A Unifying View of Functional and Logic Program Specialization

MARÍA ALPUENTE
Universidad Politécnica de Valencia, Spain
and
MORENO FALASCHI
Università di Udine, Italy
and
GERMÁN VIDAL
Universidad Politécnica de Valencia, Spain

In the last decade, much research effort in programming languages has been devoted to combining the most important declarative paradigms, namely, functional programming (FP) and logic programming (LP). A functional logic programming language allows one to write conventional functional programs as well as LP-like ones. The operational semantics of the integrated language is usually based on narrowing, a combination of unification for parameter passing and reduction as evaluation mechanism.

Partial Evaluation techniques have been known for a long time and have been applied to the optimization of functional and logic programs. However, the problem of extending LP or FP partial evaluation techniques for integrated languages is far from trivial, since logical variables in function calls place specific demands that have to be tackled to achieve effective specialization.

This article gives a general introduction to the particular problems associated with the PE of functional logic programs, explains why this is important and shows that it is useful to transfer the technology of narrowing into a technique for driving specialization in integrated languages.

1. FUNCTIONAL LOGIC PROGRAM SPECIALIZATION

The aim of *partial evaluation* (PE) is to specialize a given program w.r.t. part of its input data (hence also called *program specialization*). PE has been widely applied in the field of functional programming [5; 10] and logic programming [7; 13]. Although the objectives are similar, the general methods are often different due to the distinct underlying computation models. This separation has the negative consequence of duplicated work since developments are not shared and many similarities are overlooked. In recent years, some correspondences have been established among different techniques in some particular cases [8; 15; 17].

Narrowing generalizes the rewriting machinery of functional programs to cover LP features such as logical variables, partial data structures and built-in search as well as functional features like polymorphism, higher-order and lazy (demand-driven) evaluation. Narrowing is complete in the sense of functional programming (computation of normal forms) as well as logic programming (computation of answers). Essentially, it consists of computing an appropriate substitution such that when applied to the current goal it becomes reducible, and then reduces it [9].

Narrowing-driven PE [3] is the first generic algorithm for the specialization of functional logic languages. The method is formalized within the theoretical framework established in [13; 14] for the partial evaluation of logic programs (also known as *partial deduction*, PD), however a number of concepts have been generalized to deal with nested function calls. Our approach has better opportunities for optimization thanks to the functional dimension. Also, since unification is embedded into narrowing, we are able to automatically propagate syntactic information on the partial input (term structure) and not only constant values. The different instances of our framework which can be obtained by taking different narrowing strategies preserve some logical, strong (computed answers) program semantics under conditions easily ascertained by reusing methods and results developed for narrowing.

We are not aware of any formal antecedent of the narrowing-driven approach in the PE literature, although the idea can be traced back to [6]. A closer, automatic approach is that of positive supercompilation (PS) [8], whose basic transformation operation is *driving*, a unification-based transformation mechanism which is somehow similar to (lazy) narrowing. Although our work shares many similarities with PS, it still has some notable differences (some of which are discussed below). A different PE method for a rewriting-based, FL language is presented in [11].

2. NARROWING-DRIVEN PARTIAL EVALUATION

Given a program P and a set S of atoms, the aim of PD [13] is to derive, a new program P' which computes the same answers for any input goal which is an instance of an atom in S. The program P' is obtained by gathering together the set of *resultants*, which are constructed as follows: for each atom A of S, i) first construct a finite SLD-tree, $T(A)$, for $P \cup \{\Leftarrow A\}$, then ii) consider the leaves of the non-failing branches of $T(A)$, say $G_1, \ldots, G_r$, and the computed substitutions along these branches, say $\theta_1, \ldots, \theta_r$, and finally iii) construct the clauses: $A\theta_1 \Leftarrow G_1, \ldots, A\theta_r \Leftarrow G_r$. The basic correctness of the transformation is ensured whenever P' is *S-closed*, i.e. every atom in P' is an instance of an atom in S. An *independence* condition, which holds if no two atoms in S have a common instance, is needed to guarantee that P' does not produce additional answers. Using the terminology of [15], the constructed SLD-trees can be viewed as (i) *symbolic computations* for the atoms in S; the S-closedness of P' illustrates the idea of (ii) *regularity* of a symbolic computation; and finally, (iii) *program extraction* from a set of SLD-trees consists basically in building up the associated set of resultant rules.

We now identify these three categories for narrowing-driven PE [3].

Symbolic Execution. The execution model is similar to PD, but we use narrowing in the place of SLD-resolution. For a set S of terms (with nested calls) and a functional logic program $R = \{\lambda_i \to \rho_i \Leftarrow C_i\}_{i=1}^{n}$ (a conditional term rewriting system), a partial (finite) narrowing tree is constructed for each term s in S. The inclusion of a deterministic, normalization process between narrowing steps improves the elimination of intermediate data structures and reduces the size of the specialized program since less choices are unfolded [1]. By exploiting the results on *normalizing narrowing* [9], this is achieved in a principled way which does not compromise termination. Control issues are managed by using standard techniques in [14; 16].

Search for Regularities. Our notion of regularity is similar to the PD closedness condition, which we have generalized to recurse over the terms in order to handle nested function calls. Informally, a term t is considered S-closed iff it only contains constructors (symbols that do not appear as the outermost function symbol of any rule of the program) and variables, or i) there exists a substitution θ such that $t\theta \in S$, and ii) the terms in θ are recursively S-closed. For instance, the term $f(g(0))$ is closed w.r.t. the set of calls $\{f(x), g(x)\}$. This recursive notion of closedness substantially enhances the specialization power of the method, since it supplies a notion of "similarity" which subsumes both the standard, PD-like notion of closedness and the *perfect* ("α-identical") closedness test of [17]. It also differs from the folding operation in [16], which does not allow to fold back a function call which is closed by a different branch of the driving tree.

Program Extraction. It is not straightforward to extend the notion of resultant to our setting. Consider a narrowing derivation from the initial goal $\Leftarrow A$ to $\Leftarrow G$ computing θ, where A is an equation. This would correspond to the restriction to perform PD on single atoms used in [13]. The associated LP-like resultant (i.e. the rule $A\theta \rightarrow true \Leftarrow G$) is not useful, since it does not specialize any user-defined function of the original program, but the equality symbol in the equation A. Also, if we handle conjunctions of equations as boolean-valued terms and we define resultants by $A\theta \rightarrow G$, then few opportunities to find regularities are provided and very poor specialization is achieved (by effect of the abstraction operation required to guarantee termination), unless special partitioning techniques are introduced (in the line of conjunctive PD [12]). Thus, we decided to specialize single terms s, and so we consider derivations for initial goals $s = y$, where y is a fresh variable, that we prolong down to the leaves $(C, t = y)$ (where C are the equations brought by the conditions of the applied program rules), and we extract the resultant as $(s\theta \rightarrow t \Leftarrow C)$. As in PD, the 'root-to-leaf' notion of resultant results in fewer rules than in the case of PS, where rules are extracted from each single computation step, but has the disadvantage that there are less opportunities to find appropriate regularities, just because fewer specialized functions are produced.

3. NARROWING STRATEGIES AND THE NEED FOR RENAMING

The behavior of a concrete narrowing-driven partial evaluator greatly depends on the narrowing strategy. Our results in [1; 2] show that if computed answer semantics is to be preserved by program transformations, then the only reasonable class is that of constructor-based (CB) programs with constructor answers. Unfortunately, even if the original program is CB, the residual program might not be, which is undesirable for the completeness of some strategies such as lazy narrowing. A post-processing renaming transformation is then necessary to restore the constructor discipline [1]. Complex terms are 'folded' recursively, by replacing them by calls to new functions which satisfy the CB constraint. The renaming phase also achieves a more involved *independence* condition (w.r.t. PD) that guarantees the strong correctness of the transformation by removing *overlaps* between the specialized function calls. Renaming is also striking for conjunctive PE techniques in our setting.

4. FUTURE RESEARCH

Further studies are needed to accommodate the propagation of negative information through (constrained) narrowing derivations in our framework as well as to formalize some form of conjunctive partial evaluation comparable to [12]. We have started this line of work using the lazy higher order functional logic language TOY [4], pursuing the development of a self-applicable partial evaluator.

REFERENCES

[1] M. Alpuente, M. Falaschi, P. Julián, and G. Vidal. Specialization of Lazy Functional Logic Programs. In *Proc. PEPM'97*, pages 151–162. ACM, New York, 1997.

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

[3] M. Alpuente, M. Falaschi, and G. Vidal. Narrowing-driven Partial Evaluation of Functional Logic Programs. In *Proc. ESOP'96*, pages 45–61. Springer LNCS 1058, 1996.

[4] R. Caballero-Roldán, F.J. López-Fraguas, and J. Sánchez-Hernández. User's manual for Toy. Technical report SIP-5797, UCM, Madrid (Spain), April 1997.

[5] C. Consel and O. Danvy. Tutorial notes on Partial Evaluation. In *Proc. of 20th Annual ACM Symp. on Principles of Programming Languages*, pages 493–501. ACM, New York, 1993.

[6] J. Darlington and H. Pull. A Program Development Methodology Based on a Unified Approach to Execution and Transformation. In *Proc. of the Int'l Workshop on Partial Evaluation and Mixed Computation*, pages 117–131. North-Holland, Amsterdam, 1988.

[7] J. Gallagher. Tutorial on Specialisation of Logic Programs. In *Proc. of Partial Evaluation and Semantics-Based Program Manipulation*, pages 88–98. ACM, New York, 1993.

[8] R. Glück and M.H. Sørensen. Partial Deduction and Driving are Equivalent. In *Proc. PLILP'94*, pages 165–181. Springer LNCS 844, 1994.

[9] M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, 19&20:583–628, 1994.

[10] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, Englewood Cliffs, NJ, 1993.

[11] L. Lafave and J.P. Gallagher. Partial Evaluation of Functional Logic Programs in Rewriting-based Languages. Tech. Report CSTR-97-001, U. Bristol, England, March 1997.

[12] M. Leuschel, D. De Schreye, and A. de Waal. A Conceptual Embedding of Folding into Partial Deduction: Towards a Maximal Integration. In *Proc. JICSLP'96*, pages 319–332. The MIT Press, Cambridge, MA, 1996.

[13] J.W. Lloyd and J.C. Shepherdson. Partial Evaluation in Logic Programming. *Journal of Logic Programming*, 11:217–242, 1991.

[14] B. Martens and J. Gallagher. Ensuring Global Termination of Partial Deduction while Allowing Flexible Polyvariance. In *Proc. ICLP'95*, pages 597–611. MIT Press, 1995.

[15] A. Pettorossi and M. Proietti. A Comparative Revisitation of Some Program Transformation Techniques. In *Partial Evaluation, Int'l Seminar*, pages 355–385. LNCS 1110, 1996.

[16] M.H. Sørensen and R. Glück. An Algorithm of Generalization in Positive Supercompilation. In*Proc. ILPS'95*, pages 465–479. The MIT Press, Cambridge, MA, 1995.

[17] M.H. Sørensen, R. Glück, and N.D. Jones. A Positive Supercompiler. *Journal of Functional Programming*, 6(6):811–838, 1996.