Run-Time Profiling of Functional Logic Programs*

B. Brassel[1], M. Hanus[1], F. Huch[1], J. Silva[2], and G. Vidal[2]

[1] Institut für Informatik, CAU Kiel, Olshausenstr. 40, D-24098 Kiel, Germany.
{bb,mh,fhu}@informatik.uni-kiel.de
[2] DSIC, Tech. University of Valencia, Camino de Vera s/n, E-46022 Valencia, Spain.
{jsilva,gvidal}@dsic.upv.es

Abstract. In this work, we introduce a profiling scheme for modern functional logic languages covering notions like laziness, sharing, and non-determinism. Firstly, we instrument a natural (big-step) semantics in order to associate a symbolic cost to each basic operation (e.g., variable updates, function unfoldings, case evaluations). While this *cost semantics* provides a formal basis to analyze the cost of a computation, the implementation of a cost-augmented interpreter based on it would introduce a huge overhead. Therefore, we also introduce a sound transformation that instruments a program such that its execution—under the standard semantics—yields not only the corresponding results but also the associated costs. Finally, we describe a prototype implementation of a profiler based on the developments in this paper.

1 Introduction

The importance of profiling in improving the performance of programs is widely recognized. Profiling tools are essential for the programmer to analyze the effects of different source-to-source program manipulations (e.g., partial evaluation, specialization, optimization, etc). Despite this, one can find very few profiling tools for modern declarative languages. This situation is mainly explained by the difficulty to correctly map execution costs to source code, which is much less obvious than for imperative languages. In this work, we tackle the definition of a profiling scheme for modern functional logic languages covering notions like laziness, sharing, and non-determinism (like Curry [6] and Toy [13]); currently, there is no profiling tool practically applicable to such languages.

When profiling the run time of a given program, the results highly depend on the considered language implementation. However, computing actual run times is not always the most useful information for the programmer. Run times may help to detect that some function is expensive but they do not explain *why* it is expensive (e.g., is it called many times? Is it heavily non-deterministic?).

* This work was partially supported by the Spanish *Ministerio de Educación y Ciencia* under grant TIN2004-00231, by Generalitat Valenciana GRUPOS03/025, by the ICT for EU-India Cross-Cultural Dissemination Project ALA/95/23/2003/077-054, and by the German Research Council (DFG) under grant Ha 2457/5-1.

In order to overcome these drawbacks, we introduce a *symbolic* profiler which outputs the number of basic operations performed in a computation. For this purpose, we start from a natural semantics for functional logic programs [1] and instrument it with the computation of symbolic costs associated to the basic operations of the semantics: variable lookups, function unfoldings, case evaluations, etc. These operations are performed, in one form or another, by likely implementations of modern functional logic languages. Our *cost semantics* constitutes a formal model of the attribution of costs in our setting. Therefore, it is useful not only as a basis to develop profiling tools but also to analyze the costs of a program computation (e.g., to formally prove the effectiveness of some program transformation).

Trivially, one can develop a profiler by implementing an instrumented interpreter which follows the previous cost semantics. However, this approach is not useful in practice as it demands a huge overhead, making the profiling of realistic programs impossible. Thus, in a second step, we design a source-to-source transformation that instruments a program such that its execution—under the standard semantics—outputs not only the corresponding results but also the associated costs. We formally state the correctness of our transformation (i.e., the costs computed in a source program w.r.t. the cost semantics are equivalent to the costs computed in the transformed program w.r.t. the standard semantics). Finally, we describe a prototype implementation of a profiler for Curry programs based on the developments in this paper.

The main contributions of this work are the following. Firstly, we introduce a cost semantics for functional logic programs which covers laziness, sharing and non-determinism. This contrasts with [14], where logical features are not considered, and [2], where sharing is not covered (which drastically reduces its applicability). Secondly, we introduce a program transformation for instrumenting a program—so that its execution returns also the cost of the computation—and prove its correctness. We are not aware of any other transformation for lazy functional (logic) programs that is proved correct w.r.t. an associated cost semantics.

The paper is organized as follows. In the next section, we recall some foundations for understanding the subsequent developments. Section 3 informally introduces our model for profiling functional logic computations. Section 4 formalizes an instrumented semantics which also computes cost information. Section 5 introduces a transformation instrumenting programs to compute symbolic costs. Section 6 describes an implementation of a profiler for Curry programs and Section 7 illustrates its use by means of an example. Finally, Section 8 includes a comparison to related work and concludes. An extended version of this paper (including the proof of Theorem 1) can be found in [5].

2 Flat Programs

In this work we consider *flat* programs [9], a convenient standard representation of functional logic programs which makes explicit the pattern matching strategy

$P ::= D_1 \ldots D_m$	(program)	*Domains*	
$D ::= f(x_1, \ldots, x_n) = e$	(function definition)		
$e ::= x$	(variable)	$P_1, P_2, \ldots \in \mathit{Prog}$	(Programs)
$\mid\ c(x_1, \ldots, x_n)$	(constructor call)	$x, y, z, \ldots \in \mathit{Var}$	(Variables)
$\mid\ f(x_1, \ldots, x_n)$	(function call)	$a, b, c, \ldots \in \mathcal{C}$	(Constructors)
$\mid\ \mathit{let}\ x = e_1\ \mathit{in}\ e_2$	(let binding)	$f, g, h, \ldots \in \mathcal{F}$	(Functions)
$\mid\ e_1\ \mathit{or}\ e_2$	(disjunction)	$p_1, p_2, \ldots \in \mathit{Pat}$	(Patterns)
$\mid\ \mathit{case}\ x\ \mathit{of}\ \{\overline{p_k \to e_k}\}$	(rigid case)		
$\mid\ \mathit{fcase}\ x\ \mathit{of}\ \{\overline{p_k \to e_k}\}$	(flexible case)		
$p ::= c(x_1, \ldots, x_n)$	(pattern)		

Fig. 1. Syntax for normalized flat programs

by case expressions. This flat representation constitutes the kernel of modern functional logic languages like Curry [8, 6] or Toy [13]. We assume that flat programs are *normalized*, i.e., *let* constructs are used to ensure that the arguments of function and constructor calls are always variables (not necessarily pairwise different). As in [12], this is essential to express sharing without the use of complex graph structures. A normalization algorithm can be found in [1]. Basically, normalization introduces one new let construct for each non-variable argument, e.g., $f(e)$ is transformed into "$\mathit{let}\ x = e\ \mathit{in}\ f(x)$".

The syntax for normalized flat programs is shown in Figure 1, where we write $\overline{o_n}$ for the *sequence of objects* $o_1, \ldots, o_n$. A program consists of a sequence of function definitions such that the left-hand side has pairwise different variable arguments. The right-hand side is an expression composed by variables, data constructors, function calls, let bindings (where the local variable x is only visible in e_1, e_2), disjunctions (e.g., to represent non-deterministic operations), and case expressions of the following form (we write $(f)\mathit{case}$ for either *fcase* or *case*):

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

where x is a variable, $c_1, \ldots, c_k$ are different constructors, and $e_1, \ldots, e_k$ are expressions. The *pattern variables* $\overline{x_{n_i}}$ are locally introduced and bind the corresponding variables of the subexpression e_i. The difference between *case* and *fcase* only shows up when the argument x evaluates (at run time) to a free variable: *case* suspends whereas *fcase* non-deterministically binds this variable to the pattern in a branch of the case expression.

Laziness of computations will show up in the description of the behavior of function calls and case expressions. In a function call, parameters are not evaluated but directly passed to the body of the function. In a case expression, only the outermost symbol of the case argument is required. Therefore, the case argument should be evaluated to *head normal form* [4] (i.e., a variable or an expression with a constructor at the outermost position). Consequently, our operational semantics will describe the evaluation of expressions only to head normal form. This is not a restriction since the evaluation to normal form or the solving of equations can be reduced to head normal form computations (see, e.g., [9]).

Extra variables are those variables in a rule which do not occur in the left-hand side. Such extra variables are intended to be instantiated by flexible case expressions. In the following, we assume that all extra variables x are explicitly introduced in flat programs by a direct circular let binding of the form "*let* $x = x$ *in* e". We call such variables which are bound to themselves *logical variables*.

In the remainder of this paper, we assume that computations always start from the distinguished function `main` which has no arguments.

3 A Run-Time Profiling Scheme

Traditionally, profiling tools attribute execution costs to the functions or procedures of the considered program. Following [2, 14], in this work we take a more flexible approach which allows us to associate a *cost center* with any expression of interest. This allows the programmer to choose an appropriate granularity for profiling, ranging from whole program phases to single subexpressions in a function. Nevertheless, our approach can easily be adapted to work with automatically instrumented cost centers; for instance, if one wants to use the traditional approach in which all functions are profiled, each function can be automatically annotated by introducing a cost center for the entire right-hand side. Cost centers are marked with the (built-in) function `scc` (for `s`et `c`ost `c`enter).

Intuitively speaking, given an expression "`scc`(cc, e)", the costs attributed to cost center cc are the entire costs of evaluating e as far as the enclosing context demands it, including the cost of

- evaluating any function called by the evaluation of the expression e,

but excluding the cost of

- evaluating the *free* variables of e (i.e., those variables which are not introduced by a let binding in e) and
- evaluating any `scc`-expressions within e or within any function which is called from e.

The following program contains two versions of a function to compute the length of a list (for readability, we show the non-normalized version of function `main`):

```
len(x) = fcase x of { []     → 0
                    ; (y:ys) → let z = 1, w = len(ys) in z + w }
len2s(x) = fcase x of { []     → 0
                      ; (y:ys) → fcase ys of
                                   { []     → 1
                                   ; (z:zs) → let w = 2,
                                                  v = len2s(zs)
                                              in w + v       } }
main = let list = scc("list",[1..5000])
       in scc("len",len(list)) + scc("len2s",len2s(list))
```

Table 1. Basic costs

Cost criteria	Symbol	Cost criteria	Symbol	Cost criteria	Symbol
Function unfolding	F	Allocating a heap object	H	Case evaluation	C
Variable update	U	Non-deterministic choice	N	Variable lookup	V
Entering an `scc`	E	Binding a logical variable	B		

Here, `main` computes twice the length of the list `[1..5000]`, which is a standard predefined way to define the list `[1,2,3,...,4999,5000]`. Each computation of the length uses a different function, `len` and `len2s`, respectively. In principle, `len2s` could seem more efficient than `len` because it performs half the number of function calls (indeed, `len2s` has been obtained by unfolding function `len`). This is difficult to check with traditional profilers because the overhead introduced to build the list hides the differences between `len` and `len2s`. For instance, the computed run times in the PAKCS environment [10] for Curry are 9980 ms and 9990 ms for `len([1..5000])` and `len2s([1..5000])`, respectively.[1]

From these figures, should one conclude that `len` and `len2s` are equally efficient? In order to answer this question, a profiler based on *cost centers* can be very useful. In particular, by including the three cost centers shown in the program above (function `main`), the costs of `len`, `len2s`, and the construction of the input list can be clearly distinguished. With our execution profiler which distributes the execution time to different cost centers (its implementation is discussed in Section 6.1), we have measured the following run times:

cost center	`main`	`list`	`len`	`len2s`
run times	17710	7668966	1110	790

Here, run times are expressed in a number of "ticks" (an artificial time unit provided by the SICStus Prolog profiling facilities [7]). Thanks to the use of cost centers, we can easily check that `len2s` is slightly more efficient than `len`. However, what is the reason for these different run times? We introduce *symbolic* costs—associated with the basic operations of the language semantics—so that a deeper analysis can be made. The considered kinds of costs are shown in Table 1. For the example above, our symbolic profiler returns the following cost attributions (only the most relevant costs for this example are shown):

	`main`	`list`	`len`	`len2s`
H	5000	61700	5100	5100
V	5100	280400	5100	5100
C	5100	280400	5100	5100
F	5300	168100	5100	2600

From this information, we observe that only function unfoldings (F) are halved, while the remaining costs are equal for both `len` and `len2s`. Therefore, we can

[1] The slow execution is due to the fact that experiments were performed with a version of the above program where a symbolic (Peano) representation of natural numbers is used.

conclude that, in this example, unfolding function `len` with no input data only improves cost F (which has a small impact on current compilers, as has been shown before).

4 Cost Semantics

In this section, we instrument a natural (big-step) semantics for functional logic languages (defined in [1]) with the computation of symbolic costs. Figure 2 shows the cost-augmented semantics. A *heap*, denoted by Γ, Δ, or Θ, is a partial mapping from variables to expressions (the *empty heap* is denoted by $[\,]$). The value associated to variable x in heap Γ is denoted by $\Gamma[x]$. $\Gamma[x \overset{cc}{\mapsto} e]$ denotes a heap with $\Gamma[x] = e$ and associated cost center cc; we use this notation either as a condition on a heap Γ or as a modification of Γ. A logical variable x is represented by a circular binding of the form $\Gamma[x] = x$. A *value* v is a constructor-rooted term $c(\overline{e_n})$ (i.e., a term whose outermost function symbol is a constructor symbol) or a logical variable (w.r.t. the associated heap). We use judgements of the form "$cc, \Gamma : e \;\Downarrow_\theta\; \Delta : v, cc_v$" which are interpreted as "in the context of heap Γ and cost center cc, the expression e evaluates to value v with associated cost θ, producing a new heap Δ and cost center cc_v".

In order to evaluate a variable which is bound to a constructor-rooted term in the heap, rule VarCons reduces the variable to this term. Here, cost V is attributed to the current cost center cc to account for the variable lookup (this attribution is denoted by $\{cc \leftarrow V\}$ and similarly for the other cost symbols).

Rule VarExp achieves the effect of *sharing*. If the variable to be evaluated is bound to some expression in the heap, then the expression is evaluated and the heap is updated with the computed value; finally, we return this value as the result. In addition to counting the cost θ of evaluating expression e, both V and U are attributed to cost centers cc and cc_v, respectively.

For the evaluation of a value, rule Val returns it without modifying the heap. No costs are attributed in this rule since actual implementations have no counterpart for this action.

Rule Fun corresponds to the unfolding of a function call. The result is obtained by reducing the right-hand side of the corresponding rule (we assume that the considered program P is a global parameter of the calculus). Cost F is attributed to the current cost center cc to account for the function unfolding.

Rule Let adds its associated binding to the heap and proceeds with the evaluation of its main argument. Note that we give the introduced variable a fresh name in order to avoid variable name clashes. In this case, cost H is added to the current cost center cc.

Rule Or non-deterministically reduces an *or* expression to either the first or the second argument. N is attributed to the current cost center to account for a non-deterministic step.

Rule Select corresponds to the evaluation of a case expression whose argument reduces to a constructor-rooted term. In this case, we select the appropriate branch and, then, proceed with the evaluation of the expression in this branch

(VarCons) $cc, \Gamma[x \overset{cc_c}{\longmapsto} c(\overline{x_n})] : x \ \Downarrow_{\{cc \leftarrow V\}} \ \Gamma[x \overset{cc_c}{\longmapsto} c(\overline{x_n})] : c(\overline{x_n}), cc_c$

(VarExp) $$\frac{cc_e, \Gamma : e \ \Downarrow_\theta \ \Delta : v, cc_v}{cc, \Gamma[x \overset{cc_e}{\longmapsto} e] : x \ \Downarrow_{\{cc \leftarrow V\}+\theta+\{cc_v \leftarrow U\}} \ \Delta[x \overset{cc_v}{\longmapsto} v] : v, cc_v}$$ (where e is not a value)

(Val) $cc, \Gamma : v \ \Downarrow_{\{\,\}} \ \Gamma : v, cc$ (where v is a value)

(Fun) $$\frac{cc, \Gamma : \rho(e) \ \Downarrow_\theta \ \Delta : v, cc_v}{cc, \Gamma : f(\overline{x_n}) \ \Downarrow_{\{cc \leftarrow F\}+\theta} \ \Delta : v, cc_v}$$ (where $f(\overline{y_n}) = e \in P$ and $\rho = \{\overline{y_n \mapsto x_n}\}$)

(Let) $$\frac{cc, \Gamma[y \overset{cc}{\longmapsto} \rho(e')] : \rho(e) \ \Downarrow_\theta \ \Delta : v, cc_v}{cc, \Gamma : let \ x = e' \ in \ e \ \Downarrow_{\{cc \leftarrow H\}+\theta} \ \Delta : v, cc_v}$$ (where $\rho = \{x \mapsto y\}$ and y is fresh)

(Or) $$\frac{cc, \Gamma : e_i \ \Downarrow_\theta \ \Delta : v, cc_v}{cc, \Gamma : e_1 \ or \ e_2 \ \Downarrow_{\{cc \leftarrow N\}+\theta} \ \Delta : v, cc_v}$$ (where $i \in \{1, 2\}$)

(Select) $$\frac{cc, \Gamma : x \ \Downarrow_{\theta_1} \ \Delta : c(\overline{y_n}), cc_c \qquad cc, \Delta : \rho(e_i) \ \Downarrow_{\theta_2} \ \Theta : v, cc_v}{cc, \Gamma : (f)case \ x \ of \ \{\overline{p_k \to e_k}\} \ \Downarrow_{\theta_1+\{cc \leftarrow C\}+\theta_2} \ \Theta : v, cc_v}$$ (where $p_i = c(\overline{x_n})$ and $\rho = \{\overline{x_n \mapsto y_n}\}$)

(Guess) $$\frac{cc, \Gamma : x \ \Downarrow_{\theta_1} \ \Delta : y, cc_y \qquad cc, \Delta[y \overset{cc}{\mapsto} \rho(p_i), \overline{y_n \overset{cc}{\mapsto} y_n}] : \rho(e_i) \ \Downarrow_{\theta_2} \ \Theta : v, cc_v}{cc, \Gamma : fcase \ x \ of \ \{\overline{p_k \to e_k}\} \ \Downarrow_{\theta_1+\{cc \leftarrow V, cc \leftarrow U, cc \leftarrow B, cc \leftarrow n*H\}+\theta_N+\theta_2} \ \Theta : v, cc_v}$$ (where $p_i = c(\overline{x_n})$, $\rho = \{\overline{x_n \mapsto y_n}\}$, $\overline{y_n}$ are fresh variables, and $\theta_N = \{cc \leftarrow N\}$ if $k > 1$ and $\theta_N = \{\,\}$ if $k = 1$)

(SCC) $$\frac{cc', \Gamma : e \ \Downarrow_\theta \ \Delta : v, cc_v}{cc, \Gamma : scc(cc', e) \ \Downarrow_{\theta+\{cc' \leftarrow E\}} \ \Delta : v, cc_v}$$

Fig. 2. Rules of the cost semantics

by applying the corresponding matching substitution. In addition to the costs of evaluating the case argument, θ_1, and the selected branch, θ_2, we add cost C to the current cost center cc to account for the pattern matching.

Rule Guess applies when the argument of a flexible case expression reduces to a logical variable. It binds this variable to one of the patterns and proceeds by evaluating the corresponding branch. If there is more than one branch, one of them is chosen non-deterministically. Renaming the pattern variables is necessary to avoid variable name clashes. We also update the heap with the (renamed) logical variables of the pattern. In addition to counting the costs of evaluating the case argument, θ_1, and the selected branch, θ_2, we attribute to the current cost center cc costs V (for determining that y is a logical variable), U (for updating the heap from $y \mapsto y$ to $y \mapsto \rho(p_i)$), B (for binding a logical variable), $n * H$ (for adding n new bindings into the heap) and, if there is more than one branch, N (for performing a non-deterministic step). Note that no cost C is attributed to cost center cc (indeed, cost B is alternative to cost C).

Finally, rule SCC evaluates an scc-expression by reducing the expression e in the context of the new cost center cc'. Accordingly, cost E is added to cost center cc'.

A proof of a judgement corresponds to a derivation sequence using the rules of Figure 2. Given a program P, the *initial configuration* has the form "$cc_{main}, [\,] : main$", where cc_{main} is a distinguished cost center. If the judgement

$$cc_{main}, [\,] : main \;\Downarrow_\theta\; \Gamma : v, cc_v$$

holds, we say that *main* evaluates to value v with associated cost θ. The computed *answer* can be extracted from the final heap Γ by a simple process of *dereferencing*.

Obviously, the cost semantics is a conservative extension of the original big-step semantics of [1], since the computation of cost information imposes no restriction on the application of the rules of the semantics.

5 Cost Instrumentation

As mentioned before, implementing an interpreter for the cost semantics of Figure 2 is impracticable. It would involve too much overhead to profile any realistic program. Thus, we introduce a transformation to instrument programs in order to compute the symbolic costs:

Definition 1 (cost transformation). *Given a program P, its cost instrumented version $cost(P)$ is obtained as follows: for each program rule*

$$f(x_1, \ldots, x_n) = e$$

$cost(P)$ includes, for each cost center cc in P, one rule of the form

$$f_{cc}(x_1, \ldots, x_n) = F_{cc}([\![e]\!]_{cc})$$

where $F_{cc}(e)$ is the identity function on e. Counting the calls to F_{cc} in the proof tree corresponds to the number of F's accumulated in cost center cc. Function $[\![\ ]\!]$ (shown in Figure 3) is used to instrument program expressions; similarly to F_{cc}, functions V_{cc}, U_{cc}, H_{cc}, N_{cc}, C_{cc}, B_{cc}, and E_{cc} are also defined as the identity function on their argument.

Observe that the transformed program contains as many variants of each function of the original program as different cost centers. Semantically, all these variants are equivalent; the only difference is that we obtain the costs of the computation by counting the calls to the different cost center identity functions (like F_{cc}).

Program instrumentation is mainly performed by function $[\![\ ]\!]_{cc}$, where cc denotes the current cost center. We informally explain how the transformation proceeds by a case distinction on the expression, e, in a call of the form $[\![e]\!]_{cc}$:

- If e is a variable, a call to function V_{cc} is added to attribute cost V to cost center cc.

$$
\begin{aligned}
[\![x]\!]_{cc} &= V_{cc}(x)\\
[\![c(x_1,\ldots,x_n)]\!]_{cc} &= c(cc,x_1,\ldots,x_n)\\
[\![f(x_1,\ldots,x_n)]\!]_{cc} &= f_{cc}(x_1,\ldots,x_n)\\
[\![let\ x = e'\ in\ e]\!]_{cc} &= H_{cc}\left(\begin{array}{ll} let\ x = x\ in\ [\![e]\!]_{cc}) & \text{if } e' = x\\ let\ x = [\![e']\!]_{cc}\ in\ [\![e]\!]_{cc} & \text{if } e' = c(\overline{y_n})\\ let\ x = update([\![e']\!]_{cc})\ in\ [\![e]\!]_{cc} & \text{otherwise}\end{array}\right)\\
[\![e_1\ or\ e_2]\!]_{cc} &= N_{cc}([\![e_1]\!]_{cc}\ or\ [\![e_2]\!]_{cc})\\
[\![case\ x\ of\ \{\overline{p_k \to e_k}\}]\!]_{cc} &= case\ [\![x]\!]_{cc}\ of\ \{\overline{p'_k \to C_{cc}([\![e_k]\!]_{cc})}\}\\
&\qquad \text{where } p'_i = c(cc',\overline{y_n}) \text{ for all } p_i = c(\overline{y_n})\\
[\![fcase\ x\ of\ \{\overline{p_k \to e_k}\}]\!]_{cc}&\\
= \ & if\ isVar(x)\\
& then\ V_{cc}(U_{cc}(B_{cc}(\theta_N(fcase\ [\![x]\!]_{cc}\ of\ \{\overline{p'_k \to |p_k| * H_{cc}([\![e_k]\!]_{cc})}\}))))\\
& else\ fcase\ [\![x]\!]_{cc}\ of\ \{\overline{p'_k \to C_{cc}([\![e_k]\!]_{cc})}\}\\
& \text{where } p'_i = c(cc,\overline{y_n}) \text{ for all } p_i = c(\overline{y_n}) \text{ and } \theta_N(e) = \begin{cases} e & \text{if } k = 1\\ N_{cc}(e) & \text{if } k > 1\end{cases}\\
[\![scc(cc',e)]\!]_{cc} &= E_{cc'}([\![e]\!]_{cc'})
\end{aligned}
$$

Here, $|p|$ denotes the arity of pattern p, i.e., $|p| = n$ if $p = c(\overline{x_n})$, and the auxiliary function *update* is used to attribute cost U to the cost center of the computed value:

$$update(x) = case\ x\ of\ \{\overline{c_k(cc_k,\overline{x_{n_k}}) \to U_{cc_k}(c_k(cc_k,\overline{x_{n_k}}))}\}$$

where $c_1,\ldots,c_k$ are the program constructors.

Fig. 3. Cost transformation $[\![\ ]\!]_{cc}$ for instrumenting expressions

- If $e = c(\overline{x_n})$ is a constructor-rooted term, we add a new argument to store the current cost center. This is necessary to attribute cost U to the appropriate cost center (i.e., to the cost center of the computed value, see Figure 2).
- A call to a function $f(\overline{x_n})$ is translated to a call to the function variant corresponding to cost center *cc*.
- If $e = (let\ x = e_1\ in\ e_2)$ is a let expression, a call to function H_{cc} is always added to attribute cost H to cost center *cc*. Additionally, if the binding is neither a logical variable nor a constructor-rooted term, the cost center cc_i, $1 \leq i \leq k$, of the computed value is determined (by means of an auxiliary function *update*, see Figure 3) and a call to U_{cc_i} is added to attribute cost U to that cost center.
- If $e = (e_1\ or\ e_2)$ is a disjunction, a call to N_{cc} is added to attribute N to cost center *cc*.
- If $e = case\ x\ of\ \{\overline{p_k \to e_k}\}$ is a rigid case expression, we recursively transform both the case argument and the expression of each branch, where a call to C_{cc} is added to attribute cost C to cost center *cc*. Observe that the cost center, cc', of the patterns is not used (it is only needed in the auxiliary function *update*).
- If $e = fcase\ x\ of\ \{\overline{p_k \to e_k}\}$ is a flexible case expression, a run-time test (function *isVar*) is needed to determine whether the argument evaluates to

a logical variable or not. This function can be found, e.g., in the library `Unsafe` of PAKCS. If it does not evaluate to a logical variable, we proceed as in the previous case. Otherwise, we add calls to functions V_{cc}, U_{cc}, B_{cc}, and N_{cc} (if $k > 1$). Also, in each case branch, calls to H_{cc} are added to attribute the size of the pattern to cost center cc. Here, we use $n * H_{cc}$ as a shorthand for writing n nested calls to H_{cc} (in particular, $0 * H_{cc}$ means that no call to H_{cc} is written).
- Finally, if $e = scc(cc', e')$ is an scc-expression, a call to function $E_{cc'}$ is added. More importantly, we update the current cost center to cc' in the recursive transformation of e'.

Derivations with the standard semantics (i.e., without cost centers) are denoted by $([\,] : main \Downarrow \Gamma_c : v)$. Given a heap Γ, we denote by Γ_c the set of bindings $x \mapsto e'$ such that $x \overset{cc}{\longmapsto} e$ belongs to Γ, where $e' = e$ if e is a logical variable, $e' = [\![e]\!]_{cc}$ if $e = c(\overline{x_n})$, or $e' = update([\![e]\!]_{cc})$ otherwise. Also, in order to make explicit the output of the instrumented program with the standard semantics, we write $([\,] : main \Downarrow^{\theta} \Gamma_c : v)$, where θ records the set of calls to cost functions (e.g., H_{cc}, F_{cc}).

The correctness of our program instrumentation is stated as follows (the proof can be found in [5]):

Theorem 1 (correctness). *Let P be a program and $cost(P)$ be its cost instrumented version. Then,*

$$(cc_{main}, [\,] : main \Downarrow_{\theta} \Gamma : v, cc) \text{ in } P \text{ iff } ([\,] : main_{cc_{main}} \Downarrow^{\theta} \Gamma_c : v') \text{ in } cost(P)$$

where $v = v'$ (if they are variables) or $v = c(\overline{x_n})$ and $v' = c(cc, \overline{x_n})$.

As an alternative to the transformation presented in this section, we could also instrument programs by *partial evaluation* [11], i.e., the partial evaluation of the cost semantics of Section 4 w.r.t. a source program P should return an instrumented program which is equivalent to $cost(P)$. However, this approach requires both an implementation of the instrumented semantics as well as an optimal partial evaluator for the considered language (in order to obtain a reasonable instrumented program, rather than a slight specialization of the cost semantics). Thus, we preferred to introduce a direct transformation.

Now, we illustrate our cost transformation by means of a simple example. Consider, for instance, the following definition for function `len`:

```
len(x) = fcase x of { Nil         → Z
                    ; Cons(y,ys) → let w = scc("b",len(ys))
                                   in S(w) }
```

Then, the corresponding instrumented definition for the cost center `"a"` is the following:

```
len_a(x) =
  F_a(if isVar(x)
         then V_a(U_a(B_a(N_a(fcase V_a(x) of
```

```
                { Nil(cc) → Z(a)
                ; Cons(cc,y,ys) → H_a(H_a(H_a(let w = update(len_b(ys))
                                                in S(a,w))))
                } ))))
             else fcase V_a(x) of
                { Nil(cc) → Z(a)
                ; Cons(cc,y,ys) → C_a(H_a(let w = update(len_b(ys))
                                             in S(a,w)))
                }
      )
```

where the auxiliary function *update* is defined as follows:

```
update(x) = case x of { Z(cc)    → U_cc(Z(cc))
                      ; S(cc,x) → U_cc(S(cc,x)) }
```

6 Implementation

The main purpose of profiling programs is to increase run-time efficiency. However, in practice, it is important to obtain symbolic profiling information as well as measuring run times. As discussed before, we want to provide cost centers for both kinds of profiling in order to be able to analyze arbitrary sub-computations independently of the defined functions. For the formal introduction of costs and correctness proofs, symbolic costs are the appropriate means. Therefore, we introduced a program transformation dealing with symbolic costs. However, the presented program transformation can easily be extended for measuring run times and distribute them through cost centers. In this section, we first present our approach to measure run times and function calls (Sect. 6.1) and, then, describe the extensions to obtain symbolic profiling (Sect. 6.2).

6.1 Measuring Run Times

When trying to measure actual run times, the crucial point is to alter the run time behavior of the examined program *as little as possible*. If the program instrumented for profiling runs 50% slower or worse, one profiles the process of profiling rather than the program execution. Because of this, measuring actual run times is a matter of low-level programming and, thus, highly depending on the actual language implementation.

Our approach is specific to the Curry implementation PAKCS [10]. In this programming environment, Curry programs are compiled by transforming flat programs (cf. Section 2) to SICStus Prolog (see [3] for details about this transformation). Note, however, that in contrast to Section 2 the programs are not necessarily normalized. In order to provide low-level profiling for PAKCS, we instrument the program with the profiling mechanisms offered by SICStus Prolog. Fortunately, SICStus Prolog features low-level profiling instruments which

create an overhead of approximately 22%. The Prolog tools provide precise measuring of the number of predicate and clause calls. For measuring run time, a number of synthetic units is given which is computed according to [7].

The main challenge was to introduce the cost centers into Prolog profiling. Luckily, we found a way to do this *without* further slowing down the execution of the program being profiled. The only overhead we introduce is code duplication, since we introduce a different version of each function for each cost center, as in the program transformation described above. Thus, for the program

```
main = SCC "len" (length (SCC "list" (enumFromTo 1 10)))
```

function `main` does not call a function `length` but a variant with the name "`length{len}`" and also a function named "`enumFromTo{list}`". Gathering all run times for functions with the attachment $\{cc\}$, one gets the run time belonging to that cost center. An obvious optimization is to eliminate unreachable functions like `length{list}` in the example.

6.2 Extension for Symbolic Profiling

Our approach to symbolic profiling exactly represents the idea described in Section 5 above. For each cost, we introduce a new function, e.g., `var_lookup` for cost V. There are variations of these functions for the different cost centers, e.g., `var_lookup{list}` like in Section 6.1. After the execution of the transformed program, we simply count each call to `var_lookup{list}` to get the sum of costs V attributed to the cost center `list`.

The advantage of this method is its simplicity. The demands to use our transformation for profiling with any implementation of Curry are not very high. The run-time system must only be able to count the number of calls to a certain function which is easy to implement. The disadvantage is the considerable (but still reasonable) slowdown as we are not only introducing new functions but also new function *calls*. Nevertheless, this overhead does not influence the computation of symbolic costs.

The overhead introduced by the additional function calls is also the reason why our profiler generates different programs for run-time profiling and symbolic profiling. Since the program transformed for symbolic profiling is more than a magnitude slower than the original program, measuring run times in the program transformed for symbolic profiling would lead to results that are not strictly related to the performance of the original program.

It is worthwhile to note that, although the program transformation of Fig. 3 is equivalent to the cost semantics of Fig. 2 for *particular computations* (as stated in Theorem 1), there is one important difference:

> *While the cost semantics is don't-care non-deterministic, the instrumented programs accumulate all costs according to the search strategy.*

For instance, the cost for a failing derivation is also accumulated in the cost of the results computed afterwards. Furthermore, completely failing computations

also have an associated cost while no proof tree (and thus no costs) can be constructed in the big-step semantics. From a practical point of view, this is an advantage of the program transformation over the cost semantics, since the cost of failing derivations is relevant in the presence of non-deterministic functions.

7 Using the Profiler

In this section we present how our profiler can be used to improve the runtime of Curry programs by means of a larger example. We want to implement an algorithm solving the following problem:

> An *alphabet* is given by the algebraic datatype
>
> ```
> data Letter = A | B | ... | Y | Z
> type Word = [Letter]
> type Alphabet = [Letter]
> ```
>
> *Words* are defined as sequences of letters and (sub-)alphabets as sets (implemented as lists) of letters. Define a function `sameUsedAlphabet` that takes two words as input and, if both words use the same sub-alphabet, yields this sub-alphabet as its result.

The basic idea of an algorithm for solving this problem could be the following:

- Extract the sub-alphabets used by each string by means of removing double occurrences of letters (`rmDups`).
- Check whether the two sub-alphabets are permutations of each other (`isPerm`), otherwise fail.
- Return the sub-alphabet of the first word.

A possible implementation of these functions in Curry could be the following:

```
rmDups [] = []
rmDups (x:xs) = if elem x (rmDups xs) then rmDups xs
                                      else x:rmDups xs
```

Thus, an element is kept in the list if it is not an element of the remaining list.

```
isPerm []     [] = success
isPerm (x:xs) ys | eqWord (zs++[x]++us) ys = isPerm xs (zs++us)
    where zs,us free

eqWord [] [] = success
eqWord (x:xs) (y:ys) | eqLetter x y = eqWord xs ys

eqLetter A A = success
...
eqLetter Z Z = success
```

In the implementation of `isPerm`, we exploit the logical features of Curry. The function `isPerm` is applied to two lists. We successively delete the elements of the first list from the second list until both lists are empty. For deleting an element from the second list, we use the append function `(++)` as a relation by applying it to logical variables. When the expression `(zs++[x]++us)` is compared with `ys` by the function `eqWord`, the logical variable `zs` is bound to the part of `ys` in front of `x` and `us` to the part behind `x`. The list not containing `x` is `zs++us` which is recursively compared with `xs`.

The functions `eqWord` and `eqLetter` define unification for letters and words. In Curry this unification is generalized to arbitrary data types by means of strict equality `(=:=)` [6]. Our implementation also provides profiling information for this extension. For simplicity, we do not present the technically expensive details of this extension in this paper and define specific functions for the unification of words and letters in this example.

Finally, we combine all these functions to solve the problem by means of the function `sameUsedAlphabet`:

```
sameUsedAlphabet :: [Letter] -> [Letter] -> [Letter]
sameUsedAlphabet str1 str2
     | isPerm (rmDups str1) (rmDups str2) = rmDups str1
```

Testing `sameUsedAlphabet` for short words is reasonably efficient. Unfortunately, our algorithm does not scale well for longer words. For instance, the application of `sameUsedAlphabet` to a word containing all letters of the alphabet does not terminate within one hour.

To find the source of this inefficiency, we use our profiler. We add two cost centers `"perm"` and `"rmDups"` as follows:

```
rmDups xs = SCC "rmDups" (rmDups' xs)
rmDups' [] = []
rmDups' (x:xs) = if elem x (rmDups' xs) then rmDups' xs
                                        else x:rmDups' xs

sameUsedAlphabet :: Word -> Word -> Alphabet
sameUsedAlphabet str1 str2
 | SCC "perm" (isPerm (rmDups str1) (rmDups str2)) = rmDups str1
```

Profiling some applications of `sameUsedAlphabet`, we obtain the following measurements for function unfoldings and heap allocations (we do not present the other kinds of costs since they do not provide more information here):

`sameUsedAlphabet` applied to	`"perm"`		`"rmDups"`	
	F	H	F	H
[A,B] [B,A]	39	18	148	159
[A,B,C,D,E,F] [F,E,D,C,B,A]	237	166	3881	3924
[A,B,C,D,E,F] [F,E,D,C,B,A,F,E,D,C,B,A]	237	166	127643	128316

An analysis of this profiling results yields the following conclusions:

– The costs for `rmDups` are much higher than the costs for `perm`.
– The costs for `perm` only depend on the sub-alphabet of the words, not on the size of the word.
– The costs for `rmDups` grow exponentially in the size of the word.

Without using the profiler, we might have blamed the inefficiency to the logical part of the program (i.e., the use of the potentially inefficient constraint `isPerm`). However, thanks to the information gathered by the profiler, we know that we should focus on the code of `rmDups` to optimize our program. In fact, during the recursion of `rmDups` we compute the result of `rmDups xs` twice, which yields exponential runtime. By simply introducing a `let` binding for this result, we obtain a much more efficient version of `rmDups`:

```
rmDups [] = []
rmDups (x:xs) = let ys = rmDups xs in
                  if elem x ys then ys else x:ys
```

8 Related Work and Conclusions

The approaches closest to our work are [14] and [2]. On the one hand, [14] presents a formal specification of the attribution of execution costs to cost centers by means of an appropriate cost-augmented semantics in the context of lazy functional programs. A significant difference from our work is that our flat representation of programs provides for logical features (like non-determinism) and that we also present a formal transformation to instrument source programs. On the other hand, [2] introduces a symbolic profiling scheme for functional logic languages. However, the approach of [2] does not consider *sharing* (an essential component of lazy languages) and, thus, it is not an appropriate basis for the development of profiling tools for current implementations of lazy functional logic languages. Furthermore, we introduced a program transformation that allows us to compute symbolic costs with a reasonable overhead. Finally, in the context of the PAKCS environment for Curry, we showed how actual run times can also be computed by reusing the SICStus Prolog profiler.

References

1. E. Albert, M. Hanus, F. Huch, J. Oliver, and G. Vidal. Operational Semantics for Declarative Multi-Paradigm Languages. *Journal of Symbolic Computation*, 2005. To appear.
2. E. Albert and G. Vidal. Symbolic Profiling of Multi-Paradigm Declarative Languages. In *Proc. of Int'l Workshop on Logic-based Program Synthesis and Transformation (LOPSTR'01)*, pages 148–167. Springer LNCS 2372, 2002.
3. S. Antoy and M. Hanus. Compiling Multi-Paradigm Declarative Programs into Prolog. In *Proc. of the Int'l Workshop on Frontiers of Combining Systems (FroCoS'2000)*, pages 171–185. Springer LNCS 1794, 2000.
4. H.P. Barendregt. *The Lambda Calculus—Its Syntax and Semantics*. Elsevier, 1984.

5. B. Braßel, M. Hanus, F. Huch, J. Silva, and G. Vidal. Run-Time Profiling of Functional Logic Programs. Technical report, DSIC, Technical University of Valencia, 2005. Available at: `http://www.dsic.upv.es/users/elp/german/papers.html`.
6. M. Hanus (ed.). Curry: An Integrated Functional Logic Language. Available at: `http://www.informatik.uni-kiel.de/~curry/`.
7. M. Gorlick and C. Kesselman. Timing Prolog Programs without Clock. In *Proc. of the 4th Symposium on Logic Programming (SLP'87)*, pages 426–434, 1987.
8. M. Hanus. A Unified Computation Model for Functional and Logic Programming. In *Proc. of the 24th ACM Symp. on Principles of Programming Languages (POPL'97)*, pages 80–93. ACM, New York, 1997.
9. M. Hanus and C. Prehofer. Higher-Order Narrowing with Definitional Trees. *Journal of Functional Programming*, 9(1):33–75, 1999.
10. M. Hanus (ed.), S. Antoy, M. Engelke, K. Höppner, J. Koj, P. Niederau, R. Sadre, and F. Steiner. PAKCS 1.6.0: The Portland Aachen Kiel Curry System—User Manual. Technical report, University of Kiel, Germany, 2004.
11. N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, Englewood Cliffs, NJ, 1993.
12. J. Launchbury. A Natural Semantics for Lazy Evaluation. In *Proc. of the ACM Symp. on Principles of Programming Languages (POPL'93)*, pages 144–154. ACM Press, 1993.
13. F. López-Fraguas and J. Sánchez-Hernández. TOY: A Multiparadigm Declarative System. In *Proc. of the 10th Int'l Conf. on Rewriting Techniques and Applications (RTA'99)*, pages 244–247. Springer LNCS 1631, 1999.
14. 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.