

A Partial Evaluation Tool for Multi-Paradigm Declarative Languages

Germán Vidal

DSIC, Technical University of Valencia
Camino de Vera s/n, E-46022 Valencia, Spain
gvidal@dsic.upv.es

Abstract— This article describes a practical (online) partial evaluation tool which is applicable to modern multi-paradigm declarative languages (like, e.g., Curry). The most recent proposal for multi-paradigm declarative programming advocates the integration of features from functional, logic and concurrent programming. The developed tool has been successfully integrated into the PAKCS compiler for the language Curry.

Keywords— Partial evaluation, Multi-paradigm declarative programming.

I. INTRODUCTION

Modern proposals for multi-paradigm declarative programming combine features from functional, logic and concurrent programming (see, e.g., [18], [19]). The resulting language includes the most important features of the distinct paradigms: lazy evaluation, higher-order functions, non-deterministic computations, concurrent evaluation of constraints with synchronization on logical variables, and a unified computation model which integrates *narrowing* and *residuation* [18].

A *partial evaluator* is a program transformer which takes a program and part of its input data and constructs a *specialized* version of the program for the given data. The new, residual program (hopefully) runs more efficiently than the original one since those computations that depend only on the known data have been performed once and for all at partial evaluation time. Our program transformer is based on the *narrowing-driven* approach to partial evaluation (see [5] for a survey). The syntax of multi-paradigm declarative programs is closer to the one of pure functional programs. In spite of this, the narrowing-driven partial evaluation framework shares more similarities with unification-based methods for the partial evaluation of logic programs [26] than to traditional partial evaluators for functional programs based on constant propagation [22]. Apart from its ability to specialize programs, a narrowing-driven partial evaluator is also able to improve programs even when no input data are provided (e.g., by removing unnecessary data structures). These remarks also apply to several partial evaluation methods for logic programs, e.g., (conjunctive) partial deduction [26], [13], as well as to some (online) methods for functional programs like supercompilation [35], generalized partial computation [15], and positive supercompilation [32], the closest to our framework.

An implementation of the partial evaluator has been

successfully incorporated into the latest distribution of the PAKCS [20] programming environment for the multi-paradigm language Curry [19]. The partial evaluation tool has been implemented in Curry itself, so it is purely declarative (and, thus, easy to extend with new developments).

This article is organized as follows. Section II presents the considered source language as well as the intermediate representation used by the partial evaluator. We describe the design of the partial evaluation tool in Section III. Then, in Section IV, we discuss several possibilities to extend our tool with additional capabilities. Finally, Section V concludes.

II. THE SOURCE LANGUAGE

The partial evaluator is applicable to modern multi-paradigm languages which integrate the most important features of functional and logic programming (like, e.g., Curry [19]). In these languages, programs follow a purely functional syntax, e.g., functions are defined by a sequence of rules (or equations) of the form:

$$f\ t_1 \dots t_n = e$$

where t_1, \dots, t_n are *constructor* terms and the right-hand side e is an arbitrary expression. Constructor terms may contain variables and constructor, i.e., symbols which are not defined by the program rules. Functions can also be defined by *conditional equations*:

$$f\ t_1 \dots t_n \mid c = e$$

where the condition (or *guard*) c can be either a Boolean function or a constraint. *Equational constraints* $e_1 = e_2$ are satisfied if both expressions are reducible to a same constructor term. Higher-order features include partial function applications and lambda abstractions. Finally, we also consider the use of functions which are not defined in the user's program (*external* functions), like arithmetic operators, common higher-order functions (`map`, `foldr`, etc.), input/output facilities, etc.

The basic operational semantics of our source language is based on a combination of (needed) narrowing and residuation [18]. The *residuation* principle is based on the idea of delaying function calls until they are sufficiently instantiated to be reduced by rewriting, i.e., a pure functional reduction. On the other hand, the *narrowing* mechanism allows the instantiation of free variables in input expressions and then applies reduction steps to the function calls of the instantiated expression. This augments our language with the typical logic

\mathcal{R}	$::= D_1 \dots D_m$		
D	$::= f(x_1, \dots, x_n) = e$		
e	$::= x$	(variable)	$ c(e_1, \dots, e_n)$ (constructor call)
	$ \text{case } e_0 \text{ of } \{p_1 \rightarrow e_1; \dots; p_n \rightarrow e_n\}$	(rigid case)	$ f(e_1, \dots, e_n)$ (function call)
	$ \text{fcase } e_0 \text{ of } \{p_1 \rightarrow e_1; \dots; p_n \rightarrow e_n\}$	(flexible case)	$ \text{external}(e)$ (external function call)
	$ \text{partcall}(f, e_1, \dots, e_k)$	(partial application)	$ \text{apply}(e_1, e_2)$ (application)
	$ \text{constr}([x_1, \dots, x_n], e)$	(constraint)	$ \text{or}(e_1, e_2)$ (disjunction)
	$ \text{guarded}([x_1, \dots, x_n], e_1, e_2)$	(guarded expression)	
p	$::= c(x_1, \dots, x_n)$		

Fig. 1. The Flat Representation for Programs

Case Select:	$(f)\text{case } c(\overline{e_n}) \text{ of } \{\overline{p_k \rightarrow e'_k}\} \Rightarrow_{id} \sigma(e'_i)$ if $p_i = c(\overline{x_n})$ and $\sigma = \{\overline{x_n \mapsto e_n}\}$
Case Guess:	$\text{fcase } x \text{ of } \{\overline{p_k \rightarrow e_k}\} \Rightarrow_{\sigma} \sigma(e_i)$ if $\sigma = \{x \mapsto p_i\}$, $i = 1, \dots, k$
Case Eval:	$(f)\text{case } e \text{ of } \{\overline{p_k \rightarrow e_k}\} \Rightarrow_{\sigma} \sigma((f)\text{case } e' \text{ of } \{\overline{p_k \rightarrow e_k}\})$ if e is a function call or a case expression, and $e \Rightarrow_{\sigma} e'$
Function Eval:	$f(\overline{e_n}) \Rightarrow_{id} \sigma(e')$ if $f(\overline{x_n}) = e' \in \mathcal{R}$ and $\sigma = \{\overline{x_n \mapsto e_n}\}$

Fig. 2. The LNT calculus

programming capabilities (logical variables, search for solutions, etc). The precise mechanism—narrowing or residuation—for each function is specified by *evaluation annotations*. Functions annotated as *rigid* are evaluated by residuation, while functions annotated as *flexible* can be evaluated in a non-deterministic way by narrowing.

Example II.1: Let us consider the following rules defining the addition and subtraction in Curry:

```

add Z      y = y
add (Succ x) y = Succ (add x y)

sub x y | add y z ::= x = z

```

where natural numbers are built from **Z** and **Succ**. While the definition of **add** is purely functional, function **sub** is defined in terms of **add** by exploiting the logic programming capabilities of the language.

As discussed in [2], the definition of a *practical* partial evaluator for source programs is not feasible. Indeed, as the operational semantics gets more elaborated, the associated partial evaluation methods get also more and more complex. To overcome this problem, a promising approach successfully applied in other contexts (e.g., [11], [17], [28]) is to consider programs written in an intermediate programming language with a simple operational semantics (and to automatically translate source-level programs into this intermediate language). Recently, [21] introduced such a simplified representation for functional logic programs [7]. In the following, we present an extension of this intermediate representation which covers all the facilities of the language. The syntax of the resulting representation is depicted in Figure 1. Following the terminology of [21], we refer to this representation as “the *flat* representation for programs.”

Within this (first-order) representation, a flat program \mathcal{R} consists of a sequence of function definitions D

such that each function is defined by a single rule whose left-hand side contains only different variables as parameters. The right-hand side is an expression e composed by variables (e.g., x, y, z, \dots), constructors (e.g., a, b, c, \dots), function calls (e.g., f, g, \dots), and case expressions for pattern matching. Additionally, we also allow external functions, higher-order features like partial application and an application of a functional expression to an argument, constraints (possibly containing existentially quantified variables), guarded expressions (to represent conditional rules, where the list of variables are the local variables which are visible in the guard and the right-hand side), and disjunctions (to represent functions with overlapping left-hand sides). Let us mention that source-level programs (e.g., Curry programs) can be automatically translated to the flat representation; indeed, it essentially coincides with the standard intermediate representation, FlatCurry, used during the compilation of Curry programs [8], [20], [27].

Example II.2: Let us consider again the functions **add** and **sub** of Example II.1. Their translation into the flat representation is as follows:

```

add x y = case x of
  { Z → y;
    (Succ x') → Succ (add x' y) }

sub x y = guarded [z]
  (constr [] (add y z ::= x))
  z

```

The operational semantics of flat programs is based on the LNT calculus [21], which has been recently extended to cope with case expressions including evaluation annotations in [3]. The core of the LNT rules is presented in Figure 2. LNT steps are labeled with the substitution computed in the step. The empty substitution is

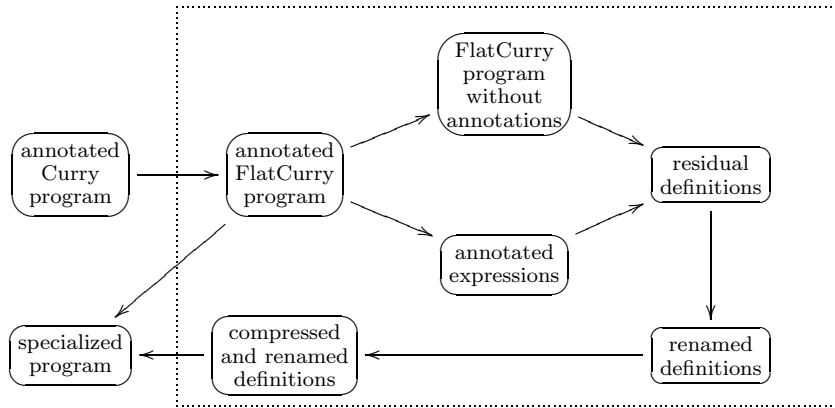


Fig. 3. Overview of the Partial Evaluation Process

denoted by *id*. Let us briefly describe the LNT rules:

- **Case Select.** It is used to select the appropriate branch of the current case expression.
- **Case Guess.** It non-deterministically selects a branch of a flexible case expression and instantiates the variable at the case argument to the appropriate constructor pattern. The step is labeled with the computed substitution σ . Rigid case expressions with a variable argument *suspend*, giving rise to an abnormal termination.
- **Case Eval.** It is used to evaluate case expressions whose argument is a function call or another case expression. This rule initiates the evaluation of the argument by creating a (recursive) call for this subexpression. If it cannot be evaluated, we stop (unsuccessfully).
- **Function Eval.** This rule performs the unfolding of a function call. As in logic programming, we assume that rules are renamed so that they always contain fresh variables.

The basic semantics can be properly extended to cover all the features of the flat language (see, e.g., [19, Appendix D] where the operational semantics of the language Curry is provided).

III. THE PARTIAL EVALUATION TOOL

Let us begin with an overview of the partial evaluator. Our partial evaluation tool constructs optimized, residual versions for some “parts” of an input program. Those “parts” to be partially evaluated are annotated in the program by means of the function **PEVAL**. Once the program is annotated, the partial evaluation process is fully automatic (i.e., there is no need for user interaction) and always terminating.

An implementation of the partial evaluator has been successfully incorporated into the latest distribution of the PAKCS [20] programming environment for Curry. The process consists of the following phases (Fig. 3):

- The process starts out with an annotated Curry program in which the expressions **exp** to be partially evaluated are replaced by (**PEVAL exp**).
- The source program is then translated into the FlatCurry syntax, the standard intermediate representation which is used during the compilation of Curry programs [8], [20], [27].

- The process continues by extracting the set of annotated expressions and by creating a copy of the FlatCurry program without the **PEVAL** annotations. Both the program and the set of expressions are the input for the proper partial evaluation process (see Section III-A below).
- The output of the partial evaluation process is a set of new, residual function definitions to execute the annotated expressions of the source program. These new definitions do not generally fulfill the syntax of FlatCurry and, therefore, a post-processing of *renaming* is applied. This is also useful to remove some redundant symbols from the residual definitions.
- Frequently, residual definitions contain a number of “useless” functions which are only used to pass control between two program points. Therefore, we finish the process by applying a *compression* phase which is useful to produce more compact and legible definitions.
- The final program is obtained from the original one as follows: first, residual definitions are added to the original program; then, each marked expression (**PEVAL exp**) of the original program is replaced by **exp'**, where **exp'** is the renaming of the expression **exp**.

The partially evaluated program will be stored in FlatCurry format, in contrast to the original program which was written in Curry. This is not a restriction since FlatCurry programs are directly executable by the compiler of PAKCS.

A. The Partial Evaluation Algorithm

The kernel of the specialization process consists of an iterative algorithm which takes a program and a set of expressions as input and returns a set of residual definitions for the given expressions. Essentially, it proceeds by (iteratively) unfolding a set of function calls, testing the *closedness* of the unfolded expressions, and adding to the current set those calls (in the derived expressions) which are not closed. This process is repeated until all the unfolded expressions are closed, which guarantees the correctness of the transformation process [6]. The computation of a closed set of expressions can be regarded as the construction of a graph with all the program points which are reachable from the initial call. Informally speaking, an expression is *closed* whenever

Input: a program \mathcal{R} and a set of expressions E_0
Output: a residual program \mathcal{R}'
Initialization: $i := 0$
Repeat
 $E' := \text{unfold}(E_i, \mathcal{R});$
 $E_{i+1} := \text{abstract}(E_i, E');$
 $i := i + 1;$
Until $E_i = E_{i-1}$ (modulo renaming)
Return:
 $\mathcal{R}' := \text{build_residual_program}(E_i, \mathcal{R})$

Fig. 4. Narrowing-Driven Partial Evaluation Procedure

its function calls are instances of the already computed expressions in the set (a formal definition can be found in [5], [6]). This iterative style of performing partial evaluation was first described by Gallagher [16] for the partial evaluation of logic programs.

The basic partial evaluation procedure can be seen in Fig. 4. The operator *unfold* takes a set of expressions, computes a *finite* set of (possibly incomplete) finite derivations, and returns the set of derived expressions. Function *abstract* is used to properly add the new expressions to the current set of (to be) partially evaluated expressions such that the termination of the whole process is guaranteed. Therefore, the main loop of the algorithm can be seen as a *pre-processing* stage whose aim is to find a closed set of expressions. Note that no residual rules are actually constructed during this phase. Only when a closed set of expressions is eventually found, residual rules are built (usually, by applying one more time the unfolding operator, followed by a post-processing of renaming and/or some post-unfolding transformations).

B. Unfolding

The unfolding of expressions is performed with a slight variation of the LNT rules of Fig. 2: the RLNT calculus [3]. The main difference with the LNT formulation is in the Case Guess rule. In particular, the new definition “residualizes” the case structure and continues with the evaluation of the different branches (by applying the corresponding substitution in order to propagate bindings forward in the computation).

Since the considered expressions can contain incomplete information, an unrestricted evaluation may not terminate. In order to ensure the termination of the process, the unfolding rule must incorporate some mechanism to stop the evaluation of expressions. For this, there are several well-known techniques in the literature, e.g., depth-bounds, loop-checks [10], well-founded/well-quasi orderings [12], [31], etc. Within the narrowing-driven approach, unfolding rules have been defined by using a particular type of well-quasi ordering: *homeomorphic embedding* [25]. Unfolding rules based on the embedding ordering allow the evaluation of expressions until reaching a term which embeds a previous term of the same derivation [6].

C. Abstraction

The abstraction phase cannot be managed with the same flexibility as the unfolding process if we want to preserve the correctness of the method. In the unfolding phase, this flexibility allows us to safely stop the evaluation of expressions at any point. In contrast, we cannot stop the iterative unfolding of expressions until all the reduced expressions are *closed* w.r.t. the corresponding set. This condition is necessary to ensure the correctness of the method [6]. On the other hand, it may also happen that this condition is never reached and, in this case, the iterative process runs forever. Therefore, the abstraction operator usually includes some kind of generalization to enforce the termination of the process. The most popular generalization operator is the *msg* (*most specific generalization*) between terms [24].

Restricting attention to the narrowing-driven approach, abstraction operators also use the embedding ordering to decide when to generalize and when to continue with the iterative unfolding of expressions. The abstraction operator *abstract* takes two sets of terms (the terms already partially evaluated E_i and the terms to be added to this set, E' , as shown in Figure 4) and returns a *safe* approximation of $E_i \cup E'$. By “safe” we mean that each term in $E_i \cup E'$ is closed w.r.t. the set of terms resulting from $\text{abstract}(E_i, E')$. This property is essential to guarantee the correctness of the partial evaluation process, i.e., to ensure that, despite the generalization performed by the abstraction operator, the final set of terms still covers the initial expression.

IV. EXTENSIONS AND FUTURE DEVELOPMENTS

In this section we discuss several possibilities to extend our tool with additional capabilities. First, we consider the use of profiling tools to assist the specialization process. Then, we consider the definition of a cost-augmented partial evaluator which not only returns a residual program, but also the cost improvement achieved by each residual rule. Finally, we discuss the definition of a program slicing tool based on our partial evaluation scheme.

A. Profiling

Profiling tools, in general, are designed for assisting the programmer in the task of generating efficient code (see, e.g., [9]). By analyzing the profiling results, the programmer may find those parts of the program which dominate the execution time. As a consequence of this, the code may be changed, recompiled and profiled again, in hopes of improving efficiency.

In the context of partial evaluation, we believe that profiling techniques can play an important role: The most immediate application consists of using the information gathered by the profiler to assess the effectiveness of the partial evaluation process. This can be done by simply comparing the cost information obtained for the original and the transformed programs. Also, profiling tools could be useful to detect which are the most expensive functions and, thus, promising candidates to be partially evaluated.

A profiling scheme for multi-paradigm functional logic programs can be found in [4]. Two features characterize this profiling scheme: it is defined for the *source* language, unlike traditional profilers which work by regularly interrupting the compiled program—hence, the attribution of costs to the user’s program constructions is straightforward—and it is *symbolic*, in the sense that it is independent of a particular implementation of the language—thus, it does not return *actual* execution times but a list of symbolic measures: the number of computation steps, the number of allocated cells, the number of nondeterministic branching points, and the number of pattern matching operations. Nevertheless, the experiments in [4] indicate that the speedup predicted using the symbolic cost criteria is a good approximation of the real speedup measured experimentally. A prototype implementation of a profiler for the language Curry [19] has been also developed.

B. Cost-Augmented Partial Evaluation

An alternative approach to the use of profiling tools is presented in [1], [36]. They introduce the scheme of a narrowing-driven partial evaluator enhanced with the computation of symbolic costs. While the previous approach of [4] presents two *independent* processes—narrowing-driven partial evaluation and the computation of profiling results—, [1], [36] fully integrate the computation of quantitative information into the specialization technique. Thus, we have available a setting in which one can discuss the effects of the program transformer in a precise framework and, moreover, to quantify these effects. This scheme may serve as a basis to develop speedup analyses and cost-guided transformers.

In particular, the work of [36] extends the standard semantics for functional logic programs in flat form—the LNT calculus—with the computation of symbolic costs. Basically, the enhanced semantics mimics the LNT calculus but additionally computes the symbolic costs attributed to a particular computation. In contrast to [1], the cost semantics is defined for *flat programs*. This makes the new approach more practically applicable, since actual implementations of functional logic languages use a flat representation as an intermediate language and, thus, realistic costs should be gathered at this level.

Since narrowing-driven partial evaluation builds residual rules by means of a residualizing variant of the standard semantics, the RLNT calculus, [36] also defines a cost-augmented version of the RLNT calculus. The relation, in terms of cost, between the standard and the residualizing semantics (augmented with costs) is established. The scheme of a narrowing-driven partial evaluator which uses the cost-augmented RLNT calculus to perform computations at specialization time is introduced. An implementation of the new scheme has been undertaken by extending an existing partial evaluator for Curry programs [2]. Preliminary experiments indicate that the approach is both practical and useful. The enhanced partial evaluator also includes a simple *speedup analysis* which may be helpful to obtain a global measure of the improvement achieved.

C. Program Slicing

Essentially, program slicing [38] is a method for decomposing programs by analyzing their data and control flow. It has many applications in the field of software engineering (e.g., program understanding, maintenance, debugging, reuse, merging, testing, etc). This concept was originally introduced in the context of imperative programs. Surprisingly, there are very few approaches to program slicing in the context of declarative programming (some notable exceptions are, e.g., [29], [30], [33]). Roughly speaking, a *program slice* is a subset of the program statements which (potentially) affect the values computed at some program point and/or variable, referred to as a *slicing criterion*. Program slices are usually computed from a *program dependence graph* [14], [23]. Program dependencies can be traversed backwards or forwards (from the slicing criterion), which is known as *backward* or *forward* slicing, respectively. A survey on program slicing can be found, e.g., in [34].

Many (online) partial evaluation schemes follow a common pattern: given a program and a partial call, the partial evaluator builds a *finite* representation—generally a graph—of the possible executions of the given call, and then extracts a *residual* program from this graph. This view of partial evaluation clearly shows the similarities with program slicing: both techniques should construct a (finite) representation of some program execution, usually with *part* of the input data.

In [37], a forward slicing method based on (online) partial evaluation is introduced. While the construction of a graph representing some program execution is quite similar in both techniques, the extraction of the final program is rather different. Partial evaluation usually achieves its effects by compressing paths in the graph and by renaming expressions while removing unnecessary function symbols. In contrast, program slicing should preserve the statements of the original program. Roughly speaking, (forward) slicing can be regarded as a rather conservative form of partial evaluation. Since the resulting method is defined in terms of an existing partial evaluation scheme, it is easy to implement by adapting current partial evaluators. Moreover, this approach helps us to clarify the relation between (forward) slicing and partial evaluation.

V. CONCLUSIONS

This article describes a practical (online) partial evaluation tool which is applicable to modern multi-paradigm declarative languages. The developed tool has been successfully integrated into the PAKCS compiler for the language Curry. Finally, we discuss several extensions of the partial evaluation scheme.

Acknowledgments

This work greatly benefits from joint work with Elvira Albert and Michael Hanus. The author gratefully acknowledges their useful comments and suggestions on the topics of this paper. This work has been partially supported by Acción Integrada Hispano-Alemana HA2001-0059 and by Spanish CICYT TIC 2001-2705-C03-01.

REFERENCES

- [1] E. Albert, S. Antoy, and G. Vidal. Measuring the Effectiveness of Partial Evaluation in Functional Logic Languages. In *Proc. of the 10th Int'l Workshop on Logic-Based Program Synthesis and Transformation (LOPSTR 2000)*, pages 103–124. Springer LNCS 2042, 2001.
- [2] E. Albert, M. Hanus, and G. Vidal. A Practical Partial Evaluation Scheme for Multi-Paradigm Declarative Languages. *Journal of Functional and Logic Programming*, 2002(1):1–34, 2002.
- [3] E. Albert, M. Hanus, and G. Vidal. A Residualizing Semantics for the Partial Evaluation of Functional Logic Programs. *Information Processing Letters*, 2002.
- [4] E. Albert and G. Vidal. Source-Level Abstract Profiling of Multi-Paradigm Declarative Languages. In *Proc. of Int'l Workshop on Logic-based Program Synthesis and Transformation (LOPSTR'01)*, 2001.
- [5] E. Albert and G. Vidal. The Narrowing-Driven Approach to Functional Logic Program Specialization. *New Generation Computing*, 20(1):3–26, 2002.
- [6] M. Alpuente, M. Falaschi, and G. Vidal. Partial Evaluation of Functional Logic Programs. *ACM Transactions on Programming Languages and Systems*, 20(4):768–844, 1998.
- [7] S. Antoy. Definitional trees. In *Proc. of the 3rd Int'l Conference on Algebraic and Logic Programming (ALP'92)*, pages 143–157. Springer LNCS 632, 1992.
- [8] S. Antoy and M. Hanus. Compiling Multi-Paradigm Declarative Programs into Prolog. In *Proc. of the 3rd Int'l Workshop on Frontiers of Combining Systems*, pages 171–185. Springer LNCS 1794, 2000.
- [9] J.L. Bentley. *Writing Efficient Programs*. Prentice-Hall, Englewood Cliffs, N.J., 1982.
- [10] R. Bol. Loop Checking in Partial Deduction. *Journal of Logic Programming*, 16(1&2):25–46, 1993.
- [11] A. Bondorf. A Self-Applicable Partial Evaluator for Term Rewriting Systems. In *Proc. of Int'l Conf. on Theory and Practice of Software Development*, pages 81–95. Springer LNCS 352, 1989.
- [12] M. Bruynooghe, D. De Schreye, and B. Martens. A General Criterion for Avoiding Infinite Unfolding. *New Generation Computing*, 11(1):47–79, 1992.
- [13] D. De Schreye, R. Glück, J. Jørgensen, M. Leuschel, B. Martens, and M.H. Sørensen. Conjunctive Partial Deduction: Foundations, Control, Algorithms, and Experiments. *Journal of Logic Programming*, 41(2&3):231–277, 1999.
- [14] J. Ferrante, K.J. Ottenstein, and J.D. Warren. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, 1987.
- [15] Yoshihiko Futamura. Partial Evaluation of Computation Process—An Approach to a Compiler-Compiler. *Higher-Order and Symbolic Computation*, 12(4):381–391, 1999. Reprint of article in *Systems, Computers, Controls* 1971.
- [16] J. Gallagher. Tutorial on Specialisation of Logic Programs. In *Proc. of the ACM SIGPLAN Symp. on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'93)*, pages 88–98. ACM, New York, 1993.
- [17] R. Glück and A.V. Klimov. Occam's Razor in Metacomputation: the Notion of a Perfect Process Tree. In *Proc. of 3rd Int'l Workshop on Static Analysis (WSA'93)*, pages 112–123. Springer LNCS 724, 1993.
- [18] M. Hanus. A unified computation model for functional and logic programming. In *Proc. of 24th ACM Symp. on Principles of Programming Languages (POPL'97)*, pages 80–93. ACM, New York, 1997.
- [19] M. Hanus. Curry: An Integrated Functional Logic Language. Available at: <http://www.informatik.uni-kiel.de/~curry/>, 2000.
- [20] M. Hanus, S. Antoy, J. Koj, R. Sadre, and F. Steiner. PAKCS 1.2: The Portland Aachen Kiel Curry System User Manual, 2000.
- [21] M. Hanus and C. Prehofer. Higher-Order Narrowing with Definitional Trees. *Journal of Functional Programming*, 9(1):33–75, 1999.
- [22] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Englewood Cliffs, NJ, 1993.
- [23] D.J. Kuck, R.H. Kuhn, D.A. Padua, B. Leasure, and M. Wolfe. Dependence Graphs and Compiler Optimization. In *Proc. of the 8th Symp. on the Principles of Programming Languages (POPL'81), SIGPLAN Notices*, pages 207–218, 1981.
- [24] J.-L. Lassez, M. J. Maher, and K. Marriott. Unification Revisited. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 587–625. Morgan Kaufmann, Los Altos, Ca., 1988.
- [25] M. Leuschel. On the Power of Homeomorphic Embedding for Online Termination. In *Proc. of the 5th Int'l Static Analysis Symposium (SAS'98)*, pages 230–245. Springer LNCS 1503, 1998.
- [26] J.W. Lloyd and J.C. Shepherdson. Partial Evaluation in Logic Programming. *Journal of Logic Programming*, 11:217–242, 1991.
- [27] W. Lux and H. Kuchen. An Efficient Abstract Machine for Curry. In *Proc. of the 8th Int'l Workshop on Functional and Logic Programming (WFLP'99)*, pages 171–181, 1999.
- [28] A.P. Nemytykh, V.A. Pinchuk, and V.F. Turchin. A Self-Applicable Supercompiler. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation. Dagstuhl Castle, Germany, February 1996*, pages 322–337. Springer LNCS 1110, 1996.
- [29] T. Reps and T. Turnidge. Program Specialization via Program Slicing. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation. Dagstuhl Castle, Germany, February 1996*, pages 409–429. Springer LNCS 1110, 1996.
- [30] S. Schoenig and M. Ducasse. A Backward Slicing Algorithm for Prolog. In *Proc. of Int'l Static Analysis Symposium (SAS'96)*, pages 317–331. Springer LNCS 1145, 1996.
- [31] M.H. Sørensen and R. Glück. An Algorithm of Generalization in Positive Supercompilation. In *Proc. of the 1995 Int'l Logic Programming Symposium (ILPS'95)*, pages 465–479. The MIT Press, Cambridge, MA, 1995.
- [32] M.H. Sørensen, R. Glück, and N.D. Jones. A Positive Supercompiler. *Journal of Functional Programming*, 6(6):811–838, 1996.
- [33] G. Szilagyi, T. Gyimothy, and J. Maluszynski. Static and Dynamic Slicing of Constraint Logic Programs. *J. Automated Software Engineering*, 9(1):41–65, 2002.
- [34] F. Tip. A Survey of Program Slicing Techniques. *Journal of Programming Languages*, 3:121–189, 1995.
- [35] V.F. Turchin. Program Transformation by Supercompilation. In *Proc. of the Int'l Workshop on Programs as Data Objects 1985*, pages 257–281. Springer LNCS 217, 1986.
- [36] G. Vidal. Cost-Augmented Narrowing-Driven Specialization. In *Proc. of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'02)*, pages 52–62, New York, 2002. ACM Press.
- [37] G. Vidal. Forward Slicing by Partial Evaluation, 2002. Available from URL <http://www.dsic.upv.es/~gvidal>.
- [38] M.D. Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.