# Source-Level Abstract 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 *abstract* 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., number of steps, number of matching operations, etc. The main contribution is the formal specification of the attribution of execution costs to *cost centers*, which is particularly difficult in the context of lazy languages. An implementation of the abstract profiler has been undertaken for the multi-paradigm language Curry. Preliminary results demonstrate the practicality of our approach.

## 1 Introduction

Profiling tools, in general, are designed for assisting the programmer in the task of generating efficient code (see, e.g., [10]). 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 in the following respects:

1. The most immediate application consists of using the information gathered by the profiler to assess the effectiveness of a program transformation tool, by simply comparing the cost information obtained for the original and the transformed programs.
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 specializes a given program w.r.t. part of its input data [18]. Usually, partial evaluators require the user to annotate the program calls to be optimized

⋆ This work has been partially supported by CICYT TIC 98-0445-C03-01 and by Acción Integrada Hispano-Italiana HI2000-0161.

by partial evaluation (e.g., [3]). In these cases, profiling tools could play an important role in order to automate the process, i.e., they could help in the task of detecting which are the most expensive functions and, thus, the best candidates to be partially evaluated.

3. A further step along the lines of the previous point would be to integrate the profiler within a program transformation tool (e.g., the definition of a "cost-guided" partial evaluator in which the decisions of evaluate/residualize expressions are made relying on the profiling results).

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 [23]. 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.

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 [13]. The resulting language includes the most important features of the distinct paradigms, e.g., 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* [13].

Two features characterize our profiler: the basic scheme is defined for the *source* language—hence, the attribution of costs to the user's program is straightforward—and it is *abstract*, 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 abstract measures: the number of computation steps, the number of allocated cells, the number of nondeterministic branching points, and the number of pattern matching operations. 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*, 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 multi-paradigm functional logic programs [7, 14, 21]. The flat representation was originally designed to provide a common interface for connecting different tools working on multi-paradigm functional logic programs. Furthermore, the definition is general enough to cover also other declarative languages, e.g., purely functional or logic languages. This allows that people working on language tools for similar languages (e.g., compiler back ends, program optimizers) develop them on the basis of the intermediate representation so that one 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 for multi-paradigm declarative languages. For instance, a partial evaluator has been recently developed for Curry programs [2, 3]. This partial evaluation tool starts out from a program with some annotated function calls and produces (hopefully) more efficient, residual definitions for the annotated function calls. Our profiler could be easily integrated with this partial evaluator by simply performing the profiling of those expressions previously annotated to be partially evaluated (i.e., cost centers are introduced on the same expressions to be improved by program transformation). The profiling results are thus useful to estimate whether significant performance improvement has been achieved by the proper specialization phase.

Finally, we provide some preliminary results from the implementation of a profiler for the multi-paradigm declarative language Curry [16]. They show evidence of the practicality of our approach.

## 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 [16] or Toy [20]). In our source language, functions are defined by a sequence of rules of the form:

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

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 [6]). 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 [16].

*Example 1.* 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`):[1]

```
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 be also 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 [13]. 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

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

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* [6] 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 non-deterministic manner by applying narrowing steps. For instance, in the multi-paradigm declarative language Curry [16], only functions of result type "`Constraint`" are considered flexible (and all other functions rigid). Nevertheless, the user can 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 reducing either $\mathtt{e}_1$ or $\mathtt{e}_2$.

## 3 Source-Level Abstract Profiling

Our profiling scheme relies on the introduction of some abstract costs—that represent the *basic* operations performed during the evaluation of expressions—which are attributed to specific *cost centers*. Our abstract costs are based on the cost criteria introduced in [1, 8] to measure the cost of functional logic computations. The main novelty lies in the introduction of a particular criterium to measure the creation 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 define the abstract cost associated to the application of a single rule:

- *Number of 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 branching points.* This cost is equal to 0 when the function call matches only one program rule. Otherwise, we count 1 when the first matching rule is applied, 2 for the second one, and so on. Thus, it abstracts the work needed either to create a choice point or to update it. Let us note that we assume a Prolog-like depth-first strategy.
- *Number of applications.* Following [8], 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 storage that must be

allocated for executing a computation, i.e., for allocating the expressions in the right-hand sides of the functions. According to [8], this cost definition is coherent with the implementation described in [7].

Profilers attribute execution costs to "parts" of the source program. Traditionally, these "parts" are identified with functions or procedures. Following [23], we take a more flexible approach which allows us to associate a *cost center* with each expression of interest. This provides to 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
```

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,

- *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 abstract 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 abstract profiling. Our cost-augmented semantics considers programs written in *flat* form. In the flat representation, we assume that all functions are defined by a single rule, whose left-hand side contains only 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 summarized as follows:

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

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:

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

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 [15]), which is properly extended to cope with

| rule | ccc cost expression | $\Rightarrow$ | ccc cost expression |
|---|---|---|---|
| *scc1* | $cc\ \ k\ \ SCC(cc', e)$ | $\Rightarrow_\sigma$ | $cc\ \ k'\ \ SCC(cc', e')$ |
| | if $e$ is a nonvariable expression which is not rooted by a constructor nor an $SCC$, and $cc'\ k\ e\ \Rightarrow_\sigma\ cc'\ k'\ e'$ | | |
| *scc2* | $cc\ \ k\ \ SCC(cc', c(e_1, \ldots, e_n))$ | $\Rightarrow_{id}$ | $cc\ \ k\ \ c(SCC(cc', e_1), \ldots, SCC(cc', e_n))$ |
| *scc3* | $cc\ \ k\ \ SCC(cc', x)$ | $\Rightarrow_{id}$ | $cc\ \ k\ \ x$ |
| *scc4* | $cc\ \ k\ \ SCC(cc', SCC(cc'', e))$ | $\Rightarrow_{id}$ | $cc\ \ k\ \ SCC(cc'', e)$ |
| *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 $k' = k[C_{cc} \leftarrow C_{cc} + 1,\ N_{cc} \leftarrow N_{cc} + i]$ | | |
| *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 a nonvariable expression which is not rooted by a constructor symbol 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. 1.** Cost-augmented LNT calculus

the combination of needed narrowing and residuation (the operational semantics informally described in Section 2). This semantics is properly augmented with cost information, as defined by the state-transition rules of Figure 1. 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 expression to be evaluated. Costs are represented by a set of cost variables $\in \{S, C, N, A\}$ 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 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 $cc$. The one-step transition relation $\Rightarrow_\sigma$ is labeled with the substitution $\sigma$ computed in the step. Let us briefly describe the state-transition rules.

The *scc* rules are used to evaluate $SCC$-expressions. The first rule, *scc1*, 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. Then, it performs an evaluation step on $e$ by fixing the current cost center to $cc'$. Rules *scc2* and *scc3* 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 $scc_2$ does not increment the current cost with a "constructor application". This happens because the considered language is first-order and, thus, we only account to constructor (or function) applications the cost accrued to the allocation of the right-hand sides (in rule *fun_eval*). Finally, $scc_4$ is used to remove an $SCC$ construct when the argument is another $SCC$ construct. Note that these rules could be greatly simplified under an eager evaluation semantics. In particular, we will 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.

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

The *c_select* (case select) rule simply selects the appropriate branch of a case expression and continues with the evaluation of this branch. This step is labeled with the identity substitution $id$. The current cost is updated by adding one to $C_{cc}$, where $cc$ is the current cost center.

The *c_guess* (case guess) rules apply when the argument of a flexible case expression is a variable. Then, these rules nondeterministically bind this variable to a pattern in a branch of the case expression. We additionally label the step with the computed binding. On the one hand, 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 more than one branch in the case expression, 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`", may 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, $N$ is not modified.

*c_guess2* : This rule corresponds to a case expression with several branches. If the $i$-th branch is selected, we add $i$ to $N_{cc}$, where $cc$ is the current cost center. In this way, we count the number of backtrackings performed. Here, we assume a Prolog-like depth-first strategy, which is also coherent with the way in which [11] handles with backtracking.

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).

The *c_eval* (case eval) rule can be only 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. 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.

Finally, the *fun_eval* rule performs the unfolding of a function call. The current cost $k$ is updated 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 [8], 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. This is coherent with, e.g., the implementation described in [7].

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 $\sigma$ 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.

There is a precise equivalence between the cost semantics of Figure 1 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. Roughly speaking, the proof of equivalence relies on an extension of the correctness proof of the original LNT calculus [15]. 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 gives 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, although we strongly believe that the latter one is more appropriate mainly because it is closer to actual implementations (like [7]).

## 5 Experimental Evaluation

The practicality of the ideas presented in this work is demonstrated by the implementation of an abstract profiler for the multi-paradigm language Curry [16].[2] Firstly, in order to have a useful profiling tool, one has to extend the basic scheme to cover all high-level features of the language (concurrent constraints, higher-order functions, calls to external functions, I/O, etc). This extension has

[2] It is publicly available at `http://www.dsic.upv.es/users/elp/soft.html`.

not been specially 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 [14] 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 Figure 1 that, otherwise, would be impossible to understand and explain clearly. Moreover, it could be integrated with minor changes into a partial evaluator for Curry programs [2, 3]. 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 actually improves 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`, respectively. 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 abstract costs for this cost center (steps, pattern matchings and applications,[3] respectively), and the actual runtime (by using larger input lists to obtain significant runtimes). Note that this does not mean that our abstract costs are not directly related to runtimes, but simply that the interpretive nature of the profiler prevents us from showing realistic comparisons between abstract costs and actual runtimes. 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

[3] 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

(including the profiler) were executed with the Curry→Prolog compiler [7] 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. 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 l1 l2 l3 = isColor l1 & isColor l2 & isColor l3
correct l1 l2 l3 = diff l1 l2 & diff l1 l3 & diff l2 l3
gen_test l1 l2 l3 = SCC cc1 (coloring l1 l2 l3 & correct l1 l2 l3)
test_gen l1 l2 l3 = SCC cc2 (correct l1 l2 l3 & coloring l1 l2 l3)
```

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 pattern matchings, 6 nondeterministic branching points, and 69 applications. This 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 the *failing* derivations. Thus, we have summed up the costs of the whole search space for each function (including failing derivations):

`gen_test`: 192 steps, 81 pattern matchings, 162 nondeterministic branching points, and 1695 applications (for 27 derivations);
`test_gen`: 165 steps, 60 pattern matchings, 120 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 improvement achieved by the partial evaluator of Curry programs over several well-known benchmarks. For instance, for the function `length_app` of Section 3, we get the following results: 42 steps, 42 pattern matchings, and 357 applications for the original program, and 21 steps, 22 pattern matchings, and 336 applications for the specialized program (considering a first input list of 20 elements). The runtime speedup is about 1.4 (560 ms. / 400 ms.) using a first input list of 200000 elements, which can be justified by our profiling results. The same situation occurs in many other examples typical of deforestation [24], which are successfully specialized by the partial evaluator. This indicates that the profiling tool could be integrated into the partial evaluator in order to check whether a sufficient reduction (w.r.t. the abstract costs) has been achieved and, thus, the residual program can be safely returned. This line of research 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. [12]. 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 this 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 [11] shows that traditional profiling techniques are inadequate for logic programming languages. In particular, this profiler is specially designed to gather information related to control flow in logic programming languages, including backtracking as well as 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 [17], which proposes a program execution monitor to gather data about Mercury executions. To do this, 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). By contrast, our approach is more related with the design of a 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.

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 [22, 23]. Nevertheless, similar ideas (i.e., the concept of "current function") already appeared in the profiler for the SML of New Jersey [9]. 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 [23] is the closer 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 nondeterministic computations), and has an operational semantics which combines narrowing and residuation. A further extension of the cost center profiling is cost center "stack" profiling [22], 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 be also extended towards this direction in order to developed a more accurate profiling tool.

Finally, it is worth mentioning the *monitor semantics* presented in [19]. 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 be also applied in our context.

## 7 Conclusions and Further Work

This paper investigates the definition of a source-level abstract profiling scheme for a multi-paradigm functional logic language. Our scheme is based on several abstract costs (like those of [1, 8] 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.

Future work includes two different lines of research. On the practical side, we plan to improve the efficiency of the profiler and to integrate it into the PAKCS programming environment for Curry. On the other hand, we will investigate its combination with existing partial evaluation techniques for functional logic languages [4, 5] in order to "guide" the specialization process.

**Acknowledgments**

We gratefully acknowledge the anonymous referees for many useful comments and suggestions.

## 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, and G. Vidal. Using an Abstract Representation to Specialize Functional Logic Programs. In *Proc. of LPAR'00*, pages 381–398. Springer LNAI 1955, 2000.
3. 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.
4. E. Albert and G. Vidal. The Narrowing-Driven Approach to Functional Logic Program Specialization. *New Generation Computing*, 2001. To appear.
5. 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.
6. S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. *Journal of the ACM*, 47(4):776–822, 2000.
7. S. Antoy and M. Hanus. Compiling Multi-Paradigm Declarative Programs into Prolog. In *Proc. of FroCoS'2000*, pages 171–185. Springer LNCS 1794, 2000.
8. S. Antoy, B. Massey, and P. Julián. Improving the Efficiency of Non-Deterministic Computations. In *Proc. of WFLP'01*, Kiel, Germany, 2001.
9. 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.
10. J.L. Bentley. *Writing Efficient Programs.* Prentice-Hall, Englewood Cliffs, N.J., 1982.
11. S.K. Debray. Profiling Prolog Programs. *Software, Practice and Experience*, 18(9):821–840, 1988.
12. 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.
13. M. Hanus. A unified computation model for functional and logic programming. In *Proc. of POPL'97*, pages 80–93. ACM, New York, 1997.
14. 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.
15. M. Hanus and C. Prehofer. Higher-Order Narrowing with Definitional Trees. *Journal of Functional Programming*, 9(1):33–75, 1999.

16. M. Hanus (ed.). Curry: An Integrated Functional Logic Language. Available at: `http://www.informatik.uni-kiel.de/~mh/curry/`.
17. E. Jahier and M. Ducasse. A Generic Approach to Monitor Program Executions. In *Proc. of ICLP'99*, pages 139–153. MIT Press, 1999.
18. N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Englewood Cliffs, NJ, 1993.
19. 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.
20. F. López-Fraguas and J. Sánchez-Hernández. TOY: A Multiparadigm Declarative System. In *Proc. of RTA'99*, pages 244–247. Springer LNCS 1631, 1999.
21. W. Lux and H. Kuchen. An Efficient Abstract Machine for Curry. In *Proc. of WFLP'99*, pages 171–181, 1999.
22. R.G. Morgan and S.A. Jarvis. Profiling large-scale lazy functional programs. *Journal of Functional Programming*, 8(3), 1998.
23. 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.
24. P.L. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.