Symbolic Profiling for Multi-Paradigm Declarative Languages *

Elvira Albert and Germán Vidal

DSIC, UPV, Camino de Vera s/n, E-46022 Valencia, Spain
{ealbert,gvidal}@dsic.upv.es

Abstract. We present the basis of a source-level profiler for multi-paradigm declarative languages which integrate features from (lazy) functional and logic programming. Our profiling scheme is *symbolic* in the sense that it is independent of the particular language implementation. This is achieved by counting the number of *basic* operations performed during the execution of program calls, e.g., the number of unfolding steps, the number of matching operations, etc. The main contribution of this paper is the formal specification of the attribution of execution costs to *cost centers*, which is particularly difficult in the context of lazy languages. A prototype implementation of the symbolic profiler has been undertaken for the multi-paradigm language Curry. Preliminary results demonstrate the practicality of our approach and its applications in the field of program transformation.

1 Introduction

Profiling tools, in general, are designed for assisting the programmer in the task of generating efficient code (see, e.g., [12]). 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 field of program transformation, in particular, we believe that profiling techniques can play an important role:

1. *The most immediate application consists of using the information gathered by the profiler to assess the effectiveness of a program transformation.* This can be done by simply comparing the cost information obtained for the original and the transformed programs. For instance, as we shall see later, we use our profiler to decide whether the optimization of a program function actually provides an improvement over the original function.
2. *In the context of* automatic *program transformation techniques, profiling tools can assist the transformation process in the task of identifying those program calls which are promising candidates to be optimized.* For instance, partial evaluation is an automatic program transformation technique which

* This work has been partially supported by CICYT TIC 2001-2705-C03-01, by Acción Integrada Hispano-Alemana HA2001-0059, by Acción Integrada Hispano-Austriaca HU2001-0019, and by Acción Integrada Hispano-Italiana HI2000-0161.

specializes a given program w.r.t. part of its input data [22]. Some partial evaluators require the user to annotate the program calls to be optimized by partial evaluation (e.g., [4]). In these cases, profiling tools could play an important role in order to automate the process, e.g., they could help in the task of detecting which are the most expensive functions and, thus, promising candidates to be partially evaluated. However, note that the most expensive functions are not the only interesting ones to be partially evaluated. In fact, sometimes the specialization of computationally inexpensive—but often called functions—can lead to dramatic speedups.

3. *A further step along the lines of the previous point would be to integrate the profiler within a program transformation tool.* For instance, the extension of a partial evaluator with the computation of cost information may be useful to determine the improvement achieved by a particular transformation process. A first step towards this direction can be found in [29], where the partial evaluator returns not only the set of residual rules, but also the cost improvement achieved by each rule. The computation of cost information during partial evaluation could also be used to *guide* the specialization process, e.g., to decide when to evaluate and when to residualize an expression depending on the computed costs.

Our work is concerned with the development of a profiling scheme to assist automatic transformation techniques for multi-paradigm declarative languages. Recent proposals for multi-paradigm declarative programming amalgamate principles from functional, logic and concurrent programming [16]. The resulting language includes the most important features of the distinct paradigms, e.g., *lazy* evaluation, higher-order functions, nondeterministic computations, concurrent evaluation of constraints with synchronization on logical variables, and a unified computation model which integrates *narrowing* and *residuation* [16].

Rather surprisingly, there are very few profiling tools for high-level declarative languages. The reason can be found in the difficulty to relate low-level operations with high-level code [27]. This is particularly difficult in the presence of a lazy evaluation mechanism, since the execution of nested calls is interleaved and each part of the execution must be attributed to the right function call.

Two features characterize our profiler: the basic scheme 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, we do 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.[1] Moreover, the user may annotate the source program with *cost centers* to which execution costs are attributed. We define a formal specification, or *cost semantics*,

[1] Nevertheless, the experiments in Section 5 indicate that the speedup predicted using our symbolic cost criteria is a good approximation of the real speedup measured experimentally.

to describe the attribution of costs to cost centers. The specification is given for programs in *flat* form, a sort of intermediate representation used during the compilation of (higher-level) source programs [9, 17, 25]. Nevertheless, we could say that our profiler is defined at a source-level since the transformation from high-level to flat programs consists mainly in a "desugaring" process. The flat representation was originally designed to provide a common interface for connecting different tools working on functional logic programs. Furthermore, the definition is general enough to cover also other declarative languages, e.g., purely functional or logic languages. This allows people working on programming tools for similar languages (e.g., compiler back-ends, program optimizers) to develop them on the basis of the intermediate representation so that they can exchange or integrate such tools.

This paper establishes a setting in which one can discuss costs attribution, formal properties about costs, or the effects of some program transformations in the context of multi-paradigm declarative languages. Moreover, we provide some preliminary results from the prototype implementation of a profiler for the multi-paradigm declarative language Curry [19]. They show evidence of the practicality of our approach and its applications in the field of program transformation. For instance, a partial evaluator has been recently developed for Curry programs [3, 4]. This partial evaluation tool starts out from a program with some annotated function calls and produces (potentially) more efficient, residual definitions for the annotated function calls. Our profiler has been greatly useful to estimate the effectiveness of this partial evaluator, i.e., to determine the efficiency speedups achieved by the partial evaluator for a given annotated program (we refer here to the application stressed in the first item at the beginning of this section). As we will see in Section 5, the idea is to simply perform the profiling of those expressions previously annotated to be partially evaluated (i.e., *cost centers* are introduced on the same expressions to be specialized). Thus, the profiling results allow us to assess whether significant performance improvements have been achieved by the specialization phase. A further step would be to use the profiler for deciding on which expressions the annotations should be introduced (as discussed in the second item at the beginning of this section). This application is a subject of ongoing research.

The rest of the paper is organized as follows. Section 2 informally describes the source language as well as its operational semantics. Section 3 introduces some symbolic cost criteria and outlines the profiling scheme. The specification of our cost semantics for profiling is formally described in Section 4. Section 5 summarizes our experiments with an implementation of the profiler for Curry. Section 6 compares with related work. Finally, Section 7 concludes and points out several directions for further research.

2 The Source Language

We consider modern multi-paradigm languages which integrate the most important features of functional and logic programming (like, e.g., Curry [19] or Toy

[24]). In our source language, functions are defined by a sequence of rules:

$$f(c_1, \ldots, c_n) \;=\; t$$

where $c_1, \ldots, c_n$ are constructor terms and the right-hand side t is an arbitrary term. *Constructor terms* may contain variables and constructor symbols, i.e., symbols which are not defined by the program rules. Functional logic programs can be seen as term rewriting systems fulfilling several conditions (e.g., inductively sequential systems [8]). Several implementations allow the use of a number of additional features, like higher-order functions, (concurrent) constraints, external (built-in) calls, monadic I/O, nondeterministic functions, etc. We do not describe these features here but refer to [19].

Example 1. Let us consider the following rules defining the well-known function `append` (where `[]` denotes the empty list and `x:xs` a list with first element `x` and tail `xs`):[2]

```
append eval flex
append []     ys = ys
append (x:xs) ys = x : append xs ys
```

The evaluation annotation "`eval flex`" declares `append` as a *flexible* function which can also be used to *solve* equations over functional expressions (see below). For instance, the equation "`append p s =:= [1,2,3]`" is solved by instantiating the variables `p` and `s` to lists so that their concatenation results in the list `[1,2,3]`.

The basic operational semantics of our source language is based on a combination of needed narrowing and residuation [16]. The *residuation* principle is based on the idea of delaying function calls until they are ready for a deterministic evaluation (by rewriting). Residuation preserves the deterministic nature of functions and naturally supports concurrent computations. On the other hand, the *narrowing* mechanism allows the instantiation of free variables in expressions and, then, applies reduction steps to the instantiated expressions. This instantiation is usually computed by unifying a subterm of the entire expression with the left-hand side of some program rule. To avoid unnecessary computations and to deal with infinite data structures, demand-driven generation of the search space has recently been advocated by a flurry of outside-in, lazy narrowing strategies. Due to some optimality properties, *needed narrowing* [8] is currently the best lazy narrowing strategy for functional logic programs.

The precise mechanism—narrowing or residuation—for each function is specified by *evaluation annotations*. The annotation of a function as *rigid* forces the delayed evaluation by rewriting, while functions annotated as *flexible* can be evaluated in a nondeterministic manner by applying narrowing steps. For instance, in the language Curry [19], only functions of result type "`Constraint`" are considered flexible (and all other functions rigid). Nevertheless, the user can

[2] Although we consider a first-order representation for programs, we use a curried notation in the examples as it is common practice in functional languages.

explicitly provide different evaluation annotations. To provide concurrent computation threads, expressions can be combined by the *concurrent conjunction operator* "`&`," i.e., the expression $\mathtt{e}_1$ `&` $\mathtt{e}_2$ can be reduced by evaluating either $\mathtt{e}_1$ or $\mathtt{e}_2$.

3 Symbolic Profiling

Our profiling scheme relies on the introduction of some symbolic costs which are attributed to specific *cost centers*. Our symbolic costs are used to represent the *basic* operations performed during the evaluation of expressions. They are coherent with the cost criteria introduced in [1, 10] to measure the cost of functional logic computations. The main novelty lies in the introduction of a particular criterium to measure the creation and manipulation of nondeterministic branching points (which are similar to choice points in logic programming). The costs of a particular functional logic computation are attributed by adding the costs of performing *each step of the computation*. Thus, we only define the symbolic costs associated to the application of a single rule:

- *Number of unfolding steps.* Trivially, the number of steps associated to the application of a program rule is 1.
- *Number of pattern matching operations.* It is defined as the number of constructor symbols in the left-hand side of the applied rule.
- *Number of nondeterministic steps.* This cost abstracts the work needed either to create, update or remove a choice point. It is equal to 0 when the function call matches exactly one program rule, and 1 otherwise. Thus, in principle, we do not accumulate the work performed in previous choices. Nevertheless, in the implementation, we wish to be more flexible and allow the user to indicate (by means of a flag) whether the cost of former nondeterministic branches should also be accrued to the current branch. This is particularly interesting in the presence of failing derivations, as will be shown in Section 5. Unfortunately, the formalization of this approach is not easy: by simply accumulating the cost of all previous derivations to the current one, we probably count the same costs several times since, for instance, two different derivations may share all the steps but the final one. In order to overcome this problem, a new semantics definition which takes into account the *search strategy* is required. This is subject of ongoing work (see [2]).
- *Number of applications.* Following [10], we define the number of applications associated to applying a program rule as the number of non-variable symbols (of arity greater than zero) in the right-hand side of this rule, plus the arities of these symbols. This cost intends to measure the time needed for the storage that must be allocated for executing a computation, i.e., for allocating the expressions in the right-hand sides of the rules.

Profilers attribute execution costs to "parts" of the source program. Traditionally, these "parts" are identified with functions or procedures. Following [27], we take a more flexible approach which allows us to associate a *cost center* with

each expression of interest. This provides the programmer with the possibility of choosing an appropriate granularity for profiling, ranging from whole program phases to single subexpressions in a function. Nevertheless, our approach can be easily adapted to work with automatically instrumented cost centers. For instance, if one wants to use the traditional approach in which all functions are profiled, we can automatically annotate each function by introducing a cost center for the entire right-hand side (since, in the flat representation, each function is defined by a single rule). Cost centers are marked with the (built-in) function `SCC` (for Set Cost Center). For instance, given the program excerpt:

```
length_app x y = length (append x y)
length [] = 0
length (x:xs) = (length xs) + 1
```

which uses the definition of `append` in Example 1, one can introduce the following annotations:

```
length_app x y = SCC cc2 (length (SCC cc1 (append x y)))
```

In this way, the cost of evaluating "`append x y`" is attributed to the cost center `cc1`, while the cost of evaluating "`length (append x y)`" is attributed to the cost center `cc2` by excluding the costs already attributed to `cc1`.

We informally describe the attribution of costs to cost centers as follows. Given an expression "`SCC cc e`", the costs attributed to `cc` are the entire costs of evaluating `e` as far as the enclosing context demands it,

- also *including* the cost of evaluating any function called during the evaluation of the expression `e`,
- but *excluding* the cost of evaluating any `SCC`-expressions within `e` or within any function called from `e`.

For example, given the initial call "`length_app (1:2:3:x) (4:5:[])`", the first result that we compute is `5`, with computed answer $\{\texttt{x} \mapsto []\}$ and associated symbolic costs:

`cc1`: 4 steps (one for each element of the first input list, plus an additional step to apply the base case, which instantiates `x` to `[]`), 4 pattern matchings (one for each applied rule, since there is only one constructor symbol in the left-hand sides of the rules defining `append`), 1 nondeterministic branching point (to evaluate the call `append x (4:5:[])`), and 18 applications (6 applications for each element of the first input list).

`cc2`: 6 steps (one for each element of the concatenated list, plus an additional step to apply the base case), 6 pattern matchings (one for each applied rule, since there is only one constructor symbol in the left-hand sides of the rules defining `length`), 0 nondeterministic branching points (since the computation is fully deterministic), and 25 applications (5 applications for each element of the list).

The next section gives a formal account of the attribution of costs.

4 Formal Specification of the Cost Semantics

This section formalizes a semantics enhanced with cost information for performing symbolic profiling. Our cost-augmented semantics considers programs written in *flat* form. In this representation, all functions are defined by a single rule, whose left-hand side contains only different variables as parameters, and the right-hand side contains case expressions for pattern-matching. Thanks to this representation, we can define a simpler operational semantics, which will become essential to simplify the definition of the associated profiling scheme. The syntax for programs in the flat representation is formalized as follows:

$\mathcal{R} ::= D_1 \ldots D_m$	$e ::= x$		(variable)
$D ::= f(x_1, \ldots, x_n) = e$	$\mid$	$c(e_1, \ldots, e_n)$	(constructor)
	$\mid$	$f(e_1, \ldots, e_n)$	(function call)
$p ::= c(x_1, \ldots, x_n)$	$\mid$	$case\ e_0\ of\ \{p_1 \to e_1; \ldots; p_n \to e_n\}$	(rigid case)
	$\mid$	$fcase\ e_0\ of\ \{p_1 \to e_1; \ldots; p_n \to e_n\}$	(flexible case)
	$\mid$	$SCC(cc, e)$	(SCC-construct)

where $\mathcal{R}$ denotes a program, D a function definition, p a pattern and e an arbitrary expression. We write $\overline{o_n}$ for the *sequence of objects* $o_1, \ldots, o_n$. A program $\mathcal{R}$ consists of a sequence of function definitions D such that the left-hand side is linear and has only variable arguments, i.e., pattern matching is compiled into case expressions. The right-hand side of each function definition is an expression e composed by variables, constructors, function calls, case expressions, and SCC-constructs. The general form of a case expression is:

$$(f)\, case\ e\ of\ \{c_1(\overline{x_{n_1}}) \to e_1; \ldots; c_k(\overline{x_{n_k}}) \to e_k\}$$

where e is an expression, $c_1, \ldots, c_k$ are different constructors of the type of e, and $e_1, \ldots, e_k$ are arbitrary expressions. The variables $\overline{x_{n_i}}$ are local variables which occur only in the corresponding subexpression e_i. The difference between *case* and *fcase* only shows up when the argument e is a free variable: *case* suspends (which corresponds to residuation) whereas *fcase* nondeterministically binds this variable to the pattern in a branch of the case expression (which corresponds to narrowing).

Example 2. By using case expressions, the rules of Example 1 defining the function `append` can be represented by a single rule as follows:

```
append x y = fcase x of { []     → y ;
                          (z:zs) → z : append zs y }
```

The standard semantics for flat programs is based on the LNT calculus (Lazy Narrowing with definitional Trees [18]). In Fig. 1, we recall from [5] an extension of the original LNT calculus able to cope with the combination of needed narrowing and residuation (i.e., the operational semantics informally described in Section 2). The calculus is defined by the one-step transition relation $\Rightarrow_\sigma$, which is labeled with the substitution σ computed in the step. The application of the substitution σ to an expression e is denoted by $\sigma(e)$. Let us informally describe the rules of the LNT calculus:

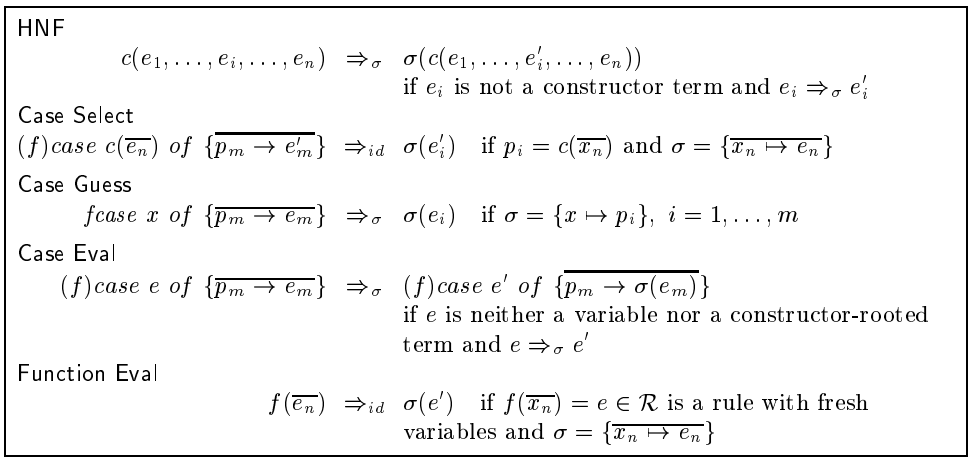

Fig. 1. LNT calculus

HNF. This rule can be applied to evaluate expressions in head normal form (i.e., rooted by a constructor symbol). It proceeds by recursively evaluating some argument (e.g., the leftmost one) that contains unevaluated function calls. Note that when an expression contains only constructors and variables, there is no rule applicable and the evaluation stops successfully.

Case Select. It simply selects the appropriate branch of a case expression and continues with the evaluation of this branch. The step is labeled with the identity substitution *id*.

Case Guess. This rule applies when the argument of a *flexible* case expression is a variable. Then, it nondeterministically binds this variable to a pattern in a branch of the case expression. We additionally label the step with the computed binding. Note that there is no rule to evaluate a *rigid* case expression with a variable argument. This situation produces a *suspension* of the evaluation (i.e., an abnormal termination).

Case Eval. This rule can be only applied when the argument of the case construct is a function call or another case construct. Then, it tries to evaluate this expression. If an evaluation step is possible, we return the original expression with the argument updated. The step is labeled with the same substitution computed from the evaluation of the case argument, which is also propagated to the different case branches.

Function Eval. Finally, this rule performs the unfolding of a function call.

This semantics is properly augmented with cost information, as defined by the state-transition rules of Fig. 2. The state consists of a tuple $\langle ccc, k, e\rangle$, where ccc is the current cost center, k is the accumulated cost, and e is the expression to be evaluated. An initial state has the form $\langle CC_0, K_0, e\rangle$, where CC_0 is the *initial* cost center—to which all costs are attributed unless an SCC construct specifies a different cost center—, K_0 is the *empty* cost, and e is the initial expression to be evaluated. Costs are represented by a set of cost variables $\in \{S, C, N, A\}$

rule	ccc	cost	expression	$\Rightarrow$	ccc	cost	expression
scc_1	cc	k	$SCC(cc', e)$	$\Rightarrow_\sigma$	cc	k'	$SCC(cc', e')$
	if e is a nonvariable expression which is not rooted by a constructor nor an SCC, and $cc'\ k\ e \ \Rightarrow_\sigma \ cc'\ k'\ e'$						
scc_2	cc	k	$SCC(cc', c(e_1, \ldots, e_n))$	$\Rightarrow_{id}$	cc	k	$c(SCC(cc', e_1), \ldots, SCC(cc', e_n))$
scc_3	cc	k	$SCC(cc', x)$	$\Rightarrow_{id}$	cc	k	x
scc_4	cc	k	$SCC(cc', SCC(cc'', e))$	$\Rightarrow_{id}$	cc	k	$SCC(cc'', e)$
hnf	$cc\ k\ c(e_1, \ldots, e_i, \ldots, e_n)$			$\Rightarrow_\sigma$	$cc\ k'\ \sigma(c(e_1, \ldots, e'_i, \ldots, e_n))$		
	if e_i is not a constructor term and $cc\ k\ e_i \ \Rightarrow_\sigma \ cc'\ k'\ e'_i$						
c_select	$cc\ k\ (f)case\ c(\overline{e_n})\ of\ \{\overline{p_m \to e'_m}\}$			$\Rightarrow_{id}$	$cc\ k'\ \sigma(e'_i)$		
	if $p_i = c(\overline{x_n})$, $\sigma = \{\overline{x_n \mapsto e_n}\}$, and $k' = k[C_{cc} \leftarrow C_{cc} + 1]$						
c_guess1	$cc\ k\ fcase\ x\ of\ \{\overline{p_m \to e_m}\}$			$\Rightarrow_\sigma$	$cc\ k'\ \sigma(e_i)$		
	if $\sigma = \{x \mapsto p_i\}$, $m = 1$, and $k' = k[C_{cc} \leftarrow C_{cc} + 1]$						
c_guess2	$cc\ k\ fcase\ x\ of\ \{\overline{p_m \to e_m}\}$			$\Rightarrow_\sigma$	$cc\ k'\ \sigma(e_i)$		
	if $\sigma = \{x \mapsto p_i\}$, $m > 1$, and $k' = k[C_{cc} \leftarrow C_{cc} + 1,\ N_{cc} \leftarrow N_{cc} + 1]$						
c_eval	$cc\ k\ (f)case\ e\ of\ \{\overline{p_m \to e_m}\}$			$\Rightarrow_\sigma$	$cc\ k'\ (f)case\ e'\ of\ \{\overline{p_m \to \sigma(e_m)}\}$		
	if e is neither a variable nor an operation-rooted term and $cc\ k\ e \ \Rightarrow_\sigma \ cc'\ k'\ e'$						
fun_eval	$cc\ k\ f(\overline{e_n})$			$\Rightarrow_{id}$	$cc\ k'\ \sigma(e)$		
	if $f(\overline{x_n}) = e \in \mathcal{R}$ is a rule with fresh variables, $\sigma = \{\overline{x_n \mapsto e_n}\}$, and $k' = k[S_{cc} \leftarrow S_{cc} + 1, A_{cc} \leftarrow A_{cc} + size(e)]$						

Fig. 2. Cost-augmented LNT calculus

indexed by the existing cost centers. Thus, given a cost center cc, S_{cc} records the number of steps attributed to cc, C_{cc} the number of case evaluations (or *basic* pattern matching operations) attributed to cc, N_{cc} the number of nondeterministic branching points attributed to cc, and A_{cc} the number of applications attributed to cc.

Let us briefly describe the cost variations due to the application of each state-transition rule of Fig. 2:

SCC rules. These rules are used to evaluate expressions rooted by an SCC symbol. The first rule, scc_1, applies to expressions of the form $SCC(cc', e)$, where e is a nonvariable expression which is not rooted by a constructor nor by an SCC symbol. Basically, it performs an evaluation step on e and, then, it returns the current cost center cc' unchanged. Rules scc_2 and scc_3 propagate SCC symbols to the arguments of a constructor-rooted expression, or remove them when the enclosed expression is a variable or a constructor constant. Note that rule scc_2 does not increment the current cost with a "constructor application". This happens because the considered language is first-order and, thus, the cost of constructor (or function) applications is only considered when performing a function unfolding (in rule *fun_eval*) in order

to allocate the symbols in the right-hand sides of the rules. Finally, scc_4 is used to remove an SCC construct when the argument is another SCC construct. These rules could be greatly simplified under an eager evaluation semantics. In particular, we would only need the rule:

$$\langle cc, k, SCC(cc', e)\rangle \quad \Rightarrow_\sigma \quad \langle cc, k', b\rangle \quad \text{if} \quad \langle cc', k, e\rangle \quad \Rightarrow^*_\sigma \quad \langle cc', k', b\rangle$$

where b is a value, i.e., a constructor term containing no unevaluated functions. Thus, the main difference is that, within an eager calculus, inner subexpressions could be fully evaluated and the SCC constructs could be just dropped; meanwhile, in our lazy calculus, we must carefully keep and appropriately propagate the SCC construct since the evaluation of an inner subexpression could be demanded afterwards.

HNF. The *hnf* rule tries to evaluate some argument (of the constructor-rooted term) which contains unevaluated function calls. If an evaluation step is possible, we return the original expression with the argument and the associated cost updated.

Case select. Rule *c_select* updates the current cost by adding one to C_{cc}, where cc is the current cost center.

Case guess. In this case, the current cost is always updated by adding one to C_{cc}, where cc is the current cost center, since a variable has been instantiated with a constructor term. Moreover, when there is nondeterminism involved, we should also increment cost variable N. In particular, we distinguish two cases:

c_guess1 : This rule corresponds to a case expression with a single branch. For instance, a function with a constructor argument defined by a single rule, like "`f 0 = 0`", will be represented in the flat representation as follows: `f x` = `fcase x of` $\{$`0` $\rightarrow$ `0`$\}$. Trivially, the application of this rule is fully deterministic, even if variable `x` gets instantiated and, thus, cost variable N is not modified.

c_guess2 : This rule corresponds to a case expression with several branches. In order to account for the nondeterministic step performed, we add 1 to N_{cc} where cc is the current cost center.

Case eval. The *c_eval* rule can be applied when the argument of the case construct is a function call, an SCC expression or another case construct. Then, it tries to evaluate this expression. If an evaluation step is possible, we return the original expression with the argument and the associated cost updated.

Function eval. The *fun_eval* rule updates the current cost k by adding one to S_{cc} and by adding $size(e)$ to A_{cc}, where cc is the current cost center and e is the right-hand side of the applied rule. Function $size$ counts the number of applications in an expression and it is useful to quantify memory usage. Following [10], given an expression e, a call to $size(e)$ returns the total number of occurrences of n-ary symbols, with $n > 0$, in e, plus their arities; of course, SCC symbols are not taken into account.

Arbitrary *derivations* are denoted by $\langle CC_0, K_0, e\rangle \Rightarrow^*_\sigma \langle cc, k, e'\rangle$, which is a shorthand for the sequence of steps $\langle CC_0, K_0, e\rangle \Rightarrow_{\sigma_1} \ldots \Rightarrow_{\sigma_n} \langle cc, k, e'\rangle$ with

$\sigma = \sigma_n \circ \cdots \circ \sigma_1$ (if $n = 0$ then $\sigma = id$). We say that a derivation $\langle CC_0, K_0, e\rangle \Rightarrow^*_\sigma \langle cc, k, e'\rangle$ is *successful* when e' contains no function calls (i.e., it is a constructor term). Then, we say that e evaluates to e' with computed answer σ and associated cost k. Regarding non-successful derivations (i.e., suspended or failing derivations), we simply return the suspended (or failing) expression together with the cost associated to the performed steps. This information may be useful when considering nondeterministic computations, as we will see in Section 5.

We note that, in order to be sound with current implementations of functional logic languages, one should consider the effect of *sharing*. By using a sharing-based implementation of the language, the system avoids repeated computations when the same expression is demanded several times. The definition of a sharing-based semantics for functional logic programs in flat form is subject of ongoing work [2].

Let us illustrate our cost-augmented LNT calculus by means of an example. In Fig. 3, we detail a successful derivation for the expression:

```
length_app (1 : x) []
```

using the following definitions for `length_app` and `len` in flat form:

```
length_app x y = len (app x y)
len x  = fcase x of { []     → 0 ;
                      (y:ys) → (len ys) + 1 }
```

Here, we use `len` and `app` as shorthands for `length` and `append`, respectively. We annotate each transition in the derivation with the name of the applied rule. Also, the values of the current cost center appear over the derived expression. In some steps, we write `[]` to denote that there is no modification of the current costs. We notice that the symbol "`+`" is considered as a built-in operator of the language and, thus, we treat it similarly to constructor symbols. This explains the fact that, in some steps, the symbol SCC is propagated to its arguments.

There is a precise equivalence between the cost semantics of Fig. 2 and the attribution of costs as explained in Section 3. To be precise, the number of steps, pattern matching operations and nondeterministic branching points coincide in both cases. There exists, however, a significant difference regarding the number of applications. Consider, for instance, function `append`. From the evaluation of "`append [] []`" in the source language we compute 0 applications, since the first rule of `append` contains no symbol of arity greater than zero. However, the same evaluation in the flat language produces 6 applications, since the right-hand side of the rule depicted in Example 2 comprises the right-hand sides of the two source rules. For the development of a profiling tool, both alternatives are possible.

5 Experimental Evaluation

The practicality of the ideas presented in this work is demonstrated by the implementation of a symbolic profiler for the multi-paradigm language Curry

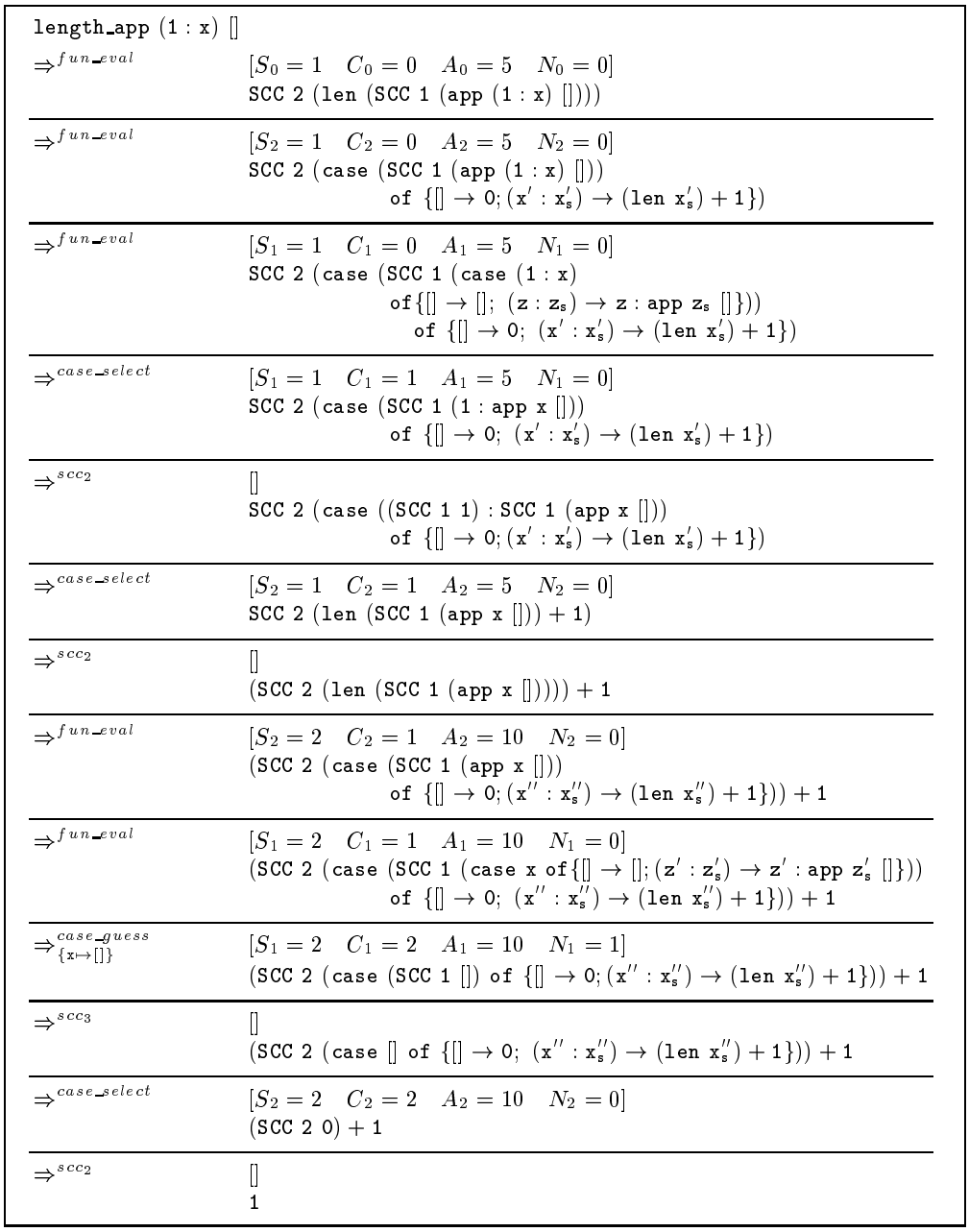

Fig. 3. Example of LNT derivation for "`length_app (1:x) []`"

[19].[3] Firstly, in order to have a useful profiling tool, one has to extend the basic scheme to cover all the high-level features of the language (concurrent constraints, higher-order functions, calls to external functions, I/O, etc). This extension has not been especially difficult starting out from the basic scheme of Section 4. An interesting aspect of our profiling tool is that it is completely written in Curry, which simplifies further extensions as well as its integration into existing program transformation tools. Source programs are automatically translated into the flat syntax by using the facilities provided by the module `Flat` of PAKCS [17] for meta-programming in Curry.

Our prototype implementation is basically a meta-interpreter for the language Curry enhanced with cost information as described throughout the paper. Thus, given its interpretive nature, it cannot be directly used to profile "real" applications. Nevertheless, in its present form, it may serve to check alternative design choices for the formal specification of Fig. 2 that, otherwise, would be impossible to understand and explain clearly. Moreover, it has been of great help to assess the effectiveness of a partial evaluator for Curry programs [3, 4]. Below we show several experiments which illustrate the usefulness of our profiling tool.

Our first example shows a traditional use of the developed tool to assist the programmer in the task of deciding whether the optimization of a program function will actually improve program performance. The source program is as follows:

```
append []     ys = ys
append (x:xs) ys = x : append xs ys
filter p []       = []
filter p (x:xs)   = if p x then x : filter p xs else filter p xs
qsort []     = []
qsort (x:l)  = append (qsort (filter (<x) l))
                      (x : qsort (filter (>=x) l))
rev [] = []
rev (x:xs) = append (rev xs) [x]
foo1 x = SCC cc1 (qsort (SCC cc2 (rev x)))
foo2 x = SCC cc3 (append (SCC cc4 (rev x)) [])
```

In this program, we have considered two possible uses of function `rev` in the body of `foo1` and `foo2`. Profiling has been applied to the initial calls:

```
foo1 [0,1,2,3]
foo2 [0,1,2,3]
```

The profiling results for these calls are shown in the first four rows of Table 1 (*naive*). For each benchmark, the columns show the cost center, the symbolic costs for this cost center (steps, pattern matchings and applications,[4] respectively), and the actual runtime (by using larger input lists to obtain significant runtimes). Note that this does not mean that our symbolic costs are not directly

[3] Available at: `http://www.dsic.upv.es/users/elp/profiler.`

[4] Nondeterministic branching points are ignored in this example since computations are fully deterministic.

	benchmark	cost center	steps	pat_mat	apps	runtime
naive	`foo1`	`cc1`	130	119	1466	1660 ms.
		`cc2`	25	25	245	
	`foo2`	`cc3`	5	5	45	340 ms.
		`cc4`	15	15	155	
optimized	`foo1`	`cc1`	130	119	1466	1640 ms.
		`cc2`	6	5	49	
	`foo2`	`cc3`	5	5	45	20 ms.
		`cc4`	6	5	49	

Table 1. Profiling results

related to runtimes, but simply that, given the interpretive nature of the profiler, it cannot handle the large inputs needed to obtain a measurable runtime. Times are expressed in milliseconds and are the average of 10 executions on a 800 MHz Linux-PC (Pentium III with 256 KB cache). All programs (including the profiler) were executed with the Curry→Prolog compiler [9] of PAKCS.

Clearly, the information depicted in Table 1 is helpful to assist the user in finding out which program function may benefit from being optimized. For instance, if one intends to use `rev` as an argument of `qsort` (like in `foo1`), there is no urgent need to optimize `rev`, since it takes only a small percentage of the total costs. On the other hand, if the intended use is as an argument of `append`, then the optimization of `rev` may be relevant. In order to demonstrate this, we have replaced `rev` with a version of reverse with an accumulating parameter (which can be obtained by program transformation). The new profiling results are shown in the last four rows of Table 1 (*optimized*). By observing `foo1`, we notice that there is no significant runtime improvement (compare the first and fifth rows). However, the runtime of `foo2` is highly improved (compare the third and seventh rows).

Our second example illustrates the interest in computing failing derivations when nondeterministic computations arise. The source program is a simplified version of the classical map coloring program which assigns a color to each of three countries such that countries with a common border have different colors:

```
isColor Red    = success
isColor Yellow = success
isColor Green  = success
coloring x y z = isColor x & isColor y & isColor z
correct x y z = diff x y & diff x z & diff y z
gen_test x y z = SCC cc1 (coloring x y z & correct x y z)
test_gen x y z = SCC cc2 (correct x y z & coloring x y z)
```

Here, "`&`" is the concurrent conjunction, "`success`" is a predefined constraint which is always solvable, and the predefined function "`diff`" is the only rigid function (it makes use of the strict equality predicate in order to check whether its arguments are different, but no instantiation is allowed). We have included two functions `gen_test` and `test_gen` which implement the "generate & test" and the (much more efficient) "test & generate" solutions, respectively. Rather

surprisingly, the profiling results for any of the six possible solutions coincide for both functions: 8 steps, 3 matching operations, 3 nondeterministic steps, and 69 applications. These counterintuitive results are explained as follows. The actual improvement is not related to the *successful* derivations, which are almost identical in both cases, but to the number (and associated costs) of *failing* derivations. Thus, our current implementation considers two possibilities:

1. the computation of the costs for each derivation is independent of previous branches in the search tree (i.e., the calculus of Fig. 2);
2. the computation of the costs for each derivation also includes the costs of all previous branches in the search tree.

By using the second alternative, we can sum up the costs of the whole search space for each of the previous functions (including failing derivations):

`gen_test:` 192 steps, 81 pattern matchings, 81 nondeterministic branching points, and 1695 applications (for 27 derivations);
`test_gen:` 165 steps, 60 pattern matchings, 60 nondeterministic branching points, and 1428 applications (for 21 derivations).

From these figures, the greater efficiency of `test_gen` can be easily explained.

Finally, we have applied our profiling tool to check the improvements achieved by a partial evaluator of Curry programs [3, 4] over several well-known benchmarks. In particular, we have performed the profiling of those expressions previously annotated to be partially evaluated (i.e., *cost centers* are introduced on the same expressions to be specialized). Table 2 shows our profiling results for several well-known benchmarks of partial evaluation. Some of them are typical from deforestation (the case of `all_ones`, `app_last`, `double_app`, `double_flip`, `length_app`) and `kmp` is the well-known "KMP test".[5] For each benchmark, we show the number of steps, pattern matching operations and applications for the original and residual programs, respectively; we do not include information about the amount of nondeterminism since it is not changed by the partial evaluator. The last column of Table 2 shows the actual speedup (i.e., the ratio *original*/*residual*, where *original* refers to the runtime in the original program and *residual* is the runtime in the partially evaluated program). Runtime input goals were chosen to give a reasonably long overall time.

From the figures in Table 2, we observe that the cost information collected by the profiler allows us to quantify the potential improvement which has been achieved by the residual program. For instance, the more significant improvements on the symbolic costs are produced for the KMP benchmark which, indeed, shows an actual speedup of 12.94 for sufficiently large input goals. Fewer improvements have been obtained for the remaining benchmarks, which is also sensible with the minor speedups tested experimentally. To be more precise, one should determine the appropriate weight of each symbolic cost for a specific language environment. Nevertheless, our experiments show the potential usefulness of our profiler to check whether a sufficient reduction (w.r.t. the symbolic costs)

[5] The complete code of these benchmarks can be found, e.g., in [7].

Benchmark	Original			Residual			Speedup
	steps	pat_mat	apps	steps	pat_mat	apps	
`all_ones`	22	22	198	11	11	110	1.29
`app_last`	22	33	209	11	11	77	2.83
`double_app`	27	27	243	16	17	254	1.17
`double_flip`	18	18	234	9	9	117	1.25
`kmp`	101	160	1279	11	60	402	12.94
`length_app`	23	23	195	12	13	184	1.39

Table 2. Improvements by partial evaluation

has been achieved by partial evaluation and, thus, the residual program can be safely returned. A further step is the "smart" use of the profiling information to guide the partial evaluation process, which is subject of ongoing work.

6 Related Work

One of the most-often cited profilers in the literature is gprof—a call graph execution profiler developed by Graham et al. [15]. Apart from collecting information about call counts and execution time as most traditional profilers, gprof is able to account to each routine the time spent by the routines that it invokes. This accounting is achieved by assembling a call graph made up by nodes that are routines of the program and directed arcs that represent calls from call sites to routines. By post-processing these data, times are propagated along the edges of the graph to attribute time for routines to the routines they call. Although these ideas have been of great influence, their approach has not been directly transferred to profiling declarative programs (which heavily depend on recursion) mainly because programs that exhibit a large degree of recursion are not easily analyzed by gprof.

To the best of our knowledge, there is no previous work about profiling within the field of (narrowing-based) functional logic programming. The remaining of this section is devoted to briefly review some related research on profiling for the logic and functional programming paradigms separately.

As for logic programming, the profiler for Prolog developed by Debray [14] shows that traditional profiling techniques are inadequate for logic programming languages. In particular, this profiler is especially designed to gather information related to the control flow in logic programming languages, including backtracking and is able to deal with primitives like `assert` and `retract`. Our profiler shares with this approach some ideas for collecting information about nondeterministic branching points. A more recent reference is the work by Jahier and Ducassé [20, 21], which proposes a program execution monitor to gather data about Mercury executions. For this purpose, they define a high-level primitive built on top of the execution tracer for Mercury, which delivers information extracted from the current state at a particular point (e.g., execution depth, port, live arguments, etc). Our approach is more related with the design of the tracer,

in the sense that our profiler relies on an operational semantics which may also show the state transitions performed along the process, apart from collecting cost information. Regarding parallel logic programs, Tick [28] developed a performance analysis for estimating the parallelism exploited using "kaleidescope visualization". This technique consists in summarizing the execution of a program in a single image or signature. According to [28], there are two inputs to the algorithm: a trace file and a source program. The trace file input contains information about events logged in the trace. This information could be targeted by profiling techniques in the style of the one described in this paper. The implementation of profiling tools has been carried out for some logic programming languages. For instance, the ECLiPSe logic programming system contains a timing profiler which interrupts the program execution every 0.01s to measure the time spent in every predicate of a program. For the Mercury language, there exist two profiling tools: the `mprof` Mercury profiler—which is a conventional call-graph profiler in the style of the above `gprof` profiler—and the Mercury deep profiler `mdprof` [13]—which is a new profiler that associates with every profiling measurement a very detailed context about the ancestor functions or predicates and their call sites. This approach shares some ideas with cost center "stack" profiling [26], which we discuss below.

Modern functional profilers have been heavily influenced by the notion of *cost center*, which basically permits to attribute costs to program expressions rather than profiling every program function [26, 27]. Nevertheless, similar ideas (i.e., the concept of "current function") already appeared in the profiler for the SML of New Jersey [11]. Recent profilers for the non-strict lazy functional language Haskell have been developed relying on the notion of cost center. Along these lines, the profiling technique developed by Sansom and Peyton-Jones [27] is the closest to our work. A similarity with them is that we also present a formal specification of the attribution of execution costs to cost centers by means of an appropriate cost-augmented semantics. A significant difference, though, is that our flat representation for programs is first-order, contains logical features (like nondeterminism), and has an operational semantics which combines narrowing and residuation. A further extension of the cost center profiling is cost center "stack" profiling [26], which allows full inheritance of the cost of functions. The basic idea is to attribute the cost of executing a profiled function to the function *and* to the stack of functions responsible for the call. Our approach could also be extended towards this direction in order to develop a more flexible profiling tool.

It is worth mentioning the *monitor semantics* presented in [23]. This is a non-standard model of program execution that captures "monitoring activity" as found in debuggers, profilers, tracers, etc. Their framework is general enough in the sense that the monitoring semantics can be automatically obtained from any denotational semantics. It could be interesting to investigate whether this framework can also be applied in our context.

In the field of functional logic programming, there is a recent line of research which investigates techniques to estimate the effectiveness achieved by partial

evaluation. The work in [1] formally defines several abstract cost criteria to measure the effects of program transformations (the similarities with our symbolic costs are discussed throughout the paper). Their goal is more restrictive than the one behind profiling techniques. In particular, the purpose of [1] is to estimate the effectiveness achieved by a concrete residual program by means of some cost recurrence equations obtained along the partial evaluation process. A further step in the line of generating residual programs with cost information has been taken in [29]. This work introduces the scheme of a narrowing-driven partial evaluator enhanced with the computation of symbolic costs. Thus, for each residual rule, the new scheme provides the cost variation due to the partial evaluation process.

Finally, Watterson and Debray [30] propose an approach to reduce the cost of "value" profiling (i.e., a profiling technique which provides information about the runtime distribution of the values taken by a variable). Their approach avoids wasting resources where the profile can be guaranteed to not be useful for optimizations. We believe that such performance optimizations could also be tried in our framework.

7 Conclusions and Further Work

This paper investigates the definition of a source-level, symbolic profiling scheme for a multi-paradigm functional logic language. Our scheme is based on several symbolic costs (comprehensive with those of [1, 10] to measure the cost of functional logic computations) and uses the idea of cost centers to attribute costs to program expressions. The formalization of our scheme is carried out by a cost-augmented semantics, carefully designed for the flat representation of multi-paradigm declarative programs. An implementation of the profiling tool (extended to cover all the features of the language Curry) is presented. Preliminary results are encouraging and give evidence of the practicality of our approach and the benefits of using profiling information in the field of program transformation.

Future work includes two different lines of research. Currently, our main concern is to investigate the combination of the symbolic profiler with existing partial evaluation techniques for functional logic languages [6, 7] in order to "guide" the specialization process. We are also working on the definition of a new semantics characterization able to cope with all the features of modern multi-paradigm language implementations: sharing of common variables, nondeterminism, search strategies, concurrency, etc. A preliminary definition of such a semantics can be found in [2]. A cost extension of this enhanced semantics may be useful to collect cost information in the context of these realistic languages.

Acknowledgments

We gratefully acknowledge the anonymous referees for many useful suggestions and the participants of LOPSTR 2001 for fruitful feedback.

Part of this research was done while the authors were visiting the University of Kiel supported by the "Acción Integrada Hispano-Alemana HA2001-0059" project. We wish to thank Michael Hanus and Frank Huch for many helpful comments on the topics of this paper.

References

1. E. Albert, S. Antoy, and G. Vidal. Measuring the Effectiveness of Partial Evaluation in Functional Logic Languages. In *Proc. of LOPSTR 2000*, pages 103–124. Springer LNCS 2042, 2001.
2. E. Albert, M. Hanus, F. Huch, J. Oliver, and G. Vidal. An Operational Semantics for Declarative Multi-Paradigm Languages. Available from URL: `http://www.dsic.upv.es/users/elp/papers.html`, 2002.
3. E. Albert, M. Hanus, and G. Vidal. Using an Abstract Representation to Specialize Functional Logic Programs. In *Proc. of LPAR 2000*, pages 381–398. Springer LNAI 1955, 2000.
4. E. Albert, M. Hanus, and G. Vidal. A Practical Partial Evaluator for a Multi-Paradigm Declarative Language. In *Proc. of FLOPS'01*, pages 326–342. Springer LNCS 2024, 2001.
5. E. Albert, M. Hanus, and G. Vidal. A Residualizing Semantics for the Partial Evaluation of Functional Logic Programs, 2002. Submitted for publication. Available at `http://www.dsic.upv.es/users/elp/papers.html`.
6. E. Albert and G. Vidal. The Narrowing-Driven Approach to Functional Logic Program Specialization. *New Generation Computing*, 20(1):3–26, 2002.
7. 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.
8. S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. *Journal of the ACM*, 47(4):776–822, 2000.
9. S. Antoy and M. Hanus. Compiling Multi-Paradigm Declarative Programs into Prolog. In *Proc. of FroCoS'2000*, pages 171–185. Springer LNCS 1794, 2000.
10. S. Antoy, B. Massey, and P. Julián. Improving the Efficiency of Non-Deterministic Computations. In *Proc. of WFLP'01*, Kiel, Germany, 2001.
11. A.W. Appel, B.F. Duba, and D.B. MacQueen. Profiling in the Presence of Optimization and Garbage Collection. Technical Report CS-TR-197-88, Princeton University, 1988.
12. J.L. Bentley. *Writing Efficient Programs*. Prentice-Hall, Englewood Cliffs, N.J., 1982.
13. T. C. Conway and Z. Somogyi. Deep profiling: engineering a profiler for a declarative programming language. Technical Report Technical Report 2001/24, Department of Computer Science, University of Melbourne (Australia), 2001.
14. S.K. Debray. Profiling Prolog Programs. *Software, Practice and Experience*, 18(9):821–840, 1988.
15. S.L. Graham, P.B. Kessler, and M.K. McKusick. gprof: a Call Graph Execution Profiler. In *Proc. of CC'82*, pages 120–126, Boston, MA, 1982.
16. M. Hanus. A unified computation model for functional and logic programming. In *Proc. of POPL'97*, pages 80–93. ACM, New York, 1997.
17. M. Hanus, S. Antoy, J. Koj, R. Sadre, and F. Steiner. PAKCS 1.2: The Portland Aachen Kiel Curry System User Manual. Technical report, University of Kiel, Germany, 2000.

18. M. Hanus and C. Prehofer. Higher-Order Narrowing with Definitional Trees. *Journal of Functional Programming*, 9(1):33–75, 1999.
19. M. Hanus (ed.). Curry: An Integrated Functional Logic Language. Available at: `http://www.informatik.uni-kiel.de/~mh/curry/`.
20. E. Jahier and M. Ducasse. A Generic Approach to Monitor Program Executions. In *Proc. of ICLP'99*, pages 139–153. MIT Press, 1999.
21. E. Jahier and M. Ducasse. Generic Program Monitoring by Trace Analysis. *Theory and Practice of Logic Programming*, 2(4,5), 2002.
22. N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Englewood Cliffs, NJ, 1993.
23. A. Kishon, P. Hudak, and C. Consel. Monitoring Semantics: A Formal Framework for Specifying, Implementing, and Reasoning about Execution Monitors. *ACM Sigplan Notices*, 26(6):338–352, 1991.
24. F. López-Fraguas and J. Sánchez-Hernández. TOY: A Multiparadigm Declarative System. In *Proc. of RTA'99*, pages 244–247. Springer LNCS 1631, 1999.
25. W. Lux and H. Kuchen. An Efficient Abstract Machine for Curry. In *Proc. of WFLP'99*, pages 171–181, 1999.
26. R.G. Morgan and S.A. Jarvis. Profiling large-scale lazy functional programs. *Journal of Functional Programming*, 8(3), 1998.
27. P.M. Sansom and S.L. Peyton-Jones. Formally Based Profiling for Higher-Order Functional Languages. *ACM Transactions on Programming Languages and Systems*, 19(2):334–385, 1997.
28. E. Tick. Visualizing Parallel Logic Programs with VISTA. In *Proc. of the Int'l Conf. on Fifth Generation Computer Systems*, pages 934–942. ICOT, 1992.
29. G. Vidal. Cost-Augmented Narrowing-Driven Specialization. In *Proc. of PEPM'02*, pages 52–62. ACM Press, 2002.
30. S. Watterson and S. Debray. Goal-Directed Value Profiling. In *Proc. of CC'01*, pages 319–334. Springer LNCS 2027, 2001.