Program Specialization via Algorithmic Unfold/Fold Transformations

Alberto Pettorossi
Department of Informatics, Systems, and Management, University of Rome II,
Via della Ricerca Scientifica, 00133 Roma, Italy. adp@iasi.rm.cnr.it
and
Maurizio Proietti
IASI-CNR, Viale Manzoni 30, 00185 Roma, Italy. proietti@iasi.rm.cnr.it

---

We briefly analyze the relationship between partial evaluation and unfold/fold program transformation. These two techniques have some common objectives, but they have been developed according to different methodologies. As a promising direction to take for the future, we propose to embed partial evaluation in the richer framework of the unfold/fold program transformation technique. We also propose the use of *algorithmic strategies*, that is, mechanically generated sequences of transformation rules, for obtaining high quality specialized programs in a fully automatic way. We finally indicate some features of the program specialization system that may be designed according to our proposal.

Two well established techniques in the field of automatic program derivation are partial evaluation (see [Jones et al. 1993] for a comprehensive survey and also [Danvy et al. 1996; Gallagher 1993; Komorowski 1982; Lloyd and Shepherdson 1991]) and unfold/fold program transformation [Burstall and Darlington 1977; Pettorossi and Proietti 1994; Tamaki and Sato 1984]. For simplicity, in this paper we will essentially refer to these techniques as they have been developed in the fields of functional and logic programming.

Partial Evaluation and Unfold/Fold Program Transformation

Partial evaluation consists in transforming a given program by taking advantage of the knowledge of part of the input data, called the *static input*. Before program execution some calculations depending on the static input are performed, thereby deriving a *residual* program which computes the same output as the original program when the rest of the input data, called the *dynamic input*, is provided.

Partial evaluation is a technique for *program specialization*. Here by program specialization we mean the goal of adapting a program to a given context of use and we do not actually refer to the operations to be performed to achieve that goal (this terminology follows [Gallagher 1993; Consel 1997] and slightly departs from [Jones et al. 1993, page 1], where partial evaluation is identified with program specialization). Thus, partial evaluation realizes program specialization where the context of use is given by the value of the static input.

Unfold/fold program transformation is a technique for stepwise program derivation where an elementary, semantics preserving transformation rule is applied at each step, so as to derive a new program version from an old one. The set of transformation *rules* includes the following ones:

(i) *definition*, which is the introduction of a new function (or predicate) definition,

(ii) *unfolding*, which is the substitution of a function (or predicate) call by the corresponding body (it is a step of symbolic evaluation),

(iii) *folding*, which is the substitution of an instance of a function (or predicate) body by the corresponding call (it is the inverse of unfolding),

(iv) *instantiation*, which is the introduction of an instance of an existing equation (or clause), and

(v) *replacement*, which is the substitution of an expression (or a goal) by a semantically equivalent expression (or goal).

Transformation *strategies* are used to drive the application of the above rules.

Common Concerns and Similarities

When deriving programs both partial evaluation and unfold/fold program transformation address the following issues:

(i) *Correctness*: the derived programs should have the same semantics as the initial programs,

(ii) *Complexity*: the derived programs should run within less time or space w.r.t. the initial programs, and

(iii) *Automation*: the process of program derivation should be supported by suitable automatic tools.

Some similarities between partial evaluation and unfold/fold program transformation have been recognized and formalized during the last two decades (see [Ershov 1982, page 59], [Jones et al. 1993, page 347], [Pettorossi and Proietti 1994, page 308]). In particular, in the case of functional and logic programming, the basic program manipulations performed by partial evaluation can be viewed as unfold/fold transformation steps with the following restricted rules: (i) *definition*, where the new function (or predicate) definition should be an instance of an already existing one, (ii) *unfolding*, and (iii) *folding*, where one should replace a function (or predicate) call with instantiated arguments by a new function (or predicate) call.

Different Features

Partial evaluation and unfold/fold program transformation also show some differences, which were particularly noticeable during the initial development of these two techniques. Among these differences let us mention the following ones.

(i) Partial evaluation is used to perform program specialization and one important area of its application is automatic compiler generation, while unfold/fold transformation is used for a wider spectrum of program manipulations, including the generation of programs from specifications and also program synthesis.

(ii) Partial evaluation is intended to be *algorithmic*, and indeed it achieves the goal of performing program specialization in a fully automatic way by applying several subsidiary techniques, such as binding-time analysis [Jones et al. 1993]. On the contrary, unfold/fold transformations are often human-directed and strategy-driven, and may also require a heuristic search similar to the one performed by theorem provers.

(iii) Standard methods for partial evaluation often allow for improvements of program efficiency which are less substantial than those which can be obtained by using unfold/fold transformations.

Algorithmic Unfold/Fold Strategies

In the last few years several techniques have been developed for enhancing partial evaluation and performing fully automatic unfold/fold program transformations. Among these techniques we recall: (i) positive supercompilation [Sørensen et al. 1996] which is a restricted, automatic version of supercompilation [Turchin 1986], (ii) conjunctive partial deduction [Leuschel et al. 1996], which performs partial evaluation of logic programs using conjunctions of several atoms together, (iii) generalized partial computation [Futamura and Nogi 1988] that incorporates some theorem proving capabilities, and its fully automatic version proposed in [Takano 1991], (iv) deforestation [Wadler 1988], which is an algorithm for eliminating intermediate data structures, (v) tupling [Pettorossi 1977; Chin 1993], which avoids the repeated computation of common subexpressions and multiple traversals of data structures, (vi) UDF* [Proietti and Pettorossi 1995], which eliminates unnecessary data structures from logic programs, (vii) the strategy proposed in [Pettorossi et al. 1997], which avoids redundant nondeterminism while specializing logic programs, and (viii) the techniques of [Pettorossi and Proietti 1996] for specializing programs when their context of use is given by properties, rather than values, of the input data.

We believe that an important topic for research in the forthcoming years is the development of a unifying framework for comparing, combining, and extending the above mentioned techniques. We also believe that the core of this framework should be provided by unfold/fold rules and strategies. Indeed, the techniques we have mentioned above can all be reformulated as strategies for the application of the unfold/fold transformation rules. In order to get fully automatic program specializers we want the strategies to be *algorithmic*, in the sense that very little search and no human intervention should be required during their execution.

The framework we propose has already demonstrated its flexibility and power for the definition of new strategies for program specialization. Indeed, in [Pettorossi et al. 1997] it has been shown that partial evaluation can be enhanced by an algorithmic strategy which employs the subsumption rule and case-split rule together with the usual definition, unfolding, and folding rules. In order to achieve efficient specialized programs, this algorithmic strategy eliminates unnecessary data structures and redundant nondeterminism, and in particular, it can derive a Knuth-Morris-Pratt matching algorithm by specializing w.r.t. a given input pattern P the following nondeterministic specification:

$$match(P, S) \leftarrow append(Left, P, I),\ append(I, Right, S)$$

which states that P occurs in a string S iff there exist two strings *Left* and *Right* such that S is the concatenation of *Left*, P, and *Right*. The derivation of a similar efficient matcher would have been impossible by partial evaluation alone.

A Transformation System Based on Algorithmic Unfold/Fold Strategies In order to use the transformation methodology for program specialization that we have now suggested, it is necessary to construct a system which supports the design and execution of algorithmic unfold/fold strategies. Here are some features of the system we propose.

(i) *User-defined rules and strategies.* The system should allow the user to introduce and apply different sets of transformation rules, strategies, and higher-order strategies. Higher-order strategies should make it possible to combine already defined strategies and also to dynamically generate new strategies. The need for higher-order manipulations has already emerged both in partial evaluation, where the concepts of generating extension and metasystem transition play an important role [Ershov 1982; Glück 1996; Turchin 1980], and in unfold/fold program transformation where, for instance, suitable generalization strategies may be applied after analyzing the outcome of other strategies [Pettorossi and Proietti 1994]. A theoretical issue to be addressed is the *completeness* of various sets of rules and strategies with respect to given classes of programs, that is, the capability of those rules and strategies to derive programs with certified levels of efficiency when specializing programs of the given classes.

(ii) *Language parametricity.* The system should be usable for different programming languages, that is, it should be parametric w.r.t. the syntax and semantics of the language in use, and also w.r.t. the calculus for establishing the computational complexity of the programs written in that language (for such a calculus see, for instance, [Vitter and Flajolet 1990]).

(iii) *Analysis tools.* The system should allow for the use of static analysis techniques, possibly based on abstract interpretation, such as those for binding-time analysis, mode analysis, and type analysis. The information collected by these techniques is used to direct the application of sophisticated specialization strategies as indicated in [Consel and Khoo 1993; Gallagher 1993; Jones et al. 1993; Pettorossi et al. 1997]. In order to enhance the strategies for program specialization some algorithmic forms of theorem proving may also be added to the system, suitably restricting the approaches of [Bossi et al. 1990; Futamura and Nogi 1988].

(iv) *Auxiliary transformation techniques.* The system should allow for the application of various transformation techniques which when used together with program specialization, often improve the performance of programs. Among these techniques we here recall: continuation-based transformations [Wand 1980], finite differencing [Paige and Koenig 1982], schema transformations (see [Partsch 1990] for a survey), and changes of data structures [Pettorossi and Proietti 1994].

REFERENCES

Bossi, A., Cocco, N., and Dulli, S. 1990. A method for specializing logic programs. *ACM Transactions on Programming Languages and Systems 12*, 2 (April), 253–302.

Burstall, R. M. and Darlington, J. 1977. A transformation system for developing recursive programs. *Journal of the ACM 24*, 1 (January), 44–67.

Chin, W.-N. 1993. Towards an automated tupling strategy. In *Proceedings of PEPM '93, Copenhagen, Denmark* (1993), pp. 119–132. ACM Press.

Consel, C. 1997. Program adaptation based on program transformation. In *Proceedings of the Workshop on Strategic Directions in Computing Research, MIT, Cambridge, MA, USA, June 14-15, 1996* (January 1997), pp. 73–75. ACM Sigplan Notices 32/1.

Consel, C. and Khoo, S. 1993. Parameterized partial evaluation. *ACM Transactions on Programming Languages and Systems 15*, 3, 463–493.

Danvy, O., Glück, R., and Thiemann, P. Eds. 1996. *Partial Evaluation*, Volume 1110 of *Lecture Notes in Computer Science* (1996). Springer-Verlag.

Ershov, A. P. 1982. Mixed computation: Potential applications and problems for study. *Theoretical Computer Science 18*, 1, 41–67.

Futamura, Y. and Nogi, K. 1988. Generalized partial computation. In D. Bjørner, A. P. Ershov, and N. D. Jones Eds., *Partial Evaluation and Mixed Computation* (1988), pp. 133–151. North-Holland.

GALLAGHER, J. P. 1993. Tutorial on specialization of logic programs. In *Proceedings of PEPM '93, Copenhagen, Denmark* (1993), pp. 88–98. ACM Press.

GLÜCK, R. 1996. On the mechanics of metasystem hierarchies in program transformation. In M. PROIETTI Ed., *Proceedings of LOPSTR '95, Utrecht, The Netherlands*, Lecture Notes in Computer Science 1048 (1996), pp. 234–251. Springer-Verlag.

JONES, N. D., GOMARD, C. K., AND SESTOFT, P. 1993. *Partial Evaluation and Automatic Program Generation.* Prentice Hall.

KOMOROWSKI, H. J. 1982. Partial evaluation as a means for inferencing data structures in an applicative language: A theory and implementation in the case of Prolog. In *Proceedings 9-th POPL, Albuquerque, New Mexico, USA* (1982), pp. 255–267.

LEUSCHEL, M., DE SCHREYE, D., AND DE WAAL, A. 1996. A conceptual embedding of folding into partial deduction: Towards a maximal integration. In M. MAHER Ed., *Proceedings of the Joint International Conference and Symposium on Logic Programming, Bonn, Germany* (1996), pp. 319–332. MIT Press.

LLOYD, J. W. AND SHEPHERDSON, J. C. 1991. Partial evaluation in logic programming. *Journal of Logic Programming 11*, 217–242.

PAIGE, R. AND KOENIG, S. 1982. Finite differencing of computable expressions. *ACM Transactions on Programming Languages and Systems 4*, 3, 402–454.

PARTSCH, H. A. 1990. *Specification and Transformation of Programs.* Springer-Verlag.

PETTOROSSI, A. 1977. Transformation of programs and use of tupling strategy. In *Proceedings Informatica 77, Bled, Yugoslavia* (1977), pp. 1–6.

PETTOROSSI, A. AND PROIETTI, M. 1994. Transformation of logic programs: Foundations and techniques. *Journal of Logic Programming 19,20*, 261–320.

PETTOROSSI, A. AND PROIETTI, M. 1996. A theory of logic program specialization and generalization for dealing with input data properties. In O. DANVY, R. GLÜCK, AND P. THIEMANN Eds., *Partial Evaluation*, Lecture Notes in Computer Science 1110 (1996), pp. 386–408. Springer-Verlag.

PETTOROSSI, A., PROIETTI, M., AND RENAULT, S. 1997. Reducing nondeterminism while specializing logic programs. In *Proceedings 24-th POPL, Paris, France* (1997), pp. 414–427. ACM Press.

PROIETTI, M. AND PETTOROSSI, A. 1995. Unfolding-definition-folding, in this order, for avoiding unnecessary variables in logic programs. *Theoretical Computer Science 142*, 1, 89–124.

SØRENSEN, M. H., GLÜCK, R., AND JONES, N. D. 1996. A positive supercompiler. *Journal of Functional Programming 6*, 6, 811–838.

TAKANO, A. 1991. Generalized partial computation for a lazy functional language. In *Proceedings of PEPM '91, New Haven, Connecticut, USA* (1991), pp. 1–11. ACM Press.

TAMAKI, H. AND SATO, T. 1984. Unfold/fold transformation of logic programs. In S.-A. TÄRNLUND Ed., *Proceedings of the Second International Conference on Logic Programming, Uppsala, Sweden* (1984), pp. 127–138. Uppsala University.

TURCHIN, V. F. 1980. The use of metasystem transition in theorem proving and program optimization. In J. W. DE BAKKER AND J. VAN LEEUWEN Eds., *Proceedings 7-th ICALP*, Lecture Notes in Computer Science 85 (1980), pp. 645–657. Springer-Verlag.

TURCHIN, V. F. 1986. The concept of a supercompiler. *ACM TOPLAS 8*, 3, 292–325.

VITTER, J. S. AND FLAJOLET, P. 1990. Average-case analysis of algorithms and data structures. In J. VAN LEUVEEN Ed., *Handbook of Theoretical Computer Science*, Volume A, pp. 431–524. Elsevier.

WADLER, P. L. 1988. Deforestation: Transforming programs to eliminate trees. In *Proceedings of ESOP '88*, Lecture Notes in Computer Science 300 (1988), pp. 344–358. Springer-Verlag.

WAND, M. 1980. Continuation-based program transformation strategies. *Journal of the ACM 27*, 1, 164–180.