington and Burstall have designed a research tool for the development of program transformation methodologies that builds upon Darlington's thesis research [9]. Program transformations are defined as schematic re-writing systems within a functional programming language, together with constraints on the instances of the schemas that must be met in order for the transformation to be valid. The system relies heavily on user instantiations together with rules for their evaluation.

Darlington's NPL transformations rely heavily on symbolic evaluation and are achieved mainly by sequences of *foldings* and *unfoldings* together with *instantiations* provided by the user. In all, there are six main *transformation rules* (re-write rules), R1 to R6, the first two of which, *unfolding* and *folding*, are defined as follows:

(R1) **unfolding**: If $E = E'$ and $F = F'$ are equations and there is some occurrence in F' of an instance of E, replace it by the corresponding instance of E' obtaining F'', then add the equation $F = F''$.

(R2) **folding**: If $E = E'$ and $F = F'$ are equations and there is some occurrence in F' of an instance of E', replace it by the corresponding instance of E obtaining F'', then add the equation $F = F''$.

The central strategy of the unfold/fold transformations consists of generating lemmas to introduce recursions into the developing target program by application of the above **folding** rule R2.

The third transformation rule is composed of numerous **Laws** such as those for associativity, commutativity, etc.

(R3) **laws**: $X + Y = Y + X$, $X + (Y + Z) = (X + Y) + Z$, $X \times Y = Y \times X$ *etc.*

An important facility is the introduction of a **where** clause, R4, by deriving from a previous equation $E = E'$, and given equations $F_1, ...F_n$, a new equation thus:

(R4) **Where Clause Introduction Rule (abstraction):** $E = E'[u_1/F_1, ..., u_n/F_n]$ **where** $\langle u_1, ..., u_n \rangle = \langle F_1, ...F_n \rangle$.

So abstraction consists of replacing parts of an expression, in the body of an equation, by variables, and then defining these variables in a **where** clause. This introduction of **where** clauses is an essential part of evaluating the recursive branches of the *target* program.

Finally, NPL employs a **definition rule**, R5 and an **instantiation rule**, R6.

(R5) **Definition**: Define a new target recursive equation in terms of the source recursive step.

(R6) **Instantiation**: Create a substitution instance of an existing recursion equation.

Darlington's system requires considerable user-interaction. The user must provide the following instantiations for the functions she or he wishes to improve:

- the inefficient ("naive") version of the program (i.e., the source);
- the instantiated left hand side of the target recursion schema base equation; and
- the instantiated left hand side of the target recursion schema step equations.

The system then proceeds to evaluate the base and recursive branches for the efficient program as follows:

> **Base case**: Armed with the above instantiations the system selects an equation, instantiates it as the user requests and then unfolds the right hand side.
>
> **Step case**: The system proceeds as above except that the unfolding may be followed by the application of **laws**, R3, then by sequences of foldings.

Folding often consists of simple matching operations, searching through the current equation list, consisting of the originally provided equations together with those developed so far, and finding an equation an instance of whose right hand side occurs within the right hand side of the developing equation. However, it is the guided control of folding that can lead to more interesting behaviour: different kinds of folding can lead to different recursive patterns and, in particular, *forced folding* uses information from a failure to achieve a simple fold to direct the development of the equation so that a fold can be achieved. So the outcome of the optimization can be determined by the control of folding.

Finding a suitable sequence of unfoldings, finding a suitable fold, and the decisions associated with when to stop unfolding and to introduce a forced fold, all contribute toward search and control problems. A *pre-set effort bound* prevents the repeated application of unfolding becoming too deep.

• *Example: Linearization of Fibonacci by Tupling Followed by Unfold/Fold*

Tupling is an important means of linearizing exponential procedures. It works by grouping together, in a single recursive tuple function, the separate recursive expressions in the source procedure. So, with $i \geq 2$ the "conditions" for tupling are as follows:

Condition 1: There exist two or more *recursive calls* (or expressions), $f(n), ..., f(n-i)$, which share some *common recursion variable(s)* in a function definition.

Condition 2: There exists a fixed sized tuple - the *eureka tuple* - within which common subsidiary recursive calls arising from the execution of each of $f(n), ..., f(n-i)$ can be merged, thus forming a recursive function without the original redundancy.

The provision of the fixed sized tuple constitutes the *eureka step* for program transformation by tupling. In most systems that employ tupling, or some similar form of tabulation, it is achieved through some variant of the abstraction rule (R4). It is also generally achieved through considerable user interaction.

The transformation process starts with the *source Fibonacci* proof of the previous section, duplicated below:

$$\begin{array}{lll} (1) & fib(0) & = 1; \\ (2) & fib(1) & = 1; \\ (3) & fib(n+2) & = fib(n+1) + fib(n). \end{array}$$

Note how this definition satisfies **condition 1** above.

The process of defining the desired optimization in terms of the course of values definition, as described in the previous section, is what Darlington refers to as the "key to the optimization", or the *eureka step.* This is done by the introduction of the auxiliary function fib_{tup} (thus satisfying condition 2 above):

$$(4)\ fib_{tup}(n) = \langle fib(n+1), fib(n) \rangle.$$

This auxiliary function acts as a tuple which, in effect, replaces the source recursion schema with a target schema which combines identical recursive calls.

So Darlington's strategy is motivated by the observation that significant optimization of a (declarative) program generally implies the use of a new recursion schema. This process depends on the *user* providing the requisite definition of the *eureka* tuple fib_{tup}.

The system proceeds to evaluate the recursive branches of the auxiliary function, given the original equations and the instantiated base cases. Armed with the original equations, it is a simple matter for the system to evaluate the base case for fib_{tup} given the left hand side of the equation, $fib_{tup}(0)$,

$$(5)\ fib_{tup}(0) = \langle 1, 1 \rangle.$$

By using R4 to introduce a *where* clause, the system produces a definition of Fibonacci, *in terms of our auxiliary function* fib_{tup},

$$(6)\ fib(n+2) = (u1 + u2), \text{ where } \langle u1, u2 \rangle = fib_{tup}(n).$$

Forced folding then comes into play for the optimization of $fib_{tup}(n+1)$: given the instantiated left hand side of the recursive step, unfolding produces the equation

$$(7)\ fib_{tup}(n+1) = \langle fib(n+1) + fib(n), fib(n+1) \rangle.$$

The system then attempts to fold this equation with

$$(8)\ fib_{tup}(n) = \langle fib(n+1), fib(n) \rangle,$$

but fails since there is no direct match between the two. By observing that all the components necessary to match equation (8) are present within equation (7) the system *forces* the match, by using R4, to rearrange equation (7) to the following

$$(9)\ fib_{tup}(n+1) = \langle u1 + u2, u1 \rangle, \text{ where } \langle u1, u2 \rangle = \langle fib(n+1), fib(n) \rangle$$

This now easily folds with (8) yielding the desired optimized function definition

$$(10)\ fib_{tup}(n+1) = \langle u1 + u2, u1 \rangle, \text{ where } \langle u1, u2 \rangle = fib_{tup}(n)$$

2.1.3 Remarks

The following remarks may be made concerning NPL's deployment of the unfold/fold technique:

1. In general, the unfold/fold strategy eureka steps correspond to the introduction, by the user, of an auxiliary function which, by the use of abstraction, defines the target program in terms of the source. Clearly, a desirable goal of transformation systems is to circumvent the eureka step by providing target definitions *automatically.*[6] As a general rule, the larger the class of programs which can be successfully transformed by a system then the more user interaction is required to provide the associated *eureka steps.*

2. Considerable user interaction is required to guide the unfold/fold process. The construction of an appropriate sequence of unfoldings, foldings, and re-writings is almost as difficult as the construction of a formal proof of a program.[7] That is, the search for a suitable *fold* presents control problems. So, recalling the previous remarks, the two most problematic steps in the unfold/fold strategy are:

   (i) the *eureka step*: obtaining the initial definition of the target in terms of the source; and

   (ii) the control problems associated with when to apply the *fold* re-writing step(s) which eliminate any reference to the source definition from the target recursive step.

   This control problem is compounded by the trade-off between the degree of automation and the size of the class of unfold/fold transformations one wishes the system to encompass (henceforth, the author will refer to this trade-off as the *automation trade-off*). Darlington's attempts at partially automating his unfold/fold technique were, by his own admission, blocked by the fact that the heuristics he used only covered a fairly small class of problem, and were not flexible enough to be used uniformly [18].

3. The unfold/fold transformations require numerous applications of laws, *cf.* (R3) §2.1.2, for which any overall strategy is difficult to characterize (and hence difficult to automate). The application of semantic laws can lead to an infinite regress wherein a newly introduced function definition requires unfolding which in turn leads to the introduction of a further new function definition (*cf.* §2.1.9).

4. The following remarks, (a) - (e), re-iterate what we said in §1.1, only within the context of Darlington's transformations.

   (a) In NPL both the specification and target language are recursion equations. However, the language is not a suitably expressive (or declarative) specification language: i.e., it is not easy to express what a program should compute without worrying about how it should be computed.

   (b) The original unfold/fold strategy, as it was presented in [9], was not provided with a *correctness guarantee* for the source to target transformations: the system itself performs no verification checking, the onus being on the user to accept or reject the "improvements" made at each stage of the transformation. However, later incarnations have been shown

[6] [11] addresses this goal within the context of tupling transformations.

[7] Recall that these are *not* the same: a formal proof will incorporate a verification that the program computes the desired (specified) input/output relation.

to have a partial correctness guarantee for specified classes of functions (notably [19, 11] and [63]) - see next remark.

(c) With systems based upon the NLP design, a correctness guarantee for the source to target transformations can, *in principle*, be provided without *directly* using any recursion/induction principle but by a sequence of identities, or equality lemmas, where each identity corresponds to one of the six rules (R1 - R6) of the transformation system. That the equality lemmas are indeed equivalence preserving must *necessarily* be proved if they are, collectively, to ensure correctness of any source to target transformation.

(d) Furthermore, each extension to the class of functions requires a corresponding extension to the set of identities, or equality lemmas, which in turn will require a corresponding extension to the (set of) equivalence proofs.

5. Darlington's system provides some useful methodological tips for optimization (regardless of whether we are concerned with the direct transformation of source code, or with optimization through proof transformation). Below we sketch the strategy involved with tupling transformations:

   - define a tuple auxiliary function, f_{tup}, in terms of the step cases of one's known function definition f;
   - try to evaluate the step case, $f_{tup}(n+1)$, of this tuple function, perhaps by using unfolding, folding and forced folding or something equivalent, such that:
     - The auxiliary function definition is cashed out in terms which *do not* have recourse to the original definition; and so,
     - the recursion schema employed in the original definition is transformed into a more efficient one, e.g., as in the linearization of course of values recursion into stepwise recursion.

2.1.4 General Strategy for Unfold/Fold

Different transformation systems employ different transformation rules although it soon becomes clear that most share a common core, even if the jargon varies. This usually consists in some combination/variant of the six rules employed by the NPL system. The main cause for divergence of these systems, from the NPL system, is generally due to the particular *logic*, or perhaps *formalism* is more appropriate, used for the specification and/or target languages.

In **fig.** 1, we have abstracted a general strategic plan for *program* transformation systems from those systems which, like NPL, employ some variant of the unfold/fold strategy. Note that the strategy requires user intervention at several key points (1, 2, 4, 5). In particular: the generation of target definitions in terms of the source, 2, and the application of a *fold* in order to introduce a recursion into the target definition, 5. Note also that the tupling technique is subsumed by the general plan.

There are many existing program transformation systems that employ some equivalent variant of the unfold/fold technique. As illustrated above, the strategy requires an *eureka step* followed by unfolding and then folding in order to introduce recursion into the target program. So, in order to achieve a complete transformation the system must have some means of *controlling* the folding and unfolding. Exactly *how* this is achieved is what distinguishes many of the current *program* transformation systems. Other distinguishing features include the transformation

1. User inputs some form of *program specification*, at the very least specifying the input/output relation, including any specially defined procedures. The specification may or may not be computable.

2. User specifies a number of suitably instantiated goal equations E (*eureka step*).

3. Each of E is *symbolically executed* – unfolded – repeatedly using some computation rule.[a]

4. Laws are optionally applied. Control usually passed to user or determined by user provided heuristics/rules.

5. Further symbolic execution – unfolding – followed by folding is performed/attempted until a recursion is obtained

6. The final equations are ordered for computational use.

[a] The *computation rule* may be, for example, that of a functional language or logic language (the later usually being the standard computation rule for Prolog or some idealized version thereof).

Figure 1: A General Plan for Unfold/Fold

application, the *language of transformation*, and the degree of automatability/user intervention required to attain a complete transformation. By *language of transformation* we mean the form of the unfold/fold technique which they employ. This depends on the form of the equations being (un)folded which, in turn, depends on the particular logic/formalism employed (e.g. functional or logic). In the following sections we provide a brief survey of some of the more notable extensions/variations on Darlington's prototypic unfold/fold model.

2.1.5 Extended NPL Functional Transformation with Automated *Eureka*

Darlington's *Functional Programming Environment*, FPE, supports the transformational development of HOPE$^+$ programs, and as such is tailor made for tupling transformations. The FPE operates as a *transformation processor* that applies user-generated transformation plans, or *scripts*, to programs.

The transformations achieve some degree of automation by, in effect, carrying around a large open-ended tuple, or more precisely a variadic function which simulates the action of tuples, whose length is tailored to whatever the particular function undergoing transformation requires. This tailoring is controlled by the system containing a large "lookup" table which, via a complex management module, provides information on how to tailor the tuple length to the function's requirements. In fact, the system must contain quite *function specific* knowledge in order to account for the substantial creativity required to formulate clauses, additional to the ones describing the problem's logic, on which the folding can be performed. This method is successful for a fairly broad range of functions, although interaction is still required in order to guide the management module. As a consequence of the *automation trade-off*, the more complex the recursive equations of the initial source program then the correspondingly more complex the heuristic set has to be in order to tailor the open-ended tuple structure. Thus much of the elegance of the original NPL system is lost.

2.1.6 ZAP: NPL Functional Transformation with "Metaprogram" Control

Feather's system, ZAP, employs the six main rules of Darlington's NPL system, R1 - R6, together with a meta-program to control the transformation of a "protopro-

gram", the latter comprising the source recursive equations [21]. The meta-program is composed of various special purpose re-write rules which reduce the transformation search space. These re-writes take the form of rules specifying which functions are to be used for unfolding and which may occur in the transformed equations. As such they are more like meta-level tactics which dictate the application of the basic six unfold/fold rules. The tactics enable two main types of optimization: *combining* embedded functions which traverse an intermediate data structure more than once into a function with a single pass, and *tupling* where by, as explained in §2.1.2, subsidiary recursive calls can be merged into a single step in the optimized definition. Feather emphasizes that the meta-level programs should be viewed as advice to the transformation system which in no way effects the correctness of the final program produced.

ZAP is a semi-automatic system. The user must intervene in order to:

- input a *context* which consists of definitions of functions, lemmas about them, names of functions to be unfolded, names of functions to be permitted to remain in the improved definition and outlines for functions to be improved.
- introduce the auxiliary *eureka tuple* as in the case of NPL;
- supply each instantiated left hand side of the developing target, together with,
- a high-level transformation plan - a meta-level program - for the desired transformed equation (this places a very big onus for the transformation on the user);
- activate "default generators" which employ user-supplied type information in order to, for example, automatically generate the instantiated left hand side; and
- prove the re-writing lemmas applied by the system.

Although all transformations take place within Darlington's NPL system and formalism, Feather's extension does not benefit from the specification/target language uniformity: the initial user-directed transformations convert the specifications into a more efficient form suitable for direct translation to programs in a conventional high level language. Clearly there is potentially a large translation overhead here.

An interesting feature of ZAP's *combining* tactic is an *approximation* facility which uses higher-order matching in order to fill in structural details when, for example, the user fails to provide enough information through the "default generators". For example, consider the following non-recursive definition for a function that sums the squares of each of a list of numbers:

$$\begin{aligned} sumsquares(L) &\Rightarrow sum(squares(L)) \\ sum(nil) &\Rightarrow 0 \\ sum(N :: L) &\Rightarrow N + sum(L) \\ squares(nil) &\Rightarrow nil \\ squares(N :: L) &\Rightarrow N * N :: squares(L) \end{aligned}$$

An approximate goal for a recursive definition of *sumsquares* would take the following form:

$$sumsquares(N :: L) \Leftarrow \mathcal{M}(N, sumsquares(L))$$

where $\mathcal{M}$ is a second-order variable. This goal unfolds to:

$$N * N + sum(squares(L)) \Leftarrow \mathcal{M}(N, sum(squares(L)))$$

where upon matching instantiates $\mathcal{M}$ to $\lambda x \lambda y.x * x + y$.

The main weaknesses of Feather's system are that the meta-level descriptions are themselves rather weak (consisting primarily of pattern-oriented transformations), and there is no real strategy for using the tactics.[8]

2.1.7 Tactic Driven Unfold/Fold

The weakness of ZAP's meta-level descriptions have motivated Green to investigate the application of meta-level transformation tactics which are partially specified within a meta-language, and identified by specific pre- and post-conditions [26]. These tactics can then be used to control the automatic transformation of programs within the context of the unfold/fold strategy. The selection of tactics is guided by special property-oriented abstractions on programs. These abstractions, in effect, allow the system to decide, heuristically, which of the tactics whose pre-conditions are satisfied is the most promising. However, once the tactic application has terminated, considerable further work is required to actually attain the optimized target: rather than producing a target program, the tactic applications produce an object-level transformation sequence. This may then require substantial refinement in order to specialize it to the program undergoing improvement. Nevertheless, the use of a meta-language to guide the automatic construction of a target transformation sequence is a neat approach, and one which is similar to using a meta-language in which tactics can be expressed in order to automatically construct proof-plans.

Green's research, although in it's early stages, holds considerable promise toward automating a large class of unfold/fold transformations: the goal-directed reasoning at the meta-level, which is responsible for the planning strategies, operates within a much smaller search space than the object-level transformation space.

2.1.8 Unfold/Fold Program Transformations with Automatic Tupling

Although many of the unfold/fold program transformation systems rely on procedures for generating new predicates/functions, and their definitions, on which the folding can be performed there has, to date, been limited success in automating the *eureka step*, although this is surely wherein most of the "intelligence" of the transformation lies.

Recently however, Chin, a student of Darlington's, has described several methods for automatic program transformation within the HOPE system [11]. Although Chin documents an impressive range of automatic methods for program transformation, which are currently undergoing implementation, the most relevant to this thesis is his description of automatic tupling techniques. By an analysis of *symbolic dependency graphs*, based on [57], Chin is able to describe an automatic procedure for finding a pair of *matching tuples* by the unfolding of selected calls to the source program, and then using matching as a means of testing for successful folding. The process is best described by example. We shall again use the Fibonacci function.

A dependency graph, DG, is a representation of a particular function call's evaluation tree which shows the calling structure of the subsidiary recursive calls. A *symbolic* DG is based on function calls which are potentially infinite in size. The initial portion of the symbolic DG for *Fibonacci* is shown below in **fig.** 2. The multiple evocations of subsidiary calls, the redundancy pattern, is exhibited by more than one arrow directed at any particular node.

The main idea taken from [57] is that:

[8] These weaknesses provided the motivation behind Green's development of transformation tactics with formally specified pre- and post-conditions.

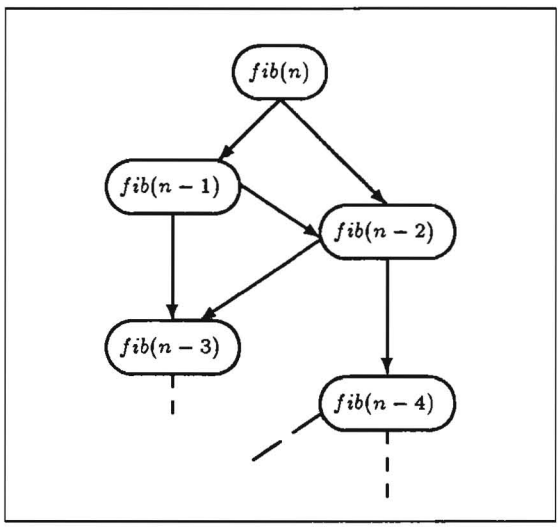

Figure 2: The symbolic DG for $fib(n)$

> An appropriate eureka tuple can be found if and only if there exists a *progressive sequence* of *cuts* that *match* one another, in the function's dependency graph.

A *cut* is defined as a subset of nodes across a dependency graph that when removed will divide the graph into two disconnected halfs. A *progressive sequence* of cuts is a sequence of cuts ordered according to size (i.e., according to the number of nodes in the subset). A pair of cuts *match* if a consistent substitution can be obtained when each function call of the first cut is matched with the corresponding function call of the second cut.[9]

The finding of an appropriate *eureka tuple* depends on the notion of a continuous sequence of cuts. This is defined by Chin as follows:

> A *continuous* sequence of cuts, $cut_1, cut_2, ..., cut_N$, is a successive series of cuts which starts with the root node as its first cut. This sequence successively obtains the next cut by giving up a subset of nodes... from the *topmost set* of the current cut in order to acquire the children for the next cut.

The topmost set of a cut is defined as a set of nodes whose ancestors are not present in the cut itself.

Returning to the example and starting with the main function call, Chin's analysis replaces $fib(n)$, the first cut, with its two subsidiary calls, $\langle fib(n-1), fib(n-2)\rangle$. This gives us the second cut. The analysis then proceeds by unfolding only that call in a cut which is *not* a subsidiary call of the other call, i.e., the topmost item. So, since the function call $fib(n-2)$ is a subsidiary call of $fib(n-1)$, only $fib(n-1)$ is unfolded. This gives the third cut, $\langle fib(n-2), fib(n-3)\rangle$. The third cut matches the second cut, thus providing the analysis with a matching tuple.

Chin's process is essentially the same as that described for Darlington's tupling technique: the unfold/fold steps required for the tupling transformation are achieved by locating a pair of matching tuples by the unfolding of appropriately selected calls and then using matching as a means of testing for successful folding. The

[9] These terms are *formally* defined in [11].

main difference is that the use of such selection ordering allows for a considerable degree of automation, since once this analysis succeeds the main task of the tupling transformation – finding a successful fold – will have been achieved.

The automation hinges on the production of a continuous sequence of cuts. This in turn hinges on producing an appropriate *ordering* for selecting nodes to unfold during the analysis. So the production of (symbolic) DGs, and their subsequent analysis, are an essential ingredient of *automating* the tupling technique. The requisite dependency information simply is not present, for inspection and modification, in the (source) program code.

In general, the tuple analysis is semi-decidable and non-deterministic. The analysis is not decidable since some programs will cause successive cuts to increase in size. I.e., the analysis cannot handle programs with *dynamic tuples*, such as the binomial function:[10]

$$\begin{aligned} binomial(0, m) &= 1; \\ binomial(m, m) &= 1; \\ binomial(n, m) &= binomial(n-1, m-1) + binomial(n-1, m). \end{aligned}$$

The analysis is non-deterministic because there is often more than one way in which a selection ordering can be produced.

2.1.9 Deforestation Transformations through Unfold/Fold

[65] characterizes a class of functions which can be optimized using the so-called *deforestation algorithm*. Deforestation consists of optimizing the manner in which intermediate outputs are *produced* by one function call and then *consumed* by another. Thus, the goal is to remove unnecessary intermediate data structures and/or to reduce the number of function calls used (i.e. to produce *good consumers* and/or good *producers*). This is achieved by *fusing* compositions of consumers and producers together such that the target program operates as a more integrated and efficient unit. Deforestation is realized through (controlled) *folding*, *unfolding* and *define* transformation steps for for eliminating intermediate terms which may occur depending on the *functional composition* of a program.

Wadler's original, and influential, deforestation algorithm was formulated as an automatic transformation algorithm that transforms any expression composed solely from a *sub*-class of first-order functions. This sub-class is characterized by a syntactic property called *treelessness* which, in fact, can be non-formally defined as functions which are both good producers and good consumers. Treelessness requires: firstly, that the expression has no *intermediate* terms, i.e. that each function making up the expression takes only variables as arguments, and; secondly, that the expression is *linear*, i.e. each free variable occurs only once in each function call (this avoids inefficiency caused by the duplication of large expressions during unfolding).[11]

An example of an expression made up solely from *purely treeless* expressions is the following:

$$append(append(x, y), z) \tag{1}$$

This is in the correct form for deforestation since the append function is a pure treeless function. Thus, using Wadler's deforestation algorithm, this can be transformed, to a new expression

$$app(x, y, z) \tag{2}$$

[10] A program that uses a *dynamic tuple* is one that may require *different* sized tuples at each successive recursive call.

[11] Formal definitions are provided in [10, 65, 22].

which uses a newly defined function, *app*. The definition of this new function (produced through unfold/fold transformations),

$$app([\,], y, z) = append(y, z) \quad (3)$$
$$app(h :: t, y, z) = h :: app(x, y, z) \quad (4)$$

is more efficient since it no longer needs to produce an intermediate list. An example of a non pure treeless function is the *sumdb* since it contains a nesting of two (treeless) functions, *sum* and *double*.

Chin's contribution to deforestation consists in a gradual extension of this sub-set to include *all* first-order functions, rather than just a sub-set, as candidates for deforestation [10]. First he introduces a novel parameter annotation scheme which allows for the deforestation of sub-expressions even if the whole expression fails Wadler's treelessness criterion for deforestation. Such expressions are called *e-treeless*. This class of expressions is then widened to include *all* first-order expressions as candidates for deforestation, by the use of a `let` construct. This basically unpacks an expression in a certain way such that it can be deforested without the possibility of an infinite regress of unfoldings. Chin calls the new treeless form that uses the `let` construct *universal-treeless* form.[12] For example consider the following *rev_flatten* function:

$$rev\_flatten([\,]) = [\,]; \quad (5)$$
$$rev\_flatten(h :: t) = append(\underbrace{rev\_flatten(t)}, h). \quad (6)$$

This function operates on a list of lists and is used to simultaneously flatten the outer list (i.e. *rev_flatten*/1 flattens trees into lists). This function is non-treeless since the step equation of the definition contains a recursive call in the first parameter of the *append* function (i.e. the underbraced term in (6)). As a result the (extended) deforestation algorithm runs into trouble when attempting to transform the *rhs* of the step equation so as to attain the following optimized version, $rev\_flatten_t$, of *rev_flatten*: (where $rev\_flatten_{bi}$ is a binary function.)

$$rev\_flatten_t(x) = rev\_flatten_{bi}(x, 0);$$
$$rev\_flatten_{bi}([\,], a) = a;$$
$$rev\_flatten_{bi}(h :: t, a) = rev\_flatten_{bi}(t, h <> a)$$

Basically, the transformation is non-terminating due to the introduction of progressively larger intermediate functions, using a `define` rule, each time the recursive call to *rev_flatten*/1 is unfolded (precisely the opposite effect that we want with deforestation). Thus, for any n, the *nth* unfolding is followed by the introduction of a new function with a new accumulator a_n:

$$rev\_flatten_n(x, a_n, ..., a_1) = append(\,...\,append(rev\_flatten(x), a_n), ..., a_1)$$

The moral of such regresses suggests that since *rev_flatten* is non-treeless it should *not* be selected for unfolding.

Chin's final extension to Wadler's deforestation is universal treeless form. This provides a "syntactic trick" whereby non-treeless functions, such as *rev_flatten*, can be made to *appear* treeless for the purpose of preventing the infinite regress of `unfolds` followed by `defines`. The method is to make use of a `let` construct so as to extract out the offending subterm(s) thus rendering the expression *pseudo-treeless*.

[12] Both Wadler's deforestation algorithm and Chin's extensions thereof are provided with termination proofs.

The *rev_flatten* definition can thus be made pseudo-treeless as follows:

$$rev\_flatten(h :: tl) = \texttt{let}\ v = rev\_flatten(tl)\ \texttt{in}\ append(v, a)$$

The `let` construct abstracts out the offending subterm, $rev\_flatten(tl)$, in $append(v, h)$ by introducing a local variable v. In this way, both the subexpressions of the `let` construct are rendered treeless. Thus the whole equation is rendered pseudo-treeless and can therefore be transformed using Chin's universal-treelessness algorithm. However, this is as far as the transformation of *rev_flatten* can go using unfold/fold based deforestation for first-order expressions. The introduction of the intermediate function with a `let` construct, in fact, leads to a loss of efficiency. The main benefit of universal treelessness appears only to be that large composite function definitions which contain non-treeless *subsidiary* functions *may* be optimized if we can convert the non-treeless subsidiary functions into pseudo-treeless form (even though the resulting pseudo-treeless subsidiary functions will *not* contribute to any increase in efficiency in any way other than allowing the transformation process to go through).

[47] provides an in depth analysis of deforestation, and [46] provides an alternative approach to achieving such optimizations (by exploiting the proofs as programs paradigm rather than using the unfold/fold transformations).

2.1.10 Transformation of Functional Programs Using Unfold/Fold Technique *and* Adaptable Specification

Manna and Waldinger have developed a synthesis/transformation system, SYNSYS, which is almost identical to the NPL system except they make use of a special purpose specification language which facilitates the problem description, or specification [54]. This means that the target language (LISP) is not the same as the specification language and there is an according increase in the complexity of the transformation operations. However, this specification language may be extended indefinitely by the user and adapted to suit particular situations: transformation rules are supplied for each construct in the specification language, to transform it eventually into a "primitive program".

The SYNSYS transformations are automatic. However, although there may be no user interaction during the transformation, there is good deal of user provided information – in the form of context-specific re-write rules – prior to the transformation: one of the main motivations behind SYNSYS is that one can *extend* and *adapt* the specification language to deal with particular function's requirements. In this way, by tailoring the specification language, the control problems associated with the unfold/fold methodology can, to some extent, be avoided. However, to be able to handle a large class of transformations, the SYNSYS system requires a large number of transformation rules, each of which, if source to target correctness is desired, require an equivalence proof.

2.1.11 Equivalence Preserving Unfold/Fold Transformations

Sato and Tamaki's logic transformation system, [63] is essentially the same as Darlington and Burstall's unfold/fold system, the main differences being:

- logic programs, as opposed to functional programs, are transformed (specifically *pure Prolog*); and
- emphasis is placed on the correctness of transformation which is not guaranteed by Darlington and Burstall's system.

Recalling §1.1, the general "advantage" of declarative programming languages is that the program need only specify the relation between input and output leaving the processor to deal with how output is computed from input. However, considerations concerning efficiency of computation remain unaccounted for. The motivation behind the transformations is to equip the declarative programmer with tools for attaining efficient declarative programs:

> To make the declarative programming style a real advantage of logic programming, we need a programming environment where lucid, specification-like programs are automatically or semi-automatically transformed into less lucid, efficiency-oriented programs, [63].

The second main difference is that Tamaki and Sato's system guarantees equivalence for *each* specific transformation rule they apply (i.e., a correctness proof is provided for their system). This is not really a very important difference since, although Darlington and Burstall's original implementation of the unfold/fold strategy was not provided with a correctness guarantee, a correctness proof for their transformations can be provided by elaborating their functional formulation and by adding a formal semantics to the language. Indeed, this has since been done within Darlington's *Functional Programming Environment*, FPE, which supports the transformational development of HOPE$^+$ programs.

2.1.12 Proving Program Equivalence through Unfold/Fold Transformations

M. Proietti and A. Pettorossi investigate the use of program transformation, specifically unfold/fold, as a tool for proving program equivalences [60]. The idea is not a new one at all (e.g. *cf.* [35]): if a target program is derived from a source program using semantics preserving transformation rules then the source and target are equivalent. Proietti and Pettorossi apply the idea to unfold/fold transformations of logic programs and describe how sets of transformations can be turned into systems for proving and verifying logic program properties. The power of such systems is increased by exploiting symmetry and transitivity of programs.

2.1.13 Synthesis of Prolog Programs from Unfold/Fold Proofs

Correctness proofs for Unfold/Fold transformations of logic programs have previously been provided by [63]. However, Proietti and Pettorossi point out that such proofs do not capture the termination behaviour of logic programs when they are evaluated under the standard depth-first strategy of Prolog. This leaves the user with the task of checking that the transformation techniques behave correctly when applied to Prolog programmers. Proietti and Pettorossi are concerned with the use of program transformation, specifically unfold/fold, as a tool for proving program equivalences [61]: if a target program is derived from a source program using semantics preserving transformation rules then the source and target are equivalent. Proietti and Pettorossi apply the idea to unfold/fold transformations of logic programs and describe how sets of transformations can be turned into systems for proving and verifying logic program properties. The power of such systems is increased by exploiting symmetry and transitivity of programs.

2.1.14 Applying Unfold/Fold Transformations Expressed with Second-Order Patterns

Huet and Lang theoretically investigate augmenting the unfold/fold technique with meta-level search strategies [34]. This is done by recasting the transformation task

as a task in second-order unification. Their research is theoretical and there is, as of yet, no implementation. The application of the various transformation schemas employed in Darlington's work is made more efficient by combining various schemas and/or eliminating redundant ones. New schemas may also be added if required.

Huet and Lang use *transformation templates* which are triples consisting of a schematic description of the source and target functional programs together with a list of constraints which apply to the descriptions. Huet and Lang are primarily concerned with the relation between templates, i.e. with the subsumption, generalization and omision of templates.

The applicability of a template is, perhaps somewhat surprisingly, only of secondary concern. Applicability is tested by matching the template's first element against the program fragment. Since no variable in any of their schemas is of a higher order than a function then second-order matching is sufficient for the transformation task. So, if Σ is a schematic description of a functional program f, and Σ' the schematic description of the corresponding optimised functional program f' then matching involves searching for a substitution σ such that $\mathcal{P} = \sigma\Sigma$, where $\mathcal{P}$ is some fragment of f. If each of the templates constraints are valid under the substitution then $\mathcal{P}$ can be replaced by $\sigma\Sigma'$.

The main problem for this approach is the search involved with the second-order matching: the difficulties in chosing from the various possible substitutions is compounded by the fact that any system would also have to identify the variables which occur in the 'output' and not in the 'input', by using proofs of the constraints.

2.1.15 The *Least General Generalization* Approach

S. Dietzen and R. Scherliss describe a transformation methodology concerning the use of *analogy* for program optimization [Dietzen-Scherliss 81]. Dietzen and Scherliss, following the technique originally developed by Huet and Lang, §2.1.14, suggest *merging* the target and source problems such that they are both *instances* of the same "prototype" generalization. That is, given a successful source transformation sequence, S, we replace sub-terms in the source initial state, i, by variables such that we obtain the *least* syntactic generalization of uninterpreted terms.[13] This *least general generalization*, LGG, can be thought of as the mirror complement to the unification process wherein a *most* general unifier is sought to "merge" two expressions. The merging is controlled by heuristically guided *pattern matching*. For example the LGG between the *factorial* definition, $fact$,

$$fact(n) \Longleftarrow if\ n = 0\ then\ 1\ else\ n \times fact(n-1)$$

and the *reverse list* function, rev,

$$rev(l) \Longleftarrow if\ L = []\ then\ []else\ append(rev(tl(l)),\ cons(hd(a), []))$$

is;

$$f(x) \Leftarrow if x = a\ then\ b\ else\ g(f(h(x)), x)$$

So once we have our LGG we can use this as a general plan for the target solution sequence.

Although this has much in common with past analogy work it shares more in common with the explanation based learning rationale: from a specific example we form a generalized "prototype" by substituting sub-terms for variables. Then we re-instantiate the "prototype" with target sub-terms and, hopefully, arrive at a solution sequence to the target problem.

[13] The terms *source* and *target* here refer to *transformation sequences* as opposed to the object-level equation developments.

For the present, note first, that the LGG formation introduces a new function g. This is equivalent to the *eureka step* whereby new lemmas, functions and definitions must be introduced in order to develop the recursive branches of the target. I will not show how this is done here since, in all essentials Dietzen and Scherliss, quite apart from not actually having a working implementation of any form, are not so much concerned with the transformation of recursive programs but rather, stepping up a level, with the (meta-level)transformation of past successful program optimizations.

2.1.16 "Standard" Representation to *Difference-list* Transformation

Grant and Zhang have presented an algorithm for automatically transforming *Prolog* programs into their equivalent difference-list forms which exhibit more efficient list-processing behaviour [25]. If we denote a difference-list by L/E then this means *the difference between* L *and* E i.e., L/E represents the part of L with E removed. The reverse of a list using a representation whereby the first argument is represented as a difference list would be as follows:

$$reverse([], L/L),$$
$$reverse([H \mid T], L/R) : -reverse(T1, L/[H \mid R]).$$
$$rev(L, R) : -reverse(L, R/[]).$$

There are no calls to append in this definition resulting in this being an $O(n)$ algorithm as opposed to the naive version which is an $O(n^2)$.[14]

The approach Grant and Zhang take in transforming "standard" Prolog procedures into their corresponding difference-list representation shares much in common with that of Darlington and Burstall's unfold/fold technique: new definitions are supplied, the *eureka step*, to allow (un)folding with the original source definition equations and/or any other equations in the current equation set. Several heuristics are supplied for the control of the unfold/fold technique. Grant and Zhang also employ further techniques in their transformations such as partial evaluation, procedure combining and data structure mapping.

The *automation trade-off* clearly affects Grant and Zhang's system: the transformation deductions, in particular the rule set used to form new procedures and definitions, are in danger of leading to a combinatorial explosion. The heuristics employed to guide the unfold/fold technique are hence somewhat specific to the difference-list transformation domain. It is this limited application which allows for the degree of automation (including that of the *eureka step*) achieved by the system.

2.1.17 Transformation of Annotated Logic Programs

S. Gregory has investigated the feasibility of "compiling" logic programs bearing the control annotations of IC-Prolog into sequential logic programs [27]. This is achieved by the application of the classical unfold/fold technique. Indeed, Gregory stays close to Darlington's NPL transformation methodology, the main differences with his system being:

- the application (namely, the transformation of control annotations of IC-Prolog into sequential, and annotated, logic programs); and
- the use of Horn clauses for both the specification and target languages, thus allowing for simpler transformation rules (although this approach renders the specification language less powerful).

[14] I.e., the naive version makes of the order of n^2 recursive calls whereas the difference-list version makes of the order of n recursive calls.

No correctness proof is provided, although we may assume that, since Gregory essentially employs the six main transformation rules of the NLP system, that the transformations are, at least in principle, equivalence preserving. Gregory also shows that the compilation of the resulting annotated logic programs can be partially automated.

2.1.18 Horn Clause Program Derivation from Standard Logic Specifications

C. J. Hogger regards program derivation as the top-down symbolic execution of a standard logic form specification, [31, 30]. The main technique of the system is, again, based on Darlington's unfold/fold approach, except here we are concerned with attempts to synthesize *Horn clause* programs as opposed to NPL type functional algorithms. Hogger views program construction as a goal-oriented derivation. The characterizing properties of Hogger's system are:

- the *specification* comprising a set of *axioms* defining the desired input-output relation to be computed;
- A special purpose *logical deduction system* used to derive a set of computationally useful Horn clauses;
- the derivations, which are equivalence preserving, consist of sequences of rule applications which are classified as follows:

  Goal simplification: replaces the current goal by logical implication,

  Goal substitution: introduces new information by substituting (sub) terms in the current goal for (sub)terms in the specification axioms – Goal substitution is user-activated by a "call";
- a variant of the *eureka step* for lemma generation: recursions are introduced into the derived clauses by a tailored version of the *folding* rule (this recursion introduction process being very much user-guided).
- a termination point corresponding to (i) failure or (ii) a successful final goal constituting the body of the derived procedure, whose head is the current substitution instance of the initial goal.

The input to the system is a single "call" for which a procedure is sought. After the application of the above rules, the resulting derivation *roughly* resembles a conventional top-down logic program.[15]

Clark also treats program derivation as the top-down symbolic execution of a standard logic form specification [13, 14]. He does, however, augment his system with further derivation rules which perform more sophisticated pattern matching, and subsequent substitution, operations by making greater use of terms in the derivation process.

An important motivation behind the systems of Hogger and Clark was the development of a suitably declarative specification language. However, more recent systems, such as Darlington's *Functional Programming Environment*, NUPRL and OYSTER are equipped with better facilities for supporting the specification of problems (as opposed to their procedural solutions).

[15] This is because a call need not be atomic, and the replacement of a call by a "body" is determined by one of the substitution rules.

2.2 Transformations Based On Explanation Based Learning/Partial Evaluation

The *explanation based learning* approaches to program transformation all have in common the use of either a *particular* instantiation of, or an abstract input to, a source program in order to *guide* the development of a *general* target program. By observing the behaviour of the source program, when run on a concrete or abstract query, the system, or user, can modify that behaviour according to the desired transformation application (hence the correlation with partial evaluation). The applications differs considerably. In [20] an example driven transformation is used to remove redundancies from simple function definitions by controlling the *folding* and *unfolding* of equations in the current equation set. In [4] a source program is improved for a small number of abstract inputs such that the target program improves on the execution speed of a given Prolog program, by manipulation *only* of the computation rule under which it is executed. That is, the Prolog *compiler* is optimized. In [34] explanation based learning techniques are a suggested means of transforming past successful source to target transformation sequence. This involves forming a generalized program "prototype" from the source, again by executing the source program on a specific query.

The relation - or, arguably, equivalence - between partial evaluation and EBL has been studied in [64]. Intuitively speaking, we can see how the two techniques merge within the optimization field since EBL guided transformation *just is* using a specific example of the source - a specific partial evaluation - to drive the unfold/fold transformation of the source such that repeated/identical subcomputations are removed.

2.2.1 Explanation Based Learning Transformation of Logic Programs

We shall henceforth refer to the transformation strategy of Brunynooghe, De Raedt and De Schreye of applying *explanation based learning* techniques to the *program* transformation domain as EBL transformation [20]. There is, as of yet, no implementation, although Brunynooghe *et al.* provide a fairly detailed description of the transformation methodology.

In general, explanation based learning involves using a *specific* example to form a generalized "prototype" by substituting sub-terms for variables. Then by subsequently re-instantiating the "prototype" with target sub-terms we, hopefully, arrive at a solution sequence to the target problem.

Basically, Brunynooghe *et al.* suggest the removal of redundancies – repeated subcomputations – from programs by observing the behaviour of particular examples (i.e., the input is fixed).

Once again, the unfold/fold technique is used in the program transformation but by using a fixed input example to *control* the (un)folding, Brunynooghe *et al.* are able to automate (in principle) the crucial folding of subgoals in order to create new predicates. This automation is, however, also due in no small way to the system being limited to a very specific class of transformations, namely those which result in the removal of identical sub-computations (again, a consequence of the *automation trade-off* between the degree of automatability and the size of the class of transformations).

Brunynooghe *et al.* suggest working with logic programs (specifically Prolog). The initial *source* program for the EBL transformation of *Fibonacci* will be equivalent to the course of values definition of *section 2.12.* As far as the unfold/fold technique is concerned nothing is *essentially* gained by this although the declarative nature of the language does allow for easier manipulation and we can see exactly what is going on in the transformation process.

If we look at the computational tree, or *dependency graph*, for $fib(5)$ displayed

below, in **Fig.** 3, we can see that in order to evaluate $fib(5)$ numerous repeated sub-computations are performed (for example, a call on $fib(2)$ appears 3 times in the tree).

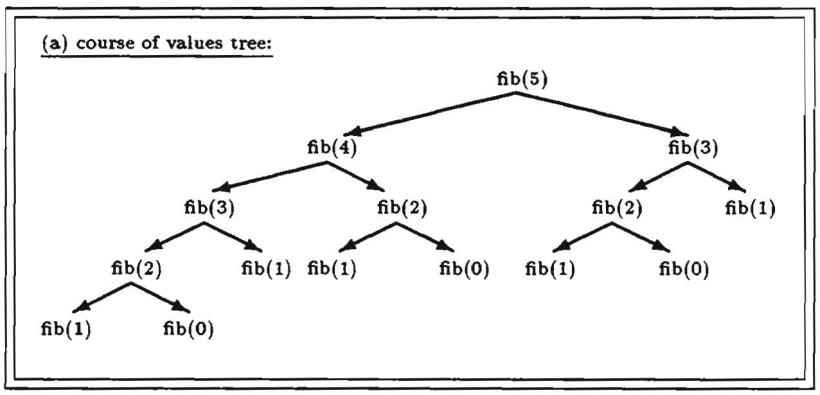

Figure 3: Computational tree for $fib(5)$ induced by (a) course of values induction, and, (b), stepwise induction

It is such *redundancy information* which guides the folding responsible for introducing new predicates.[16] So what the EBL transformation system does is to execute a query, in this case $fib(5)$, and then observe any duplication of subgoals in the computational tree. The repetition of $fib(3)$ is eliminated by adding its output, 2, as an extra output argument to the subgoal $fib(4)$. This is realised by a basic *unfold/fold* technique which employs the *eureka step* to allow for folding. By unfolding $fib(4)$ both occurrences of $fib(3)$ are obtained in one goal statement, and the undesired one is eliminated by factoring (full details are provided in [20].

2.2.2 Compiling Control transformation of Logic Programs using Partial Evaluation

Brunynooghe *et al.* have also partially implemented a transformation system [4]. The system is different from that outlined in [20] but also borrows from EBL techniques. EBL, effectively *partial evaluation*, is employed in a novel technique for avoiding the overhead caused by the execution of control languages when executing a program under the standard computation rule for *Prolog*. Hence, execution speed is increased by manipulation only of the computation rule under which a given Prolog program is executed and not by logical transformation like that between naïve and accumulating reverse.

This system is less versatile than the later system discussed above but is interesting since there is no cause for an *eureka step* during the transformation: the target program realizes a Prolog computation which is equivalent to a computation of the source program. However, no computation rule makes provision for lemma generation - i.e., we cannot provide clauses, additional to those describing the problem's logic, upon which folding can be performed in order to yield new predicates (and definitions), which in turn are used to introduce recursion into the target procedure. So, for example, this system can *not* improve, automatically or otherwise, on the course of values definition for Fibonacci.

The system operates by following two procedures:

[16] The term *redundancy information* is my own and was originally coined to describe the process of *pruning* of branches from a constructive proof tree which result in redundant computation (*cf.* [42]).

1. the production of a *symbolic trace tree*;

2. the production of a new program specialized only to admit the efficient execution of the program, even under the standard computation rule (the new program is referred to, by Brunynooghe *et al.*, as a *meta-interpreter*).

The processes required to realize 2 above hold much in common with *specialization* whereby we, in effect, partially evaluate a program's symbolic trace tree. The difference is that *specialization* amounts to a logical transformation of the program in question *and*, of course, we operate on synthesis proofs.

Similar in spirit to [20], the production of the symbolic trace tree involves improving the program by successively running it on a small number of *abstract* queries and inspecting the dependency information revealed by the (partially evaluated) traces. Each successive execution reveals new dependency information which causes a corresponding modification of – redundancy removal from – the preceding tree. Much responsibility rests with the user to make good query choices such that the eventual target abstraction covers all the data for which the program can succeed, and the resulting tree is a correct abstract representation for all possible successful executions of the program.

2.2.3 Translation of Clausal Specifications

W. F. Clocksin describes a technique for translating numerical algorithms, specified as clauses, into data-flow graphs. These graphs have the desirable property that common subexpressions are computed only once. The translation is not, strictly speaking, a program (nor proof) transformation but is of interest since the technique may be extended to provide a general program transformation technique: by fixing the input of an algorithm, and using *partial evaluation*, we obtain an example computational tree wherein repeated sub-computations can be observed. Such observations can then be used to control the development of the target algorithm. Indeed, this is precisely how Bruynooghe *et al.* arrive at their system, and is similar in spirit to the *specialization* of programs (§2.3 below).

2.3 Program Transformation Through Proof Transformation

We now turn our attention to systems which transform programs through transformations performed on synthesis proofs. The first such working system, as far as the author is aware, is Goad's *specialization* system.

In addition to surveying Goad's work, we also provide a brief account of a suggested system design, [58], for optimizing programs through the application of meta-level transformation operators.

2.3.1 Program Specialization Through Proof Transformation

Although perhaps not as well-known within the program transformation community as the influential work of Darlington *et al.*, [24] and [23] offer, the author believes, an equally pioneering body of research: the first working system which (necessarily) requires the use of proof transformation to achieve source to target program transformation. Goad, building upon the more theoretical work of Kreisel, [37, 38], demonstrates how proof transformations – specializations – can improve the efficiency of extracted programs in situations where a general purpose program is applied to inputs satisfying some given constraints.

The constraints are realized through partial evaluation on the input parameters of the program (dubbed *initialization*), and the transformations essentially consist in two pruning operations:

Normalization: which performs optimizations by the removal of any case split branch in the proof tree whose corresponding case condition evaluates to false (when evaluated by the initialization stage).

Dependency pruning: which performs optimizations by the elimination of case analyses – *cut elimination* – whose outcome was decided by formulae already assumed on the branch so far taken in the proof tree.

The use of proof transformations is essential since the functionality (input/output) of the specialized program in general may be different from the functionality of the original program, but they both satisfy the same specification (where, as usual, program transformations do not have a specification present, and hence transformations have to be restricted to those that preserve input/output behaviour).

The EBL approaches to program transformation, §2.2, provide the closest analogue to specialization: in both cases a target program is sought by (i) observing the (sub)goal inter-dependencies within a partially instantiated, and (ii) proof *pruning* any redundant proof (sub)trees accordingly. The main difference is that the goal of specialization is *not* a generalized solution but a target optimized for the *specific* input values chosen by the user.

Goad notes that constructive proofs of program specifications differ from straightforward programs in that more information is formalized in the proof than in the program, and that therefore 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", (p. 40 [23]).

A sizable amount of the extra information in the proofs is concerned with how to *efficiently* compute the input/output behaviour of the corresponding program.

• *Some Properties of Goad's System*

At the outset of [23] the following points are made:

1. The partial evaluation can be done on an incomplete proof with unproved lemmas without compromising the computational usefulness of the proof as a whole.

2. Although specialization followed by pruning is not *guaranteed* to decrease the *execution time* of an algorithm, it will do so most of the time simply because its purpose is to tailor algorithms (or proofs) to a specific task, or rather to a specific class of input. Pruning is, however, guaranteed to reduce the *size* of the algorithm.

3. Pruning is guaranteed to preserve the validity of an algorithm for the specification embodied in the root node of the proof describing the algorithm. That is, given the constructive proof and a partial evaluation, pruning is guaranteed to prune *only* the computationally redundant parts of the proof tree without effecting the input/output behaviour relative to the desired partial evaluation of the function parameters.

4. Conventional computational descriptions (such as the conditional form or some logic programming description) are *not* subject to the pruning transformations. This is because any valid transformations on conventional descriptions must preserve extensional meaning since they only contain information about the function to be computed. This is a nice illustration of the benefits of proof transformation as opposed to program transformation.

about the function to be computed. This is a nice illustration of the benefits of proof transformation as opposed to program transformation.

- *A Simple Example*

To illustrate specialization and pruning, Goad uses the following algorithm for computing an upper bound for both the sum and product of two positive rational numbers x and y:

- **(1)** $u(x, y) = x \leq 1$ then $(y + 1)$ else $(if\ y \leq 1$ then $(x + 1)$ else $2xy)$.

The algorithm specified above is in its *conditional* form and as such may only be slightly simplified as a result of partial evaluation. Suppose the value 0 is supplied for y, this results in the "specialized" conditional:

- **(2)** $u(x, 0) = x \leq 1$ then $(0 + 1)$ else $(if\ 0 \leq 1$ then $(x + 1)$ else $2x0)$,

which, upon evaluation, reduces to:

- **(3)** $u(x, 0) = x \leq 1$ then 1 else $(x + 1)$.

This simplification corresponds to the first stage of pruning: *normalization.*

The formalization of the upper bound algorithm as a *constructive existence proof*, its subsequent specialization (partial evaluation) and the application of normalization pruning then allows $u(x, 0)$ to be *automatically* simplified, by the use of dependency pruning, to the expression:

- **(4)** $(x + 1)$.

This is because the constructive existence proof will contain a case analysis whereby the case split is dependent on the size of x. Now, the fact that $(x + 1)$ is an upper bound for both $(x + 0)$ and $0 \times x$ does not depend on x being greater than one. This dependency information is contained in the proof and, via partial evaluation and pruning, allows the removal of the "computationally redundant" case split according to the size of x. Note that **(3)** and **(4)** are different functions (e.g. different input/output behaviour is observed for $x = 0.5$). However, as far as the partial evaluation (specialization) is concerned, in this case y being set to 0, subsequent *normalization* will preserve input/output behaviour. In other words whilst normalization will transform the algorithm, reducing its size, it is guaranteed to preserve the input/output behaviour. Dependency pruning, on the other hand, may change *both* the algorithm and the function.

The representation of the conditional expressions, (1) - (4), as natural deduction proofs, based upon the Prawitz natural deduction system, allows for the pruning transformations to be performed automatically [59].[17]

2.3.2 A Methodology For Program Through Proof Transformation Using Meta-Level Control

Pfenning has recently sketched a methodology for program transformation through proof transformation by using LF, [28], to formulate meta-theorems which can then be used as tactics for transforming object-level proofs [58]. This differs from our approach of using Prolog to specify the tactics. However, although no implementation

[17] We need not concern ourselves with the precise nature of the Prawitz natural deduction system since Goad in fact employs a deduction system only loosely based on the Prawitz system (see next section). The Prawitz natural deduction system can be viewed as similar to the sequent calculus without actually employing the sequent syntax (i.e expressions of the form $A \vdash B$). There are, of course, other differences, some of which do have computational significance. For these the reader should consult [59].

is yet available, Pfenning is in agreement with us that NUPRL type proof development systems would be particularly suited for providing the object-level proof structures.

Similar to the approach of the author's OMTS system, Pfenning suggests using higher order Martin-Löf type theory as a medium for program optimization. To quote from [58]:

> We present a methodology for deriving verified programs that combines theorem proving and transformation steps. It extends the paradigm employed in systems like NUPRL where a program is developed and verified through the proof of the specification in a constructive type theory.

The author has been in personal communication with Pfenning, primarily to determine whether or not his high-level design accords with the author's OYSTER implementation. Pfenning intends to operate within the same class of transformations as those performed by the OMTS: the introduction of a lemma followed by a number of *proof reductions*. Pfenning also regards the correctness guarantee of the source to target transformations, that is bought by virtue of the presence of a (complete) target specification, as a benefit of the proof transformation approach. Furthermore Pfenning observes that the meta-programs, or tactics, that apply the proof reductions can themselves be extracted from proofs of theorems which guarantee that the transformations are equivalence preserving.

Pfenning suggests using LF as the formal system for describing a logic. The operational interpretation of the tactics would be based on ideas from λ-Prolog [55].[18] The reason given for the choice of LF as the meta-logic, is that it has the expressive power to describe a wide class of transformations. This allows for the formal statement, and proof, of *meta-theorems*. The proof of a meta-theorem then may be used to transform proofs in the object logic. Pfenning holds that LF is better suited as a meta-logic than NUPRL since it is better equipped for deriving composite transformations from the definition of tactics.

A disadvantage of having distinct object and meta languages is that this can lead to a proliferation of complexity. For example, we cannot use the same unification algorithm for both inference and control reasoning.

It will, however, prove interesting in to see how Pfenning's approach of using two distinct languages, NUPRL for the object-language and LF for the meta-language, compares in practice to the approach taken with the OMTS. A foreseeable advantage of using an amalgamated logic, like LF, is that one avoids the proliferation of complexity that can occur with distinct object and meta-languages. For example, we cannot use the same unification algorithm for both inference and control reasoning. A disadvantage is that function/predicate definitions in an amalgamated logic may, if care is not taken, violate consistency (for example, by having truth predicates we may run foul of Russell's paradox).

2.3.3 Higher-order Logic Proof Transformation

P. Anderson has implemented and extended the basic ideas of Pfenning. The Elf higher-order programming language is used for the purposes of program optimization [1]. A partially verified implementation of support for the proofs-as-programs strategy (*c.f.* §2.3.4) and proof transformations is provided. Well known program transformations are expressed as (derived) rules of the Elf meta-logic. These rules can then be applied to proof constructs to realize the desired transformation. A stated advantage of Anderson's approach is the "integration between the flexibility

[18] λ-Prolog is a *higher-order* implementation of Prolog, such that it allows functions, or predicates, to be bound to variables, passed as parameters, and returned from function calls.

of program transformation and the formal connection between a program and it's specification of the proofs-as-programs methodology".

We shall not discuss Anderson's work in any greater detail since it shares with earlier work, surveyed below, the notion of exploiting the proofs-as-programs paradigm for the purposes of (correct) program transformation. However, it is as well to bear in mind that Anderson takes a different methodological stance than this earlier work: the proof transformation work surveyed in subsequent sections is concerned, primarily, with correctness and automation, where as Anderson is concerned with building a distinct, and transparent, layer of meta-level proof transformation rules. That is, a guiding factor in Anderson's system design is the seperation of transformation from heuristic concerns. This modualrity, albeit presently at the expense of automatability, is a very nice feature of Anderson's system, and allows for extensions, and augmentation, of the transformation rules without effecting the underlying logic system.

2.3.4 Optimizing Recursion Through Inductive Proof Transformation

For the readers benefit, we shall first digress briefly in order to provide a little introductory background to this, and the subsequent, section regarding formal methods, and their use in program development.

Background 1: Formal Methods A current 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 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 recent failure 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 general theorem proving techniques for automatic program optimization and synthesis. In both cases the target program is a significant improvement on the source (efficiency), and is guaranteed to satisfy the desired program specification (reliability).

Further applications of this research include the optimization of electronic circuit design and the optimization of computer configurations. This is because both these design problems can be formally cast as processes of inference [2, 39]. Thus, we can apply the same automated theorem proving techniques that we use for high quality software production.

Background 2: the Proofs as Programs Paradigm Exploiting the *Proofs as Programs Paradigm* for the purposes of program development has already been addressed within the AI community [32, 15]. 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 [16, 33].

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.[19] 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.

Program Optimization by Proof Transformation We have implemented a system for optimizing programs through the transformation of synthesis proofs [41, 42, 44, 48, 49]. These proofs are based on a Martin-Löf type theory logic and proved within the OYSTER proof refinement system (Martin-Löf, 1979; Martin-Löf, 1984).[20] The system 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 with synthesis and verification, knowledge of theorem proving, and in particular automatic proof guidance techniques, can be brought to bear on the transformation task. Furthermore, 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.

To accomplish program transformation *through* proof transformation, we have successfully, and for the first time, adapted a range of program transformation techniques to the proofs as program paradigm, notably: the *tupling* technique for "merging" repeated (sub)computations, [57] [11], and the unfold/fold technique for transforming inefficient functional programs into equivalent, more efficient, functional programs by a process of unfolding and folding definitions [18].

Synthesis proofs differ from straightforward programs in that more information is formalized in the proof than in the program: a description, or *specification*, of

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

[20] OYSTER is the Edinburgh Prolog implementation, and extension, of NuPRL; version "nu" of the *Proof Refinement Logic* system originally developed at Cornell [8],[32, 15].

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. In particular, dependency information abstracted from the source proof guides the transformations without the need for any extensive analysis of a programs recursive behaviour (such as the use of symbolic dependency graphs to analyse a programs recursive calling structure).

A key feature of our approach to program optimization 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. Such dualities offer the user a handle on the type, and efficiency, of recursive behaviour exhibited by the extracted algorithm.

The source and target programs of traditional program transformation systems do not have a formal specification present. This means there is no immediate means of checking that the target program meets the desired operational criteria. Regarding proof transformation, all transformed proofs yield programs that are correct with respect to the specification goals. By having a specification present we also ensure that the target computes the same specified input/output relation as the source, only more efficiently.

2.3.5 Program Optimization by Proof Synthesis

A substantial research project, at the Edinburgh University Department of AI, is involved with automating inductive theorem proving using a meta-level control paradigm called 'proof planning' [5]. Our proof planning system, CLAM, is able to prove a large number of inductive theorems automatically [7]. 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 [6, 52]. 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.

Both Goad and Pfenning use, or suggest using, the "proofs as programs" paradigm in order to exploit the properties of proofs so as to guide the transformation of (extract) programs. However, neither mentions exploiting the global property of inductive existence proofs, that they all pertain to a common structure or shape in order both to increase expectations of generality of, and as a guide for, the source to target transformation of inductive proofs.

Proof plans are being used to control the (automatic) synthesis of efficient functional programs, specified in a standard equational form, $\mathcal{E}$, by using the proofs as programs principle [29, 50, 51]. 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 the duality between induction and recursion, which forms one aspect of the *Curry-Howard isomorphism*, is exploited.

Program synthesis is viewed as the combination of verification and *middle-out reasoning*. 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.

More interesting however, is to introduce meta-variables into the proof to represent the unknown parts of an *improved* program (e.g., constraint functions). Proceeding in the same way, proof planning gradually instantiates these variables until a new, improved program is synthesized. Different proof plans provide different (inductive) proof structures, and hence different recursive extracts.

This approach has been found to be very promising. This is demonstrated by the success obtained in expressing a wide variety of well-known, but disparate, program improvement techniques within the proof planning framework [20]. For example, constraint-based transformation, generalization, fusion and tupling can be seen as proof planning. In addition, the work of Wadler, [65] and later Chin, [11], has been extended to encompass a larger class of functions that can be usefully optimized using the influential *deforestation technique* [66]. 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, and existential witnesses (which correlate to recursion schemata and recursive data types).

3 Summary

We began by discussing the unfold/fold technique for program transformation within the context of Darlington's NPL transformation system.

We abstracted a general program transformation strategy from the unfold/fold systems reviewed. The *eureka step* and the introduction of a *fold* during the target development were singled out as the main obstacles to automation: user guidance was required for lemma introduction, or the eureka step construction, and for guiding the sequence of unfoldings which follow in order to find a fold (and thereby introduce the new target recursion schema).

We made the observation that one factor that compounds the control problem is a result of the nature of many of the recursive equation re-writing systems that employ the unfold/fold technique: such systems (or usually the system's user) have to direct the unfolding towards finding a fold in order to attain the desired recursion schema.

A further problem which compounds the difficulty of automation is the extent of the class of transformations one wishes a system to encompass. This generally increases the burden on the human user in guiding lemma introduction and controlling unfolding. We called this the *automation tradeoff*.

We noted that, of those systems that ensure the correctness of their transformations, many employ equivalence preserving equality (identity) lemmas, within some suitable logic sub-set, thus avoiding any direct need for lengthy verification proofs. They do, however, require individual proofs for each lemma, *and* for each extension to the set of re-writes employed, further lemmas, with corresponding proofs, are required.

We paid particular attention to Darlington's use of *tupling* for removing redundancy by grouping together a collection of potentially re-usable function calls. We also discussed Chin's extensions to the tupling technique, in particular with regards to automation [11]. The important features of automatic tupling were the use of

dependency graphs in analysing repeated sub-computations.

We reviewed some of the influential program transformation systems in the literature. The majority of these employ some variant of the unfold/fold technique. However they also introduced new features such as some form of meta-level control, and the use of EBL techniques. Regarding the latter, partial evaluation is used to fix the input of a program. Repeated sub-computations are then removed from the *general* case by observing the behaviour of the partially evaluated particular example.

We provided a description and discussion of Goad's *specialization* system which adapts algorithms to particular situations through partial evaluation and pruning transformations performed on proofs. This enabled us to identify that part of the process which, if automation is a desirable goal, requires more information than the target program language provides. The missing information is *redundancy information.* Such information can be abstracted from the proof in the form of the dependency information between various proof (sub)goals and hypotheses (which is tantamount to the dependencies between facts involved in the computation of the proof extract program). We also made the observation that proof transformation provides a unique opportunity for transforming the functionality of a program whilst retaining the same specification. We noted, however, that, due to the system design, Goad does not exploit the nature of the proof specification language in order to verify the source to target transformations.

Apart from Goad, none of the existing systems approach program transformation through proof transformation. [Pfenning 89] has recently discussed how programs can be transformed by applying meta-level tactics to program synthesis proofs. This general idea has, prior to [Pfenning 89], been implemented by the author in the OYSTER proof development system [42].

References

Below is an extensive bibliography which aims to serve as a representative selection of the program transformation field. The reader can further follow up his/her interest by pursuing the references cited in the papers below.

---

[1] P. Anderson. Program Derivation By Proof Transformation. In D. Kapur, editor, *CADE12*, pages 446–461. Springer Verlag, 1994. Lecture Notes in Computer Science No. 607.

[2] David A. Basin. Extracting circuits from constructive proofs. In *Proceedings of the IFIP-IEEE International Workshop on Formal Methods in VLSI Design*, Miami USA, 1991. Also available from Edinburgh as DAI Research Paper No. 533.

[3] W. Bibel and K.M. Hörning. Lops — a system based on a strategical approach to program synthesis. In A. Biermann, G. Guiho, and Y. Kodratoff, editors, *Automatic Program Construction Techniques*, pages 69–90. MacMillan, 1984.

[4] M. Bruynooghe, D. De Schreye, and B. Krekels. Compiling control. *Journal of Logic Programming*, pages 135–162, 1989.

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

[6] A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill. Rippling: A heuristic for guiding inductive proofs. *Artificial Intelligence*, 62:185–253, 1993. Also available from Edinburgh as DAI Research Paper No. 567.

[7] 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.

[8] 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.

[9] 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.

[10] W. N. Chin. *Automatic Methods for Program Transformation.* PhD thesis, University of London (Imperial College), 1990.

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

[12] K. L. Clark. The synthesis and verification of logic programs. Research report (ccd), Imperial College, 1977.

[13] K.L. Clark. Predicate logic as a computational formalism. Report 79/59, Department of Computing, Imperial College, London, December 1979.

[14] K.L. Clark and J. Darlington. Algorithm classification through synthesis. *The Computer Journal*, 23(1):61–65, 1980.

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

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

[17] J. Darlington. *A Semantic Approach to Automatic Program Improvement.* PhD thesis, Dept. of Artificial Intelligence, Edinburgh, 1972.

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

[19] J. Darlington. A functional programming environment supporting execution, partial evaluation and transformation. In *PARLE 1989*, pages 286–305, Eindhoven, Netherlands, 1989.

[20] D. De Schreye and M. Bruynooghe. On the tranformation of logic programs with instantiation based computation rules. *Journal of Symbolic Computation*, (7):125–154, 1989.

[21] M. S. Feather. *A System For Developing Programs by Transformations.* PhD thesis, University of Edinburgh, 1979.

[22] A. B. Ferguson and P. Wadler. When will deforestation stop. In *Proc. of 1988 Glasgow Workshop of Functional Programming, Rothesay, Isle of Bute 1988*, pages 39–56, Feb 1988. Available as Glasgow University, Computer Science Dept Research Report 89/R4.

[23] C. A. Goad. Computational uses of the manipulation of formal proofs. Technical report, Stanford University, 1980. STAN-CS-80-819.

[24] C. A. Goad. Proofs as descriptions of computation. In *Lecture Notes in Computer Science.* Academic Press, 1980.

[25] P. W. Grant and J. Zhang. Heuristics for the automatic transformation of Prolog programs. In *UK IT 88 Conference Publication*, pages 135–139. UK IT 88 Conference (sponsors: SERC, DTI, MOD), 1988.

[26] I. Green. Flexible program transformation. Technical report, University of Cambridge, 1991.

[27] S. Gregory. Towards the compilation of annotated logic programs. Technical Report CCD no 80/16, Imperial College, London, 1980.

[28] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. In *Proc. of the Second Symposium on Logic in Computer Science*, 1987.

[29] J. Hesketh, A. Bundy, and A. Smaill. Using middle-out reasoning to control the synthesis of tail-recursive programs. In D. Kapur, editor, *11th Conference on Automated Deduction*, pages 310–324, Saratoga Springs, NY, USA, June 1992. Published as Springer Lecture Notes in Artificial Intelligence, No 607.

[30] C.J. Hogger. *Derivation of Logic Programs.* PhD thesis, Imperial College, 1980.

[31] C.J. Hogger. Derivation of logic programs. *JACM*, 28(2):372–392, April 1981.

[32] 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.

[33] 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.

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

[35] L. Kott. About a transformation system: A theoretical study. In *Proc. of 3rd Symposium on Programming*, pages 232–267, Paris, 1978. 3rd Symposium on Programming.

[36] R. Kowalski. Algorithm = Logic + Control. *Communications of ACM*, 22:424–436, 1979.

[37] G. Kreisel. Some uses of proof theory for finding computer programs. *Colloques Internationaux du Centre National de la Researche Scientifique*, 249:123–134, 1975.

[38] G. Kreisel. Some uses of proof theory for finding computer programs. *Recueil Des Travaux de L'Institut Mathematique*, 2:63–72, 1977.

[39] Helen Lowe. The Use of Theorem Proving Techniques in Expert Systems for Configuration. In J.-C. Rault, editor, *Proceedings of the Eleventh International Workshop on Expert Systems and their Applications, Avignon.* EC2, May 1991. Also available from Edinburgh as DAI Research Paper 536.

[40] P. Madden. The Lops approach to program synthesis. Research Paper 409, Dept. of Artificial Intelligence, Edinburgh, 1988.

[41] 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.

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

[43] 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.

[44] 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 from Edinburgh as DAI Research Paper 604.

[45] P. Madden. Recursive program optimization through inductive proof transformation. Research Paper 604, Dept. of Artificial Intelligence, Edinburgh, 1992. Submitted To JAR.

[46] P. Madden. A General Technique for Automatically Generating Efficient Programs Through the Use of Proof Planning. Research Paper, Dept. of Artificial Intelligence, University of Edinburgh, 1993. To appear in the proceedings of LOPSTR'93.

[47] P. Madden. An Evaluation of Deforestation Through Unfold/Fold Transformations, and a Comparison with Deforestation Through Proof-Planning. Technical Report in preperation, Dept. of Artificial Intelligence, University of Edinburgh, 1993.

[48] P. Madden. Formal methods for automated program improvement. In B. Nebel and L. Dreschler-Fischer, editors, *KI-94: Advances in Artificial Intelligence. Proceedings of 18th German Annual Conference on Artificail Intelligence*, Saarbrueken, Germany, September 1994. Springer-Verlag. A longer version is available from the Max-Planck-Institut as MPI-I-94-38.

[49] P. Madden. Recursive Program Optimization Through Inductive Synthesis Proof Transformation. *Journal of Automated Reasoning*, to appear.

[50] 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.

[51] 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. Springer Verlag. To Appear.

[52] P. Madden, J. Hesketh, I. Green, and A. Bundy. A general technique for automatically optimizing programs through the use of proof plans (extended abstract). In Y. Deville, editor, *Proceedings of LoPSTr'93*. Springer Verlag, 1993. Forthcoming: WICS series. The full paper is available as DAI Research Paper No. 608, Dept. of Artificial Intelligence, Edinburgh.

[53] Z. Manna and R. Waldinger. Knowledge and reasoning in program synthesis. Technical Note 98, SRI International, Menlo Park, November 1974.

[54] Z. Manna and R. Waldinger. A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems*, 2(1):90–121, 1980.

[55] 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.

[56] H. Nakayama. Program transformation under the principle of proofs as program. In ??, editor, *1991 International Logic Programming Symposium*, page ?? ??, 1991.

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

[58] F. Pfenning. Logic programming in the LF logical framework. In *Logical Frameworks*, pages 149 – 182. Cambridge University Press, 1991.

[59] D. Prawtiz. *Natural deduction: a proof-theoretical study*. Almqvist & Wiksell, Stockholm, 1965.

[60] M. Proietti and A. Pettorossi. Synthesis of Programs from Unfold/Fold Proofs. In Y. Deville, editor, *Proceedings of LoPSTr'93*. Springer Verlag, 1993. WICS series.

[61] M. Proietti and A. Pettorossi. Synthesis of Programs from Unfold/Fold Proofs. In Y. Deville, editor, *Proceedings of LoPSTr'93*. Springer Verlag, 1993. Forthcoming: WICS series.

[62] W. L. Scherlis. *Expression Procedures and Program Derivations*. PhD thesis, Stanford University, 1980.

[63] H. Tamaki and T. Sato. A transformation system for logic programs that preserves equivalence. Technical Report TR-018, ICOT, 1984.

[64] F. van Harmelen and A. Bundy. Explanation-based generalization = partial evaluation. *Artificial Intelligence*, 36(3):401–412, 1988. Also available as Edinburgh DAI Research Paper 347.

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

[66] G. A. Wiggins, A. Bundy, I. Kraan, and J. Hesketh. Synthesis and transformation of logic programs through constructive, inductive proof. In K-K. Lau and T. Clement, editors, *Proceedings of LoPSTr-91*, pages 27–45. Springer Verlag, 1991. Workshops in Computing Series.