Transformation Rules and Strategies for Functional-Logic Programs [*]

Ginés Moreno
Dep. Informática, UCLM, 02071 Albacete, Spain.
E-mail: gmoreno@info-ab.uclm.es

This paper abstracts the contents of a Ph.D. dissertation entitled *"Transformation Rules and Strategies for Functional-Logic Programs"* which has been recently defended. These techniques are based on fold/unfold transformations and they can be used to optimize integrated (functional-logic) programs for a wide class of applications. Experimental results shows that typical examples in the field of Artificial Intelligence are successfully enhanced by our transformation system SYNTH. The thesis presents the first approach of these methods for declarative languages that integrate the best features from functional and logic programming.

Keywords: Program Transformation, Functional Logic Programming, Fold/Unfold

1. Introduction

The integration of functional and logic programming languages is one of the most interesting research problems in the area of declarative programming. In order to develop useful and practical integrated languages, it is essential to succeed in shrinking the efficiency gap with respect to imperative languages, as is already being done for Prolog. To this goal, formally based, practical tools for the analysis and transformation of functional logic programs which are able to improve the current implementations are a pressing need.

Program transformation aims to derive better semantically equivalent programs. Of the great amount of literature on program transformation, the so-called "Rules + Strategies" approach has been intensively exploted over the last two decades. It has been extensively applied both in functional and in logic programming. This approach is commonly based on the construction, by means of a strategy, of a sequence of equivalent programs each obtained by the preceding ones using an elementary transformation rule. The essential rules are folding and unfolding, i.e., contraction and expansion of subexpressions of a program using the definitions of this program (or of a preceding one). Other rules which have been considered are, for example, instantiation, definition introduction/elimination, and abstraction.

In this thesis, we extend this approach to an integrated functional logic setting. We study the semantic properties of the transformation as well as the conditions under which the transformation rules can be safely applied in order to preserve the set of values and computed answer substitutions for queries in both the original and the transformed progmams. We also adapt the composition and tupling strategies to our setting for appropriately guiding the application of transformation rules in order to obtain efficient programs. To the best of our knowledge, this is the first formal approach of this kind of transformation systems for functional logic programs. The work presented in this thesis has been partially published in [2,3,1,4].

2. Fold/Unfold Transformations

Basically, the unfolding transformation consists of applying a narrowing step to the rhs of a given rule. The use of narrowing empowers the unfold transformation by implicitly embedding the instantiation rule (the operation of the Burstall and Darlington framework [6] which introduces an instance of an existing equation) into unfolding by means of unification. We have defined different instances of the unfolding rule by simply considering unrestricted, strict or lazy variants of narrowing as the underlying mechanism.

[*]This work has been partially supported by CICYT under grant TIC 2001-2705-C03-03 and by Acción Integrada Hispano-Italiana HI2000-0161.

In the thesis, we first show what are the problems with naïve extensions of the unfolding rule when considering unrestricted narrowing as the language operational semantics. With such a general semantics it results quite difficult to preserve the computed answers. Thus, we first identify the conditions under which we can prove the soundness and completeness of an unfolding rule. Then we show a non standard and extremely useful relationship of Partial Evaluation with unfolding. We show that a slightly modified trasformation (*generalized unfolding*) can be formulated in terms of partial evaluation.

Since the use of unrestricted narrowing to perform unfolding may produce an important increase in the number of program rules, we have then instantiated the general unfolding definition to the case of an strict (*call-by-value*) narrowing strategy called innermost conditional narrowing. We have proved that the unfolding transformation preserve the computed answers under the usual conditions for the completeness of innermost conditional narrowing. Finally, as an example application of the unfolding technique we have defined an *unfolding semantics* consisting of a (possibly infinite) set of unconditional rules, computed as the limit of the unfolding expansions of the initial program. We show that this allows one to compute the set of computed answers for a goal by syntactic unification of the goal with elements in the unfolding semantics.

When defining the unfolding rule in terms of *call-by-name* variants of narrowing (e.g., lazy narrowing [10]) we observe that , in general, the structure of the original program (orthogonality) is not preserved, thus seriously restricting the applicability of the resulting system: the transformed program could not be further transformed and, what is even worse, it might not even be safely executed, as it might not satisfy the conditions for the completeness of the considered operational mechanism. Fortunately, this fact is not true for the case of needed narrowing, which is complete for *inductively sequential* programs [5]. Thus, we have demonstrated that such a program structure is preserved by an unfolding rule based on needed narrowing which is a key point for proving its correctness as well as its effective use.

On the other hand the folding transformation is intended to be the inverse of the unfolding operation, that is, an unfolding step followed by the corresponding folding step (and viceversa) is expected to give back the initial program. Roughly speaking, the folding operation consists of substituting a function call (*folding call*) for a definitionally equivalent set of calls (*folded calls*).

The definition of a folding transformation for an unrestricted narrowing strategy requires conditions which are too strong to preserve computed answers. For this reason we have firstly defined a folding rule directly for innermost narrowing that can be seen as an extension to functional logic programs of the reversible folding of [11] for logic programs. We have chosen this form of folding since it exhibits the useful, pursued property that the answer substitutions computed by innermost narrowing are preserved through the transformation. This definition has two sources of nondeterminism: the choice of the folded calls and in the choice of a generalization (folding call) of the heads of the instantiated function definitions which are used to substitute the folded calls (this definition is called "disjunctive", since it allows the folding of multiple rules).

In order to reduce the high level of nondeterminism, we have also proposed a new variant of folding similar to the one presented by [8] where a rule is folded by a unique rule in the same program. This new definition can be seen as a "conjunctive" version of the previous one. The main property of this style of folding is that it is reversible since it can always be undone by an unfolding step. This greatly simplifies the correctness proofs —correctness of folding follows immediately from the correctness of unfolding—, but usually require too strong applicability conditions, such as requiring that both the folded and the folding rules belong to the same program, which drastically reduces the power of the transformation.

Our last folding rule defined in the thesis can be combined with the needed narrowing based unfolding transformation and it is able to fold rules belonging to different programs (similarly to the one presented in [13] for logic programs). It is more powerful than the previous ones and the applicability conditions are less restrictive. Therefore, its use within a transformation system —when guided by appropriate strategies— is able to produce more effective optimizations for lazy functional logic programs.

3. A Complete Transformation System

The set of rules presented so far (together with two ones that allow the introduction/elimination of new function definitions) constitutes the kernel of our transformation system. These rules suffice for automatizing the *composition* strategy. However, the transformation system must be empowered for achieving the *tupling* optimization, which we attain by extending the transformation system with a rule of abstraction [6,12]. It essentially consists of replacing the occurrences of some expression e in the rhs of a rule R by a fresh variable z, adding the "local declaration" $z = e$ within a *where* expression in R. The new rules introduced by the where–abstraction do contain extra variables in the right-hand sides (which is explicitly forbidden in our setting). However, as noted in [12], this can be easily amended by using standard "lambda lifting" techniques. Our abstraction rule is inspired by the standard lambda lifting transformation of functional programs and it allows the abstraction of different expressions in one go.

The building blocks of strategic program optimizations are transformation tactics (*strategies*), which are used to guide the process and effect some particular kind of change to the program undergoing transformation [11]. The composition and tupling strategies were originally introduced in [6], for the optimization of pure functional programs. By using the composition strategy (or its variants), one may avoid the construction of intermediate data structures that are produced by some function and consumed as inputs by another function. On the other hand, the tupling strategy is very effective when several functions require the computation of the same subexpression, in which case we tuple together those functions. By avoiding either multiple accesses to data structures or common subcomputations one often gets linear recursive programs (i.e., programs whose rhs's have at most one recursive call) from nonlinear recursive programs [11].

The basic rules presented so far have been implemented by a prototype system SYNTH. It is written in SICStus Prolog and includes a parser for the language Curry, a modern multiparadigm declarative language based on needed narrowing which is intended to become a standard in the functional logic community [9]. It also includes a fully automatic composition strategy based on some (apparently reasonable) heuristics. We are currently extending the system in order to mechanize tupling (e.g., by using the analysis method of [7]).

References

[1] M. Alpuente, M. Falaschi, C. Ferri, G. Moreno, and G. Vidal. Un sistema de transformación para programas multiparadigma. *Revista Iberoamericana de Inteligencia Artificial*, X/99(8):27–35, 1999.

[2] M. Alpuente, M. Falaschi, G. Moreno, and G. Vidal. Safe folding/unfolding with conditional narrowing. *Proc. of ALP'97, Southampton (England)*, pages 1–15. Springer LNCS 1298, 1997.

[3] M. Alpuente, M. Falaschi, G. Moreno, and G. Vidal. A Transformation System for Lazy Functional Logic Programs. *Proc. of FLOPS'99, Tsukuba (Japan)*, pages 147–162. Springer LNCS 1722, 1999.

[4] M. Alpuente, M. Falaschi, G. Moreno, and G. Vidal. An Automatic Composition Algorithm for Functional Logic Programs. *Proc. of SOFSEM'2000*, pages 289–297. Springer LNCS 1963, 2000.

[5] S. Antoy. Definitional trees. *Proc. of ALP'92*, pages 143–157. Springer LNCS 632, 1992.

[6] R.M. Burstall and J. Darlington. A Transformation System for Developing Recursive Programs. *Journal of the ACM*, 24(1):44–67, 1977.

[7] W. Chin. Towards an Automated Tupling Strategy. In *Proc. of Partial Evaluation and Semantics-Based Program Manipulation, 1993*, pages 119–132. ACM, New York, 1993.

[8] P. A. Gardner and J. C. Shepherdson. Unfold/fold Transformation of Logic Programs. In J.L Lassez and G. Plotkin, editors, *Computational Logic, Essays in Honor of Alan Robinson*, pages 565–583. The MIT Press, Cambridge, MA, 1991.

[9] M. Hanus, H. Kuchen, and J.J. Moreno-Navarro. Curry: A Truly Functional Logic Language. In *Proc. ILPS'95 Workshop on Visions for the Future of Logic Programming*, pages 95–107, 1995.

[10] J.J. Moreno-Navarro and M. Rodríguez-Artalejo. Logic Programming with Functions and Predicates: The language Babel. *Journal of Logic Programming*, 12(3):191–224, 1992.

[11] A. Pettorossi and M. Proietti. Rules and Strategies for Transforming Functional and Logic Programs. *ACM Computing Surveys*, 28(2):360–414, 1996.

[12] D. Sands. Total Correctness by Local Improvement in the Transformation of Functional Programs. *ACM Transactions on Programming Languages and Systems*, 18(2):175–234, March 1996.

[13] H. Tamaki and T. Sato. Unfold/Fold Transformations of Logic Programs. In S. Tärnlund, editor, *Proc. of Second Int'l Conf. on Logic Programming, Uppsala, Sweden*, pages 127–139, 1984.