Using abstraction to automate program improvement by transformation*

Ian Green
Department of Engineering
University of Cambridge
Cambridge CB2 1PZ
England
img@eng.cam.ac.uk

Abstract

The problem of automatically improving functional programs using Darlington's unfold/fold technique is addressed. Transformation tactics are formalized as methods consisting of pre- and post-conditions, expressed within a sorted meta-logic. Predicates and functions of this logic induce an abstract program property space within which conventional monotonic planning techniques are used to automatically compose methods (hence tactics) into a program improving strategy. This meta-program reasoning casts the undirected search of the transformation space as a goal-directed search of the more abstract space. Tactics are only weakly specified by methods. This flexibility is required if they are to be applicable to the class of generalized programs that satisfy the pre-conditions of their methods. This is achieved by allowing the tactics to generate degenerate scripts that may require refinement. Examples of tactics and methods are given, with illustrations of their use in automatic program improvement.

Introduction

The effectiveness of transformational models of programming is limited by the cost of controlling search in transformation space. One such transformational model is Darlington's generative set approach, based around unfold/fold transformations; this is a general technique for effecting efficiency-improving, correctness preserving transformations on functional programs expressed as recursion equations (Burstall and Darlington, 1977). In contrast to catalogue based transformation systems such as CHI (Green *et al.*, 1983) and TI (Balzer, 1981), the generative set style is characterized by a small number of simple transformations which are selected and applied to the program under development. The effect of a single transformation is typically negligible, nevertheless a significant improvement in program efficiency is possible if a number of them are arranged correctly.

Darlington's Functional Programming Environment supports the transformational development of Hope$^+$ programs, and is the vehicle for this work. The FPE (Sephton *et al.*, 1990) is a *transformation processor* which automatically applies externally generated transformation scripts to programs. The manual construction of program-improving scripts is time-consuming and difficult.

This paper reports current theoretical activity in automating script construction. We have formalized transformation tactics in a principled, flexible way using property-oriented program descriptions expressed within a sorted logic. The abstract space defined by this logical formalization can be searched in a goal directed way, permitting efficient reasoning with tactics in order to construct an overall program-improving strategy. We believe that goal directed program development is central to the success of transformation techniques.

The following example illustrates use of unfold/fold and serves to outline the problem of automatic program transformation.

Example problem. The unfold/fold technique consists of six elementary rules: define, instantiate, unfold, fold, abstract and replace. We assume the reader has some familiarity with the unfold/fold technique (see Burstall and Darlington (1977) for details), but detailed knowledge is not required. Briefly, unfold corresponds to replacing a function with its (suitably instantiated) body, and fold is the reverse of this. A replace transformation is a equality lemma such as the associativity of addition; define introduces a new equation with a unique left hand side; instantiate creates a substitution instance of an existing equation; abstract introduces a **let** clause. We shall base our presentation on natural numbers, tuples, and lists of natural numbers (lists are constructed with the infix *cons*, written '::' and nil, written '[]').

The following example program illustrates the use of unfold/fold. We see that *av* is intuitively defined in

*This research is funded by a SERC Research Studentship.

terms of *count* and *sum*.

$$(1) \quad av(l) \Leftarrow (count(l), sum(l));$$
$$(2) \quad count([]) \Leftarrow 0;$$
$$(3) \quad count(p :: ps) \Leftarrow 1 + count(ps);$$
$$(4) \quad sum([]) \Leftarrow 0;$$
$$(5) \quad sum(p :: ps) \Leftarrow p + sum(ps);$$

Using the unfold/fold technique we can derive by transformation an improved *av* shown in Figure 1 on the following page.

The improved *av* (shown below) is now an immediately recursive linear function which traverses its argument once, not twice.

$$av([]) \Leftarrow (0, 0);$$
$$av(p :: ps) \Leftarrow \textbf{let } (u, v) \equiv av(ps) \textbf{ in } (1 + u, p + v);$$

This example illustrates the need to control the search when deciding what transformation ought to be applied in order to arrive at this improvement. The aim of making *av* immediately recursive stems from one of two places: (1) certain program structures may be efficiently mapped onto certain machine architectures, and are desirable, or, (2) improvement of another function is dependant upon *av* being in this form; we address this more general case throughout this paper.

Our thesis is that AI planning techniques can be used to automate program transformation, but that a straightforward mapping of transformation patterns into, for example, STRIPS-like operators places a prohibitively large emphasis on object-level search. Instead, we describe transformations at the level of tactics that encapsulate pieces of programming knowledge, in much the same way that catalogue based transformation systems do. By weakly specifying these tactics they remain flexible and are scalable to larger problems. By reasoning with these tactic formalizations, sequences of tactics, called strategies, are generated that effect a global program improvement.

The remainder of this paper is organized as follows: the following section discusses previous work in automating program transformation and highlights aspects which influenced this work. Then we describe our formulation of a flexible approach to the transformation planning problem, using a sorted logic to represent program abstractions. Finally, three examples illustrate these ideas with a small selection of simple tactics and their formalizations.

Relevant work

Green's PSI project (Green, 1976) heralded the beginning of a number of catalogue-based transformation systems; PSI was entirely automatic, but experience with it underlined the difficulty of selecting suitable transformations from its large database. PSI's successor, CHI (Green *et al.*, 1983; Smith *et al.*, 1985), covered a broader spectrum of transformation problems by keeping the user in the transformation process. Closer to our work is the Glitter system, a component of which is Fickas' Jitterer (Fickas, 1980), capable of performing short, conditioning transformations on programs. This 'jittering' process was driven by the user who was responsible for selecting non-trivial transformations. In a slightly different way, Feather's ZAP system (Feather, 1982) partially automated the unfold/fold approach we use. ZAP allowed the user to express goals in terms of patterns which were then converted into transformation sequences automatically. With a higher-level approach, Chin (1990) and Smith's CYPRESS (Smith, 1985) have focussed on constructing tactics to manipulate programs in specific ways, but without an overall guiding strategy. Darlington's own efforts at partially automating his unfold/fold technique (Darlington, 1981) were not entirely successful because the heuristics embodied in his system failed to scale to larger problems, and were not flexible enough to be used uniformly. As a result, the reliance on pure search became prohibitive and this work was stopped.

Each of these low-level systems adopts an *ad hoc* approach to following a transformation strategy. With the exception of PSI, they need significant user involvement for realistic programs; indeed, the majority of them are interactive. Furthermore, with the exception of generative-set systems, scant regard is paid to correctness of the final program due to the difficulty of verifying the complex transformations present in catalogue-based systems.

Abstractions, Tactics and Methods

Abstractions

Input abstractions have been used to remove inessential detail from complex input domains in a object-level to object-level mapping (*e.g.*, clause set to clause set (Plaisted, 1981) or wff to wff (Tenenberg, 1988)). We have investigated this style of degenerate abstraction by constructing abstraction mappings from programs to programs: a program P is mapped to P' and a script is generated for P' by some means. In a refinement process this 'abstract' script guides the search for a 'concrete' script, that may be applied to improve P. A mapping similar to Plaisted's generalized propositional mapping has been used with some success on small problems but these mappings were not scalable to more complex programs. This is because the *properties* of programs which influence transformation cannot easily be expressed at the object-level (*i.e.*, as a program). For this reason we have chosen a more expressive, logical description which effectively generalizes programs if they satisfy some meta-level wff.

Tactics and Methods

A tactic is a sequence of transformations that exhibits a definite invariant structure commonly used in unfold/fold transformation and theorem proving. One of

$$
\begin{array}{lll}
av([]) \Leftarrow (count([]), sum([])); & & \text{instantiate } l \text{ to } [] \text{ in (1)} \\
\Leftarrow (0, sum([])); & & \text{unfold } count([]) \text{ using (2)} \\
\Leftarrow (0, 0); & & \text{unfold } sum([]) \text{ using (4)} \\
\\
av(p :: ps) \Leftarrow (count(p :: ps), sum(p :: ps)); & & \text{instantiate } l \text{ to } p :: ps \text{ in (1)} \\
\Leftarrow (1 + count(ps), sum(p :: ps)); & & \text{unfold } count(p :: ps) \text{ using (3)} \\
\Leftarrow (1 + count(ps), p + sum(ps)); & & \text{unfold } sum(p :: ps) \text{ using (5)} \\
\Leftarrow \textbf{let } v \equiv sum(ps) & & \\
\quad \textbf{in } (1 + count(ps), p + v); & & \text{abstract } sum(ps) \\
\Leftarrow \textbf{let } u \equiv count(ps) & & \\
\quad \textbf{in let } v \equiv sum(ps) & & \\
\quad\quad \textbf{in } (1 + u, p + v); & & \text{abstract } count(ps) \\
\Leftarrow \textbf{let } (u, v) \equiv (count(ps), sum(ps)) & & \\
\quad \textbf{in } (1 + u, p + v); & & \text{flatten lets (a replace)} \\
\Leftarrow \textbf{let } (u, v) \equiv av(ps) & & \\
\quad \textbf{in } (1 + u, p + v); & & \text{fold } (count(ps), sum(ps)) \text{ with (1)}
\end{array}
$$

Figure 1: Derivation of an improved version of av

the challenges in automating unfold/fold is the need to express the behaviour of these tactics in a formalization that is amenable to mechanized reasoning so that they may be organized into a global program improving strategy. After Bundy *et al.* (1988) we call these descriptions *methods*. In Feather's ZAP system, tactics are used extensively during program improvement, but no strategy for using those tactics is given. Furthermore, the tactics are expressed in terms of pattern-oriented transformations, *i.e.*, weak meta-level descriptions. We choose a more expressive logical description following Bundy *et al.* (1988) and Green *et al.* (1983).

A method is a 3-tuple consisting of a tactic, and pre- and post-conditions that formalize the selection and effect of that tactic on a program in such a way that:

- the method embodies in a formalized way the conditions required of a program in order that it is worth attempting the tactic, and,
- the method may be used to *predict* the effect of the tactic on program properties; in this way the actual tactic does not require execution in order to determine its effect on important program properties—an important computational saving.

Methods are similar to triangle tables in STRIPS (with MACROPS), and the abstract operators in (Tenenberg, 1988), but, unlike our methods, these are guaranteed to have specializations which succeed due to consistency restrictions imposed on their formation. The *skeletal plans* of MOLGEN which are refined by descending a user determined abstraction hierarchy, are similar to methods but methods are not part of an object-level hierarchy. This gives us greater expressive freedom in describing tactics, a limitation of ZAP's weak patterns and, as mentioned above, input abstractions.

In order to form tactics into a strategy, we perform simple STRIPS-like reasoning with methods. Methods whose post-conditions unify with the current goal are chosen according to some search policy and their pre-conditions are used as new goals. When all goals are satisfied the corresponing tactics are executed and a script is generated. In short, reasoning with methods is simple theorem proving; the advantage is that we are searching an abstract space which is a better space to search than the undirected transformation space.

The unfold/fold technique is monotonic in that new equations are always consistent with existing ones (we shall not discuss the possibility of losing total correctness); our formalization preserves this monotonicity hence we do not require a mechanism to circumvent the frame problem.

Flexible program transformation

Degenerate abstraction within the object-level are limited to the expressiveness of the object-level. Programs are not good representations of properties that influence transformation and so a meta-program logic is used to represent these properties. Using this logic we define methods that specify the behaviour of tactics.

By *partially* specifying a tactic with weak method conditions, the method generalizes a class of programs differing perhaps only in detail: the tactic must be applicable to this class. Tactics accommodate this flexibility by generating degenerate scripts which may require refinement before being passed to the FPE for execution. This is where the system gains much of its flexibility because the details of the script can be completed more easily when a basic structure is known.

These degenerate scripts provide subgoals which enable conventional means-end analysis and goal regression to be used to constrain search within object-level space, as in other degenerate object-level planners *e.g.*, ABSTRIPS and Plaisted's theorem prover (Plaisted, 1981).

It is possible that a refinement of a degenerate script cannot be found and so search must return to the meta-level by rejecting that (instance of the) tactic (backtracking) and re-planning. We are experimenting with pre- and post-condition strength to strike a balance between amount of backtracking, the cost of computing these conditions and the extra search incurred by increasing the number of methods.

Tactics, methods and examples

Here we present three tactics and methods, namely TUPLE, COMPOSE and FOLD. The detail of these and other tactics is ommited here due to space limitations, but may be found elsewhere (Green, 1991). The table below provides an informal interpretation of the predicates and functions used (in the examples, an underscore indicates an anonymous placeholder).

Predicate/Function	*Interpretation*
EQN(f, f_D)	$f \Leftarrow f_D$
OCCURS(f, g, pos)	f is a symbol appearing in g at position pos
TUPLE(t)	t is a tuple
FUNCTION(pos, f)	f contains a function symbol at position pos
ELEMENTS(t, f)	positions in t at which there are non-foldable expressions containing recursion argument of f
ARGS(f)	collection of arguments of f

A brief description of the tactic is followed by the method, shown as a table of pre- and post-conditions.

TUPLE tactic

This tactic is used to improve functions containing tuples whose elements are immediately recursive functions of the recursion parameter. The recursion argument is instantiated with [] and then unfolded. The constructor $(p :: ps)$ is used similarly, but additionally is, folded with the top-level function. Note that this fold may not be possible, in which case the tactic will fail.

TUPLE method

pre-conditions	*post-conditions*
EQN(f, f_D)	
$\wedge$ OCCURS(t, f_D, pos)	
$\wedge$ TUPLE(t)	
$\wedge$ $(\forall p \in$ ELEMENTS$(t, f))$	OCCURS$(f, f_D, _)$
(FUNCTION(p, f_D)	
$\wedge$ OCCURS(h, f_D, p)	
$\wedge$ EQN$(h, h_D) \wedge$ OCCURS$(h, h_D, _))$	

COMPOSE tactic

Removes nested function calls from expressions with define; a new function symbol is introduced on the left of an equation whose right hand side is this nested call. A fold with this new equation removes the nested call from the expression. A more general case of this, fusion (or deforestation), is presented by Chin (1990).

COMPOSE method

pre-conditions	*post-conditions*
EQN(f, f_D)	
$\wedge$ EQN(g, g_D)	
$\wedge$ OCCURS$(g, g_D, _)$	FUNCTION(pos, f_D)
$\wedge$ OCCURS(f, f_D, pos)	$\wedge$ OCCURS(f', f_D, pos)
$\wedge$ OCCURS$(g,$ ARGS$(f), _)$	$\wedge$ EQN(f', f'_D)
$\wedge \neg$OCCURS$(f,$ ARGS$(f), _)$	

FOLD tactic

Performs the classic unfold/fold sequence on nested functions (Burstall and Darlington, 1977) in which recursion parameters are instantiated to type constants and constructors followed by repeated unfolding and unfolding/folding, respectively.

FOLD method

pre-conditions	*post-conditions*
EQN$(h, f(g(x)))$	
$\wedge$ EQN(g, g_D)	OCCURS$(h, h_D, _)$
$\wedge$ EQN(f, f_D)	
$\wedge$ OCCURS(g, g_D, pos)	

Examples

The following three examples illustrate how the above tactics can be reasoned with, as described, in order to automatically improve a number of functions. Examples 1 and 2 require only tactics FOLD and TUPLE respectively; example 3, which requires all three tactics, illustrates how meta-level reasoning chains tactics together into an overall transformation strategy. In these last two examples the generated scripts require refinement, the details of which are not presented here.

Example 1. Let us derive an immediately recursive version of *foo* where *foo* is defined by:

$$foo(l) \Leftarrow sum(squares(l));$$
$$sum([]) \Leftarrow 0;$$
$$sum(p :: ps) \Leftarrow p + sum(ps);$$
$$squares([]) \Leftarrow [];$$
$$squares(p :: ps) \Leftarrow (p * p) :: squares(ps);$$

An immediately recursive form of *foo* is represented by the meta-level description

$$\text{EQN}(foo, foo_D) \wedge \text{OCCURS}(foo, foo_D, \_).$$

The first conjunct is satisfied, the second is not. A method is found whose post-conditions unify with this

unsatisfied goal. Both the TUPLE and FOLD methods fulfil this requirement and so this is a backtracking point. FOLD is chosen as its pre-conditions are satisfied. The tactic is executed and the following script is generated:[1]

```
type constant ([])
 instantiate l to []
 unfold squares([])
 unfold sum([])
type constructor (p :: ps)
 unfold squares(p :: ps)
 unfold sum((p * p) :: squares(ps))
 fold foo(l)
```

which does not require refinement. When applied to *foo* this yields:

$$foo([]) \Leftarrow 0;$$
$$foo(p :: ps) \Leftarrow (p * p) + foo(ps);$$

Example 2. Consider the introductory example with an additional parameter x added to av:

$$av(l, x) \Leftarrow (count(l), x + sum(l));$$

With the goal

$$\text{EQN}(av, av_D) \wedge \text{OCCURS}(av, av_D, \_)$$

the TUPLE method indicates that this tactic is to be tried. As there are no other goal to satisfy, the tactic is executed resulting in the script shown in Figure 2a.

```
type constant ([])
 instantiate l to []
 unfold count([])
 unfold sum([])

type constructor (p :: ps)
 instantiate l to p :: ps
 unfold count(p :: ps)
 unfold sum(p :: ps)

 fold av(l, x)
```

(a)

```
instantiate l to []
unfold count([])
unfold sum([])

instantiate l to p :: ps
unfold count(p :: ps)
unfold sum(p :: ps)
replace x + (p + sum(ps))
 with (p + sum(ps)) + x
replace (p + sum(ps)) + x
 with p + (sum(ps) + x)
replace sum(ps) + x
 with x + sum(ps)
abstract count(ps)
abstract x + sum(ps)
replace (flatten lets)
fold av(l, x)
```

(b)

Figure 2: Scripts for *foo* before (a), and after (b) refinement

This script would fail as the fold $av(l, x)$ cannot succeed on

$$av(p :: ps, x) \Leftarrow (1 + count(ps), x + (p + sum(ps)));$$

[1]Scripts shown are simplified for clarity.

In order to avoid failure of this kind the pre-conditions of methods could be strengthened. However, it is not clear how this would be implemented, and evaluation of these conditions is likely to be computationally expensive. But more importantly it would make the TUPLE method more rigid and specialized requiring more tactics and methods to handle anomalous cases.

When the script fails in this way it is used to guide search for a suitable refinement. Means-end analysis and backward reasoning from the goal of folding av results in a number of replace transformations which complete the script as shown in Figure 2b above.

The meta-level reasoning in this case gives us better than 10:1 (object-level:meta-level) gearing on the number of steps needed, as well as reducing search needed to discover them. The improved av is:

$$av([], x) \Leftarrow (0, x + 0);$$
$$av(p :: ps, x) \Leftarrow \textbf{let } (u, v) \equiv av(ps, x) \textbf{ in } (1 + u, p + v);$$

Example 3. In this example we illustrate the way a transformation strategy is automatically constructed from tactics by searching in the abstract space.

$$foo(l, x) \Leftarrow (count(l), x + sum(l), sum(squares(l)));$$
$$sum([]) \Leftarrow 0;$$
$$sum(p :: ps) \Leftarrow p + sum(ps);$$
$$squares([]) \Leftarrow [];$$
$$squares(p :: ps) \Leftarrow (p * p) :: squares(ps);$$

The goal is for *foo* to be immediately recursive which is expressed at the meta-level as

$$\text{EQN}(foo, foo_D) \wedge \text{OCCURS}(foo, foo_D, \_).$$

The post-conditions of TUPLE unify with this goal but its pre-conditions are unsatisfied. We seek to satisfy its unsatisfied pre-condition, which is:

$$(\forall p \in \text{ELEMENTS}(t, foo))\ (\text{FUNCTION}(p, foo_D) \wedge \text{OCCURS}(h, f_D, p) \wedge \text{EQN}(h, h_D) \wedge \text{OCCURS}(h, h_D, \_))$$

(We abbreviate the right hand side of *foo* as foo_D.) This pre-condition fails because $sum(squares(l))$ is not a function symbol. The COMPOSE method achieves this goal by defining a new function, *sumsquares*, and folding it into foo_D. Choosing this method leaves

$$\text{OCCURS}(sumsquares, sum(squares(l)), \_)$$

unsatisfied so we seek to derive a new right hand side to *sumsquares* that contains *sumsquares*. The post-conditions of the FOLD and TUPLE methods unify with this goal. The FOLD tactic is chosen as its pre-conditions are satisfied. Now the entire pre-condition of the TUPLE method is satisfied and so the TUPLE tactic may be applied.

This strategy deployed the tactics **COMPOSE**, **FOLD** and **TUPLE** in a simple sequence. Note that this sequence was obtained automatically, was goal-directed and required only these three tactics. When these tactics are executed the script they generate is refined and would be executed by the FPE to produce the improved program shown below. This refined script consists of the following transformations: 1 define, 4 instantiate, 10 unfold, 2 replace, 3 fold and 3 abstract; a total of 23 object-level transformations.

$$foo([], x) \Leftarrow (0, x + 0, 0);$$
$$foo(p :: ps, x) \Leftarrow \textbf{let } (u, v, w) \equiv foo(ps, x) \textbf{ in } (1 + u, p + v, (p * p) + w);$$

Conclusions and further work

This work investigates the use of AI planning techniques in the automatic improvement of functional programs using the unfold/fold technique. By formalizing programming knowledge in a abstract way using methods, reasoning about programs takes place above the program level in an abstract space in which search is more efficient. Reasoning in this way automatically builds sequences of tactics into a global program improving strategy. Furthermore, the weak nature of methods ensures their flexibility and applicability to a wide-class of programs. Due to this weak specification, tactics generate scripts which may require refining before they are applied.

We are developing these ideas in a number of orthogonal directions including adding new tactics and their methods, experimenting with method pre- and post-condition strength to assess effects on backtracking, cost of condition evaluation and number of tactics. Complexity analysis will be investigated as a possible heuristic evaluation function to bias choice between competing tactics.

Acknowledgements

The author would like to thank Dr. Tony Holden and Professor John Darlington for comments.

References

(Balzer, 1981) Robert Balzer. Transformational implementation: An example. *IEEE Transactions on Software Engineering*, 7(1):3–14, 1981.

(Bundy *et al.*, 1988) Alan Bundy, Frank van Harmelen, Jane Hesketh, and Alan Smaill. Experiments with proof plans for induction. Technical Report 413, Department of AI, Edinburgh, 1988.

(Burstall and Darlington, 1977) Rod M. Burstall and John Darlington. A transformation system for developing recursive programs. *JACM*, 24(1):44–67, 1977.

(Chin, 1990) Wei Ngan Chin. *Automatic Methods for Program Transformation*. Ph. D. thesis, Imperial College, University of London, March 1990.

(Darlington, 1981) John Darlington. An experimental program transformation and synthesis system. *Artificial Intelligence*, 16:1–46, 1981.

(Feather, 1982) Martin S. Feather. A system for assisting program transformation. *ACM TOPLAS*, 4(1):1–20, 1982.

(Fickas, 1980) Stephen Fickas. Automatic goal-directed program transformation. In *Proceedings of the 1st Annual National Conference on Artificial Intelligence*, pages 68–70, Stanford, California, USA, August 1980.

(Green *et al.*, 1983) Cordell Green, David Luckham, Robert Balzer, Thomas Cheatham, and Charles Rich. Report on a knowledge-based software assistant. Technical report, Kestrel Institute, 1801 Page Mill Road, Palo Alto, California, USA, 1983.

(Green, 1976) Cordell Green. The design of the PSI program synthesis system. In *Proceedings of the 2nd International Conference on Software Engineering*, San Francisco, California, USA, 1976.

(Green, 1991) Ian M. Green. A review of program transformation tactics, with examples. Technical Report CUED/F-INFENG/TR.61, Department of Engineering, University of Cambridge, 1991.

(Plaisted, 1981) David A. Plaisted. Theorem proving with abstraction. *Artificial Intelligence*, 16:47–108, 1981.

(Sephton *et al.*, 1990) Keith M. Sephton, Wei Ngan Chin, L. M. J. McLoughlin, H. M. Pull, and R. L. While. *Using "tuples": The Flagship Programming Environment*. Functional Programming Section, Imperial College, October 1990.

(Smith *et al.*, 1985) Douglas R. Smith, G. B. Kotik, and Stephen J. Westfold. Research on knowledge-based software environments at kestrel institute. *IEEE Transactions on Software Engineering*, 11(11):1278–1295, 1985.

(Smith, 1985) Douglas R. Smith. Top-down synthesis of divide and conquer algorithms. *Artificial Intelligence*, pages 43–96, 1985.

(Tenenberg, 1988) Josh Tenenberg. *Abstraction in Planning*. Ph. D. thesis, Department of Computer Science, University of Rochester, May 1988.