Measuring the Effectiveness of Partial Evaluation[⋆] [⋆⋆]

Elvira Albert[1], Sergio Antoy[2], and Germán Vidal[1]

[1] DSIC, Technical University of Valencia, {ealbert,gvidal}@dsic.upv.es
[2] Department of Computer Science, Portland State University, antoy@cs.pdx.edu

Abstract. We introduce a framework for assessing the effectiveness of partial evaluators for functional logic programs. Our framework is based on properties of the rewrite system that models a functional logic program and, consequently, our assessment is independent of any specific language implementation or computing environment. We define several criteria for measuring the cost of a computation: number of steps, number of function applications, and effort for pattern matching. Most importantly, we express the cost of each criterion by means of recurrence equations over algebraic data types, which can be automatically inferred from the partial evaluation process itself. In some cases, the equations can be solved by transforming their arguments from arbitrary data types to natural numbers. In other cases, it is possible to estimate the improvement of a partial evaluation by analyzing the recurrence equations.

1 Introduction

Partial evaluation (PE) is a source-to-source program transformation technique for specializing programs w.r.t. parts of their input (hence also called *program specialization*). This technique has been studied, among others, in the context of functional [8, 14], logic [10, 15], and functional logic programming languages [4]. A common motivation of all PE techniques is to improve the efficiency of a program while preserving its meaning. Rather surprisingly, relatively little attention has been paid to the development of formal methods for reasoning about the effectiveness of this program transformation; usually, only experimental tests on particular languages and compilers are undertaken. Clearly, a machine-independent way of measuring the effectiveness of PE would be useful to both users and developers of partial evaluators.

Predicting the speedup achieved by partial evaluators is generally undecidable. Andersen and Gomard's [5] *speedup analysis* predicts a relative interval of

[⋆] This work has been partially supported by CICYT TIC 98-0445-C03-01 and the National Science Foundation under grant INT-9981317.

[⋆⋆] Extended Abstracts of LOPSTR 2000, Tenth International Workshop on Logic-based Program Synthesis and Transformation, 24-28 July 2000, London, UK. Technical Report Series, Department of Computer Science, University of Manchester, ISSN 1361-6161. Report number UMCS-00-6-1. http://www.cs.man.ac.uk/cstechrep/titles00.html

the speedup achieved by a program specialization. Nielson's [16] type system formally expresses when a partial evaluator is better than another. Other interesting works study *cost analyses* for logic and functional programs which may be useful for determining the effectiveness of program transformations [9, 17].

All these efforts mainly base the cost of executing a program on the number of steps performed in a computation. However, simple experiments show that the number of steps and the computation time are not easily correlated.

Example 1. Consider the well-known operation app to concatenate lists: [1]

```
app [] y          = y
app (x1 : xs) y = x1 : app xs y
```

and the following PE (obtained by the partial evaluator INDY [2]):

```
app2s [] y               = y
app2s (x : []) y        = x : y
app2s (x1 : x2 : xs) y = x1 : x2 : (app2s xs y)
```

This residual program computes the same function as the original but in approximately half the number of steps. This might suggest that the execution time of app2s (for sufficiently large inputs) should be about one half the execution time of app. However, executions of function app2s in several environments (e.g., in the lazy functional language Hugs [13] and the functional logic language Curry [12]) show that speedup is hardly measurable, i.e., between 1% and 2%.

In order to reason about these counterintuitive results, we introduce several formal criteria to measure the efficiency of a computation. We consider *inductively sequential* rewrite systems as programs and *needed narrowing* [6] as operational semantics (a call-by-name mechanism that has been proved *optimal* in functional logic programming). We formally define the cost of evaluating an expression in terms of the number of steps, the number of function applications, and the complexity of pattern-matching or unification involved in the computation. Similar criteria are taken into account (though experimentally) in traditional profiling approaches (e.g., [18]). The above criteria are useful to estimate the speedup achieved by the narrowing-driven PE scheme of [4]. In particular, we use *recurrence equations* to compare the cost of executing the original and residual programs. These equations can be automatically derived from the PE process itself. Unlike traditional recurrence equations used to reason about the complexity of programs, our equations are defined on data structures rather than on natural numbers. This complicates the computation of their solutions, although in some cases useful statements about the improvements achieved by PE can be made by a simple inspection of the sets of equations. In other cases, these equations can be transformed into ordinary recurrence equations and then solved by well-known mathematical methods.

An extended version of this paper can be found in [3].

[1] Although we consider in this work a first-order language, we use a curried notation in the examples (as is usual in functional languages).

2 Preliminaries

For the sake of completeness, we recall in this section some basic notions of term rewriting [7] and functional logic programming [11]. We consider a (*many-sorted*) *signature* Σ partitioned into a set $\mathcal{C}$ of *constructors* and a set $\mathcal{F}$ of (defined) *functions* or *operations*. The set of *terms* and *constructor terms* with *variables* (e.g., x, y, z) from $\mathcal{V}$ are denoted by $\mathcal{T}(\mathcal{C} \cup \mathcal{F}, \mathcal{V})$ and $\mathcal{T}(\mathcal{C}, \mathcal{V})$, respectively. The set of variables occurring in a term t is denoted by $\mathcal{V}ar(t)$. A term is *linear* if it does not contain multiple occurrences of one variable.

A *pattern* is a term $f(d_1, \ldots, d_n)$ where $f/n \in \mathcal{F}$ and $d_1, \ldots, d_n \in \mathcal{T}(\mathcal{C}, \mathcal{V})$. A *position* p in a term t is represented by a sequence of natural numbers. $t|_p$ denotes the *subterm* of t at position p, and $t[s]_p$ denotes the result of *replacing the subterm* $t|_p$ by the term s. We denote a *substitution* σ by $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ with $\sigma(x_i) = t_i$ for $i = 1, \ldots, n$ (with $x_i \neq x_j$ if $i \neq j$), and $\sigma(x) = x$ for all other variables x. A substitution σ is *constructor*, if $\sigma(x)$ is a constructor term for all x. The identity substitution is denoted by $\{\,\}$. A term t' is a (constructor) *instance* of t if there is a (constructor) substitution σ with $t' = \sigma(t)$.

A set of rewrite rules $l \to r$ such that $l \notin \mathcal{V}$, and $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$ is called a *term rewriting system* (TRS). Terms l and r are called the *left-hand side* (*lhs*) and the *right-hand side* (*rhs*) of the rule, respectively. A TRS $\mathcal{R}$ is *left-linear* if l is linear for all $l \to r \in \mathcal{R}$. A TRS is *constructor-based* if each lhs l is a pattern. A functional logic *program* is a left-linear constructor-based TRS. A *rewrite step* is an application of a rewrite rule to a term, i.e., $t \to_{p,R} s$ if there exists a position p in t, a rewrite rule $R = l \to r$ and a substitution σ with $t|_p = \sigma(l)$ and $s = t[\sigma(r)]_p$. Given a relation $\to$, we denote by $\to^+$ its transitive closure, and by $\to^*$ its transitive and reflexive closure.

To evaluate terms containing variables, *narrowing* non-deterministically instantiates the variables so that a rewrite step is possible. Formally, $t \leadsto_{(p,R,\sigma)} t'$ is a *narrowing step* if p is a non-variable position in t and $\sigma(t) \to_{p,R} t'$. We denote by $t_0 \leadsto^n_\sigma t_n$ a sequence of n narrowing steps $t_0 \leadsto_{\sigma_1} \ldots \leadsto_{\sigma_n} t_n$ with $\sigma = \sigma_n \circ \cdots \circ \sigma_1$. Due to the presence of free variables, an expression may be reduced to different values after instantiating free variables to different terms. Given a narrowing derivation $t_0 \leadsto^*_\sigma t_n$, we say that t_n is a computed *value* and σ is a computed *answer* for t_0. To avoid unnecessary narrowing computations and to provide computations with infinite data structures, the most recent work has advocated *lazy narrowing strategies*. *Needed narrowing* [6] is based on the idea of evaluating only subterms which are *needed* in order to compute a result.

3 Formal Criteria for Measuring Computational Cost

The cost criteria that we introduce in this section are independent of the particular implementation of the language. Rather, they are formulated for a rewrite system, which we intend as a program, and are based on operations that are, in one form or another, performed by likely implementations of rewriting and narrowing. For simplicity, we do not consider non-deterministic computations, although our cost measures could be extended along the lines of [9]. On the other hand, it is often the case that PE methods cannot significantly change the non-determinism of computations, i.e., the *search spaces* for a given goal in the original and residual programs have essentially the same structure.

The first cost criterion that we consider has been widely used in the literature. This is the *number of steps*, or *length*, of the evaluation of a term.

Definition 1 (number of steps). *We denote by $\mathcal{S}$ a function on rewrite rules, called the number of steps, as follows. If R is a rewrite rule, then $\mathcal{S}(R) = 1$.*

The second cost criterion is the number of symbol *applications* that occur within a computation. Counting applications is interesting because, in most implementations of a functional logic language, an evaluation will execute some machine instructions that directly correspond to each symbol application. The following definition bundles together all applications. It can be easily specialized to constructor or defined symbol applications only, denoted by $\mathcal{A}_c$ and $\mathcal{A}_d$ respectively.

Definition 2 (number of applications). *We denote by $\mathcal{A}$ a function on rewrite rules, called the number of applications. If $R = l \to r$ is a rewrite rule, then $\mathcal{A}(R)$ is the number of occurrences of non-variable symbols in r.*

The above definition is appropriate for a first-order language in which function applications are not curried. In a fully curried language, $\mathcal{A}(l \to r)$ would be one less the number of symbols in r (including variables).

The third cost criterion abstracts the effort performed by *pattern matching*. We assume that the number of rewrite rules in a program does not affect the efficiency of a computation. The reason is that in a first-order language a reference to the symbol being applied can be resolved at compile-time. However, when a defined operation f is applied to arguments, in non-trivial cases, one needs to inspect (at run-time) certain occurrences of certain arguments of the application of f to determine which rewrite rule of f to fire.

Definition 3 (pattern matching effort). *We denote by $\mathcal{P}$ a function on rewrite rules, called pattern matching effort, as follows. If $R = l \to r$ is a rewrite rule, then $\mathcal{P}(R)$ is the number of constructor symbols in l.*

In the following, we denote by C any cost criterion, i.e., C stands for $\mathcal{S}$, $\mathcal{A}$ or $\mathcal{P}$.

The previous definitions establish the cost of a derivation as the total cost of its steps: given a narrowing derivation $E : t \leadsto_{(p_1,R_1,\sigma_1)} \ldots \leadsto_{(p_n,R_n,\sigma_n)} u$, then $C(E) = \sum_{i=1}^{n} C(R_i)$. However, it is more convenient to reason about efficiency when a cost measure is defined over terms rather than entire computations. We use "computation" as a generic word for the *evaluation* of a term, i.e., a narrowing derivation ending in a constructor term. In general, different strategies applied to a same term may produce evaluations of different lengths and/or fail to terminate. For instance, if term t contains uninstantiated variables, there may exist distinct evaluations of t obtained by distinct instantiations of t's variables. Luckily, the *needed* narrowing strategy gives us some leeway. We allow uninstantiated variables in a term t as long as these variables are not instantiated by some evaluation, $t \leadsto^*_{\{\}} c$, of t. In this case, the value and answer computed by any other needed narrowing derivation of t are c and $\{\}$, respectively. Therefore, we fix a concrete strategy of needed narrowing, i.e., that denoted by λ in [6]. In the following, we consider that the *cost* of term t, $C(t)$, is n iff there exists a needed narrowing derivation $t \leadsto_{(p_1,R_1,\sigma_1)} \ldots \leadsto_{(p_k,R_k,\sigma_k)} c$, where c is a constructor term, $\sigma_k \circ \cdots \circ \sigma_1 = \{\,\}$, and $n = \sum_{i=1}^{k} C(R_i)$.

Example 2. Continuing Ex. 1, the next table summarizes (with minor approximations to ease understanding) the cost of computations with both functions:

	$\mathcal{S}$	$\mathcal{A}_c$	$\mathcal{A}_d$	$\mathcal{P}$
`app`	n	n	n	n
`app2s`	$0.5\,n$	n	$0.5\,n$	n
`other`	$2\,n$	n	$2\,n$	$2\,n$

Here n denotes the *size* of the inputs to the functions, i.e., the number of elements in the first argument of `app` and `app2s`. The third row represents the overhead for constructing the input and evaluating the output of either operation (see [3]). If we assume equal unit cost for each criterion, the total cost of the computations with `app` is $11\,n$. Likewise, the cost of computations with `app2s` is $10\,n$. The comparison gives a speedup of only 9%. If we increase the unit cost of $\mathcal{A}_c$ and decrease that of $\mathcal{S}$—probably a more realistic choice—the improvement of `app2s` over `app` estimated by our criteria closely explains the disappointing speedup measured experimentally (between 1% and 2%).

4 Measuring the Effectiveness of a Partial Evaluation

In this section, we are concerned with the problem of determining the improvement achieved by a PE in the context of a functional logic language.

4.1. Narrowing-driven PE. Here we briefly recall some basic notions of the narrowing-driven PE scheme of [4] for the specialization of inductively sequential programs based on needed narrowing.

Roughly speaking, given a program $\mathcal{R}$ and a set of finite (possibly incomplete) narrowing trees for a set of calls S, a PE of S in $\mathcal{R}$ is obtained in two stages. (i) Compute an *independent renaming* ρ for the terms in S, where ρ is mapping from terms to terms such that, for all $s \in S$, $\rho(s) = f(x_1, \ldots, x_n)$, $x_1, \ldots, x_n$ are the distinct variables of s, and f is a *fresh* function symbol. The renaming of residual rules is necessary to ensure the generation of *legal* program rules (i.e., with linear patterns in the lhs's) and to remove some redundant structures. To rename the rhs's of residual rules, the auxiliary function ren_ρ is introduced. It *recursively* replaces each call in a given expression by a call to the corresponding renamed function (according to ρ). (ii) Construct a renamed resultant $\sigma(\rho(s)) \to ren_\rho(t)$, for each narrowing derivation $s \leadsto^+_\sigma t$ in the considered narrowing trees.

Following [15], we adopt the convention that any derivation is potentially incomplete. A *failing derivation* is a needed narrowing derivation ending in an expression that is neither a constructor term nor can be further narrowed.

Definition 4 (partial evaluation). *Let $\mathcal{R}$ be a program, $S = \{s_1, \ldots, s_n\}$ a finite set of terms, and ρ an independent renaming of S. Let $\mathcal{N}_1, \ldots, \mathcal{N}_n$ be finite needed narrowing trees for s_i in $\mathcal{R}$, $i = 1, \ldots, n$. A partial evaluation of S in $\mathcal{R}$ (under ρ) is obtained by constructing a renamed resultant, $\sigma(\rho(s)) \to ren_\rho(t)$, for each non-failing needed narrowing derivation $s \rightsquigarrow^+_\sigma t$ in $\mathcal{N}_1, \ldots, \mathcal{N}_n$.*

Example 3. Consider again the function `app` together with the set of calls:

$$S = \{\texttt{app (app x}_\texttt{s}\texttt{ y}_\texttt{s}\texttt{) z}_\texttt{s}\texttt{), app x}_\texttt{s}\texttt{ y}_\texttt{s}\}\ .$$

An independent renaming ρ for S is the mapping:

$$\{\texttt{app x}_\texttt{s}\texttt{ y}_\texttt{s} \mapsto \texttt{app1s x}_\texttt{s}\texttt{ y}_\texttt{s}\texttt{, app (app x}_\texttt{s}\texttt{ y}_\texttt{s}\texttt{) z}_\texttt{s} \mapsto \texttt{dapp x}_\texttt{s}\texttt{ y}_\texttt{s}\texttt{ z}_\texttt{s}\}\ .$$

A possible partial evaluation of S in $\mathcal{R}$ (under ρ) is:

```
dapp [] ys zs       = app1s ys zs
dapp (x : xs) ys zs = x : dapp xs ys zs
app1s [] ys         = ys
app1s (x : xs) ys   = x : app1s xs ys
```

4.2. Automatic Generation of Recurrence Equations. The development of this section is inspired by the standard use of recurrence equations to analyze the complexity of algorithms in terms of their inputs (see, e.g., [1] for imperative, [17] for functional, and [9] for logic programs). We present a technique for deriving recurrence equations which parallels the construction of resultants:

Definition 5. *Let $\mathcal{R}$ be a program, S a finite set of terms, and ρ an independent renaming for S. Let $\mathcal{R}'$ be a PE of S in $\mathcal{R}$ (under ρ) computed from the finite needed narrowing trees $\mathcal{N}_1, \ldots, \mathcal{N}_n$. Then, we produce a pair of equations*

$$C(\sigma(s)) \;=\; C(t) + k \quad / \quad C(\sigma(\rho(s))) \;=\; C(ren_\rho(t)) + k'$$

for each needed narrowing derivation $s \rightsquigarrow^+_\sigma t$ in $\mathcal{N}_1, \ldots, \mathcal{N}_n$.[2] *Constants k and k' denote the observable cost of the considered derivation in the original and residual programs, respectively, i.e., $k = C(s \rightsquigarrow^+_\sigma t)$ and $k' = C(\rho(s) \rightsquigarrow_\sigma ren_\rho(t))$.*

Example 4. Consider the operation `app` of Ex. 1. Given the narrowing derivation:

$$s = \mathsf{app\ (app\ x\ y)\ z} \rightsquigarrow_{\{\mathsf{x}\mapsto \mathsf{x}':\mathsf{x_s}\}} \mathsf{app\ (x'\ :\ app\ x_s\ y)\ z} \rightsquigarrow_{\{\,\}} \mathsf{x'\ :\ app\ (app\ x_s\ y)\ z} = t$$

which produces the residual rule:

$$\underbrace{\texttt{dapp (x}'\texttt{ : x}_\texttt{s}\texttt{) y z}}_{\sigma(\rho(s))} = \underbrace{\texttt{x}'\texttt{ : dapp x}_\texttt{s}\texttt{ y z}}_{ren_\rho(t)}$$

with $\sigma = \{x \mapsto x' : x_s\}$ and $\rho = \{\texttt{app (app x y) z} \mapsto \texttt{dapp x y z}\}$, we get:

$$\begin{cases} \mathcal{S}(\sigma(\texttt{s})) \;=\; \mathcal{S}(\texttt{t}) + 2 \quad / \quad \mathcal{S}(\sigma(\rho(\texttt{s}))) \;=\; \mathcal{S}(\texttt{ren}_\rho(\texttt{t})) + 1 \\ \mathcal{A}(\sigma(\texttt{s})) \;=\; \mathcal{A}(\texttt{t}) + 4 \quad / \quad \mathcal{A}(\sigma(\rho(\texttt{s}))) \;=\; \mathcal{A}(\texttt{ren}_\rho(\texttt{t})) + 2 \\ \mathcal{P}(\sigma(\texttt{s})) \;=\; \mathcal{P}(\texttt{t}) + 2 \quad / \quad \mathcal{P}(\sigma(\rho(\texttt{s}))) \;=\; \mathcal{P}(\texttt{ren}_\rho(\texttt{t})) + 1 \end{cases}$$

[2] Note that there is no risk of ambiguity in using the same symbols for both equations, since the signatures of $\mathcal{R}$ and $\mathcal{R}'$ are disjoint by definition of PE.

The generated equations are correct in the sense that each equation (locally) holds w.r.t. the original definitions of each cost criterion (see Th. 1 in [3]).

Reasoning about recurrence equations of this kind is not easy. The problem comes from the laziness of the computation model, since interesting cost criteria are not compositional for non-strict semantics [17]. In particular, the cost of evaluating an expression $f(e)$ will depend on how much function f needs argument e. The following condition of *closedness* avoids this problem.

Definition 6. *Let $\mathcal{R}$ be a program, S a finite set of terms, and ρ an independent renaming for S. Let $\mathcal{R}'$ be a PE of S in $\mathcal{R}$ (under ρ). Let E be the set of recurrence equations computed according to Def. 5. Then, we say that E is closed iff for each pair of equations: $C(\sigma(s)) = C(t)+k \;\; / \;\; C(\sigma(\rho(s))) = C(ren_\rho(t))+k'$, t is a constructor instance of some term in S.*

The relevance of closed recurrence equations stems from their use to *compute* the cost of a term. In particular, given a term t that is a constructor instance of the lhs of some equation, if $t \leadsto^*_{\{\}} u$ is a needed narrowing derivation for t and $C(t) = n$, then $C(t)$ can be *computed* by rewriting, i.e., $C(t) \to^* n$ using the recurrence equations and the definition of the addition "+" (see Th. 2 in [3]).

In practice, recurrence equations are not usually closed. However, some simplification rules can be applied to achieve this form in many cases. For instance, the following (trivial) properties can be used: i) $C(x) = 0$, for all $x \in \mathcal{V}$, and ii) $C(c(t_1, \ldots, t_n)) = C(t_1) + \ldots + C(t_n)$, for all $c \in \mathcal{C}$, with $n \geq 0$.

In some cases, recurrence equations can be transformed into ordinary recurrence equations over natural numbers and then solved by well-known mathematical methods (see [3] for some examples). In other cases, we can still extract useful information about the overall speedup of a program from its loops, since sufficient long runs will consume most of the execution time inside loops. In our method, loops are represented by recurrence equations. Therefore, they constitute by themselves a useful aid for determining the speedup (or *slowdown*) associated to each loop. Actually, for each residual rule $\sigma(\rho(s)) \to ren_\rho(t)$ with a pair of associated recurrence equations: $C(s) = C(t) + k \;/\; C(\rho(s)) = C(ren_\rho(t)) + k'$, we can add the set of tuples (k, k') which describe its variation. For instance, given the partial derivation of Ex. 4, we produce the (decorated) residual rule:

$$\mathtt{dapp}\ (\mathtt{x}' : \mathtt{x_s})\ \mathtt{y}\ \mathtt{z}\ =\ \mathtt{x}' : \mathtt{dapp}\ \mathtt{x_s}\ \mathtt{y}\ \mathtt{z} \qquad /\!*\ \{(2,1),(4,2),(2,1)\}\ *\!/$$

From this information, the user can easily see that all cost criteria have been improved (and also quantify this improvement).

5 Conclusions and Future Work

To the best of our knowledge, this is the first attempt to formally measure the effectiveness of PE with cost criteria different from the number of evaluation steps. Our characterization of cost enables us to estimate the effectiveness of a PE in a precise framework. We also provide an automatic method to infer some recurrence equations which allow us to reason about the improvement achieved.

Both contributions mixed together help us to reconcile theoretical results and experimental speedups. Although the introduced notions and techniques are tailored to narrowing-driven PE, they could be transferred to other related PE methods (e.g., positive supercompilation [19] or partial deduction [15]).

There are several possible directions for further research. On the practical side, we plan to develop an analytical tool for estimating the improvements achieved by residual programs. On the theoretical side, we will investigate which cost criteria are always improved by PE and which are not.

References

1. A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, MA, 1974.
2. E. Albert, M. Alpuente, M. Falaschi, and G. Vidal. INDY User's Manual. Technical Report DSIC-II/12/98, UPV, 1998.
3. E. Albert, S. Antoy, and G. Vidal. A Formal Approach for Reasoning about the Effectiveness of Partial Evaluation. Technical Report DSIC, UPV, 2000. Available from URL: `http://www.dsic.upv.es/users/elp/papers.html`.
4. M. Alpuente, M. Hanus, S. Lucas, and G. Vidal. Specialization of Functional Logic Programs Based on Needed Narrowing. *ACM Sigplan Notices*, 34(9):273–283, 1999.
5. L.O. Andersen and C.K. Gomard. Speedup Analysis in Partial Evaluation: Preliminary Results. In *Proc. of PEPM'92*, pages 1–7. Yale University, 1992.
6. S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. In *Proc. 21st ACM Symp. on Principles of Programming Languages*, pages 268–279, 1994.
7. F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.
8. C. Consel and O. Danvy. Tutorial notes on Partial Evaluation. In *Proc. of POPL'93*, pages 493–501. ACM, New York, 1993.
9. S.K. Debray and N.W. Lin. Cost Analysis of Logic Programs. *ACM Transactions on Programming Languages and Systems*, 15(5):826–975, 1993.
10. J. Gallagher. Tutorial on Specialisation of Logic Programs. In *Proc. of PEPM'93*, pages 88–98. ACM, New York, 1993.
11. M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, 19&20:583–628, 1994.
12. M. Hanus (ed.). Curry: An Integrated Functional Logic Language. Available at `http://www-i2.informatik.rwth-aachen.de/~hanus/curry`, 2000.
13. M.P. Jones and A. Reid. The Hugs 98 User Manual. Available at `http://haskell.cs.yale.edu/hugs/`, 1998.
14. N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, Englewood Cliffs, NJ, 1993.
15. J.W. Lloyd and J.C. Shepherdson. Partial Evaluation in Logic Programming. *Journal of Logic Programming*, 11:217–242, 1991.
16. F. Nielson. A Formal Type System for Comparing Partial Evaluators. In *Proc. of Int'l Ws on PE and Mixed Computation*, pages 349–384. N-H, 1988.
17. D. Sands. A Naive Time Analysis and its Theory of Cost Equivalence. *Journal of Logic and Computation*, 5(4):495–541, 1995.
18. P.M. Sansom and S.L. Peyton-Jones. Formally Based Profiling for Higher-Order Functional Languages. *ToPLaS*, 19(2):334–385, 1997.
19. M.H. Sørensen, R. Glück, and N.D. Jones. A Positive Supercompiler. *Journal of Functional Programming*, 6(6):811–838, 1996.