Induction and Synthesis for Automatic Program Transformation*

Françoise Bellegarde,
Oregon Graduate Institute of Science & Technology
PO Box 91000 Portland, Oregon, USA
503-690-1558
bellegar@cse.ogi.edu

The transformational approach to the development of programs is attractive for writing small components of large software systems. In this approach, developing a software component consists simply of writing an initial, possibly inefficient, but correct program P_0 and, then, in transforming P_0 into a sequence of programs $P_1, P_2, \ldots, P_n$ to get a new, semantically equivalent program P_n which is more efficient. The transformation phase, to be effective, must be automated so that it is not necessary to be an expert in transformation strategies to use the transformational approach for software design. In our work, induction, synthesis and generalization are used as mechanisms for an automated system for transformation of functional programs. Synthesis of a new program version is done by a completion procedure [4]. The programs in the sequence $P_0, P_1, \ldots, P_n$ are presented by a terminating, constructor-based and orthogonal first-order rewrite system. For example, the transformation strategies that are automated are: *fusion or deforestation* (eliminating useless intermediate data structures), *tupling* (consolidation of similar control structures) and *accumulator introduction* (recursion elimination). Presently, the following mechanisms control or enhance the completion procedure.

(1) There is a *mechanism for suggesting strategies* which is accomplished by automatically introducing new functions, called *eurekas* in the fold-unfold methodology [2] (previously these new functions had to be introduced through the insight of a clever user). Consider, for example, that the transformation step P_i to P_{i+1} performs deforestation of a term t in P_i. The transformation consists in looking for a set of rewrite rules S that reduce t to a term t' free of intermediary data structures. The mechanism for suggesting strategies is able to propose a left-hand side s of a rule in S which encompasses t and which contains no useless data structures. The right-hand side is built with a new functional symbol h which has as arguments all the variables in s. Rules in S are called synthesis rules.

A synthesis rule triggers completion process. The completion synthesizes the constructor-based rules, forming a complete definition of the function corresponding to the new functional symbol h.

(2) There is a *mechanism to control the production of critical pairs* during completion. For example, during a deforestation, the control mechanism rejects critical pairs between a

*The work reported here is supported in part by a contract with Air Force Materiel Command (F1928-R-0032)

constructor-based rule and a *synthesis rule* that do not substitute a constructor-term in an inductive position of the new symbol h. The control guarantees the termination of completion.

(3) There is a *mechanism for suggesting induction lemmas* during the synthesis. For example, associativity laws facilitate recursion removal synthesis, and endomorphisms or distributivity laws facilitate fusion synthesis. In such cases, the suggestion mechanism is followed by a proof or a disproof of the lemma. If the proof succeeds, the result can be exploited by the synthesis. If the proof fails, it knows that the transformation cannot be done and the eureka can be rejected.

There is a problem when the suggestion is incomplete. For example, recursion removal is often achieved by finding an identity. But we don't know if this identity exists or not. If the identity does not exist, the narrowing technique will search forever for it. Future work could solve this problem in some cases, possibly by utilizing techniques presented in [1].

Fortunately most induction lemmas do not need to be known completely. Suppose a fusion synthesis finds the critical pair: $h(x :: xs) = length(x@flat(xs))$, an interesting lemma is $flat(x@y) = flat(x)@flat(y)$. However suggesting the completion to synthesize a function k such that $length(x@u) \rightarrow k(x, length(u))$ is enough to get what we want: $h(x :: xs) \rightarrow k(x, h(xs))$. In such cases, the rules synthesized for k by the completion are not quite constructor-based but are *generalizable to constructor-based rules*.

(4) Cleaning mechanisms follow the synthesis. They build a constructor-based rewrite system from the rewrite system issued from the synthesis. Cleaning uses deletion, generalization, normalization, and specialization techniques.

(5) Finally the order in which strategies are chosen matters. Therefore, a hybrid mechanism to suggest how to combine strategies is required.

It can be proved that the process preserves the termination of a constructor-based orthogonal rewrite system. We guarantee that each distinct synthesis terminates. Moreover, we also guarantee that the process does not call for an infinite number of syntheses.

Related approaches to automating elimination of useless intermediate data structures are the deforestation algorithm proposed by Wadler [6] and the extended deforestation algorithm proposed by Chin [3]. Also an automatic way to implement deforestation inside the Haskell's compiler has been shown in [5]. By using completion[1] as a tool for synthesis, our transformation system is not restricted to the deforestation strategy and is able to automate transformations for a larger class of programs than the above algorithms.

References

[1] J. Arsac and Y Kodratoff. Some techniques for recursion removal from recursive functions. *ACM Transaction on Programming Languages and Systems*, 4,(2):295-322, 1982.

[2] R. M. Burstall and J. Darlington. A Transformation System For Developing Recursive Programs. *Journal of the Association for Computing Machinery*, 24, pages 44-67, 1977.

[3] W. N. Chin. *Safe Fusion of Functional Expressions. Proc. of the Conference on Lisp and Functional Programming*, San Francisco, 1992.

[4] N. Dershowitz. Completion and its Applications. *Resolution of Equations in Algebraic Structures,*, 2, pages 31-86, Academic Press, 1988.

[5] A. Gill, J. Launchbury and S.L. Peyton Jones. A short cut to Deforestation. *Proc. of the 6th Conf. on Functional Programming Languages and Computer Architecture*, Copenhagen, pages 223-232, June 1993.

[6] P. Wadler, Deforestation: Transforming programs to eliminate trees. *ESOP'88*. LNCS 300, 1988.

[1] Presently our prototype automates deforestation and tupling.