Time Equations for Lazy Functional (Logic) Languages*

Elvira Albert[1], Josep Silva[2], and Germán Vidal[2]

[1] DSIP, Universidad Complutense de Madrid
Email: elvira@sip.ucm.es

[2] DSIC, Technical University of Valencia
Email: {jsilva,gvidal}@dsic.upv.es

Abstract. There are very few approaches to measure the execution costs of *lazy* functional (logic) programs. The use of a lazy execution mechanism implies that the complexity of an evaluation depends on its *context*, i.e., the evaluation of the same expression may have different costs depending on the degree of evaluation required by the different contexts where it appears. In this paper, we present a novel approach to complexity analysis of functional (logic) programs. We focus on the construction of equations which compute the time-complexity of expressions. In contrast to previous approaches, it is simple, precise—i.e., it computes exact costs rather than upper/lower bounds—, and fully automatic.

1 Introduction

There are very few approaches to formally reason about program execution costs in the field of lazy functional (logic) languages. Most of these approaches analyze the cost by first constructing (recursive) equations which describe the time-complexity of a given program and then producing a simpler cost function (ideally as a *closed form*) by some (semi-)automatic manipulation of these equations. In this paper, we concentrate on the *first* part of this process: the (automatic) construction of equations which compute the time-complexity of a given lazy functional (logic) program.

Laziness introduces a considerable difficulty in the time-complexity analysis of functional (logic) programs. In eager (call-by-value) languages, the cost of nested applications, e.g., $f(g(v))$, can be expressed as the sum of the costs of evaluating $g(v)$ and $f(v')$, where v' is the result returned by the call $g(v)$. In a lazy (call-by-name) language, $g(v)$ is only evaluated as much as needed by the outermost function f (in the extreme, it could be no evaluated at all). Therefore, one cannot determine (statically) the cost associated to the evaluation of the arguments of function calls. There exist several methods which simplify this problem by transforming the original program into an equivalent one under a call-by-value semantics and, then, by analyzing the transformed program (e.g.,

* This work has been partially supported by CICYT TIC 2001-2705-C03-01, by the Generalitat Valenciana CTIDIA/2002/205 and by the MCYT HA2001-0059.

[13]). The translation, however, makes the program significantly more complex and the number of evaluation steps is not usually preserved through the transformation. A different approach is taken by [14, 17], where the generated time-equations give upper (resp. lower) bounds by using a strictness (resp. neededness) analysis. In particular, [17] may produce equations which are non-terminating even if the original program terminates; in contrast, [14] guarantees that the time-equations terminate at least as often as the original program. As an alternative approach, there are techniques (e.g., [6]) which produce time equations which are *precise*, i.e., they give an *exact* time analysis. In this case, the drawback is that the time equations cannot be solved mechanically.

In this work, we introduce a novel program transformation which produces (automatically) time equations which are *precise*—i.e., give an exact time—and, moreover, they can be executed in the same language of the source program—i.e., we define a source-to-source transformation. To the best of our knowledge, this is the first approach which achieves both goals in the context of a lazy language.

To be more precise, our aim is to transform a program so that it computes not only values but their associated time-complexities. More formally, given a lazy (call-by-name) semantics Sem and its cost-augmented version Sem_{cost}, we want to define a program transformation $trans(P) = P_{cost}$ such that the execution of program P with the cost-augmented semantics gives the same result as the execution of the transformed program P_{cost} with the standard semantics, i.e., $Sem_{cost}(P) = Sem(P_{cost})$. Let us note that the considered problem is decidable. First, it is not difficult to construct a cost-augmented semantics which precisely computes the cost of a program's execution (see, e.g., Section 2). Then the Futamura projections [7] ensure the existence of the desired transformed program P_{cost}: the *partial evaluation* [10] of Sem_{cost} w.r.t. P gives a program P' such that P' produces the same outputs as $Sem_{cost}(P)$, i.e.,

$$Sem(P, input) = output$$
$$Sem_{cost}(P, input) = (output, cost)$$
$$mix(Sem_{cost}, P) = P' \text{ with } P'(input) = (output, cost)$$

where mix denotes a partial evaluator. Therefore, P_{cost} (equivalently, P' in the above equations) exists and can be effectively constructed. However, by using a partial evaluator for the considered language (e.g., [3, 4]), P_{cost} will be probably a cost-augmented interpreter slightly extended to only consider the rules of a particular program P. Therefore, similarly to the cost-augmented interpreter, the program P_{cost} generated in this way will still use a sort of "ground representation" to evaluate expressions. This makes the complexity analysis of the program P_{cost} as difficult as the analysis of the original program (or even harder). Furthermore, its execution will be almost as slow as running the cost-augmented interpreter with the original program P (which makes it infeasible for profiling).

Our main concern in this work is to provide a more practical transformation. In particular, our new approach produces simple equations, it gives exact time-complexities, and it is fully automatic. To the best of our knowledge, this is the first transformation fulfilling such properties. For the sake of generality, our developments are formalized in the context of a lazy functional *logic* language.

Program:	$P ::= D_1 \ldots D_m$	
Function definition:	$D ::= f(x_1, \ldots, x_n) = e$	
Expression:	$e ::= x$	(variable)
	$\mid c(x_1, \ldots, x_n)$	(constructor call)
	$\mid f(x_1, \ldots, x_n)$	(function call)
	$\mid$ *let* $x_1 = e_1, \ldots, x_n = e_n$ *in* e	(let binding)
	$\mid$ *case* x *of* $\{p_1 \to e_1; \ldots; p_n \to e_n\}$	(rigid case)
	$\mid$ *fcase* x *of* $\{p_1 \to e_1; \ldots; p_n \to e_n\}$	(flexible case)
Pattern:	$p ::= c(x_1, \ldots, x_n)$	

Fig. 1. Syntax of lazy functional logic programs

In particular, the use of *logical variables* may be useful to analyze the complexity of the time-equations (as we will discuss in Section 4.1). Nevertheless, our ideas can be directly applied to a pure (first-order) lazy functional language.

The paper is organized as follows. Section 2 introduces the syntax and cost-augmented semantics of the considered lazy functional logic language. In Sect. 3, we present our program transformation and state its correctness. The usefulness of the transformation is illustrated in Sect. 4 and several applications are outlined. Finally, Sect. 5 concludes and gives some directions for future work.

2 Language Syntax and Cost Semantics

In this section, we introduce the syntax of our lazy functional logic language as well as a precise characterization of its cost semantics. A survey of functional logic languages can be found in [8].

Functional Logic Programs. The syntax for programs is shown in Figure 1. A program P consists of a sequence of function definitions D such that the left-hand side has pairwise different variable arguments. The right-hand side contains variables $\mathcal{X} = \{x, y, z, \ldots\}$, data constructors (e.g., a, b, c,...), user-defined functions (e.g., f, g, h,...), case expressions, and let bindings where the local variables $x_1, \ldots, x_n$ are only visible in $e_1, \ldots, e_n, e$. Note that we only consider function and constructor calls whose arguments are all variables (not necessarily different). This is not a restriction in practice since several simple *normalization* algorithms exist (see, e.g., [2]). Basically, these algorithms proceed by denoting all non-variable arguments in function or constructor calls by means of let bindings. In this work, we only consider programs with no *extra-variables*, i.e., all the free variables in the right-hand side of a program rule must appear in the left-hand side. A case expression has the following form:[3]

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

[3] We write $\overline{o_n}$ for the *sequence* $o_1, \ldots, o_n$ and $(f)case$ for either *fcase* or *case*.

(VarCons) $\Gamma[x \mapsto t] : x \;\Downarrow_0\; \Gamma[x \mapsto t] : t$ where t is constructor-rooted

(VarExp) $$\frac{\Gamma[x \mapsto e] : e \;\Downarrow_k\; \Delta : v}{\Gamma[x \mapsto e] : x \;\Downarrow_k\; \Delta[x \mapsto e] : v}$$ where e is not constructor-rooted and $e \neq x$

(Val) $\Gamma : v \;\Downarrow_0\; \Gamma : v$ where v is constructor-rooted or a variable with $\Gamma[v] = v$

(Fun) $$\frac{\Gamma : \rho(e) \;\Downarrow_k\; \Delta : v}{\Gamma : f(\overline{x_n}) \;\Downarrow_{k+1}\; \Delta : v}$$ where $f(\overline{y_n}) = e \in P$ and $\rho = \{\overline{y_n \mapsto x_n}\}$

(Let) $$\frac{\Gamma[\overline{y_k \mapsto \rho(e_k)}] : \rho(e) \;\Downarrow_k\; \Delta : v}{\Gamma : let\ \{\overline{x_k = e_k}\}\ in\ e \;\Downarrow_k\; \Delta : v}$$ where $\rho = \{\overline{x_k \mapsto y_k}\}$ and $\overline{y_k}$ are fresh variables

(Select) $$\frac{\Gamma : e \;\Downarrow_{k_1}\; \Delta : c(\overline{y_n}) \qquad \Delta : \rho(e_i) \;\Downarrow_{k_2}\; \Theta : v}{\Gamma : (f)case\ e\ of\ \{\overline{p_k \to e_k}\} \;\Downarrow_{k_1+k_2}\; \Theta : v}$$ where $p_i = c(\overline{x_n})$ and $\rho = \{\overline{x_n \mapsto y_n}\}$

(Guess) $$\frac{\Gamma : e \;\Downarrow_{k_1}\; \Delta : x \qquad \Delta[x \mapsto \rho(p_i), \overline{y_n \mapsto y_n}] : \rho(e_i) \;\Downarrow_{k_2}\; \Theta : v}{\Gamma : fcase\ e\ of\ \{\overline{p_k \to e_k}\} \;\Downarrow_{k_1+k_2}\; \Theta : v}$$ where $p_i = c(\overline{x_n})$, $\rho = \{\overline{x_n \mapsto y_n}\}$, and $\overline{y_n}$ are fresh variables

Fig. 2. Step-Counting Semantics for Lazy Functional Logic Programs

where e is an expression, $c_1, \ldots, c_k$ 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 e is a free variable: *case* suspends whereas *fcase* non-deterministically binds this variable to the pattern in a branch of the case expression.

Programs which follow the syntax of Figure 1 only differs from typical lazy (first-order) functional programs in the use of *flexible* case expressions. Therefore, our developments apply straightforwardly to pure lazy functional programs.

A Step-Counting Semantics. We base our developments on the big-step semantics of [2] for (first-order) lazy functional logic programs. The only difference is that our cost-augmented semantics does not model *sharing*, since then the generated time-equations would be much harder to analyze (see Section 4.2). In the following, we present an extension of the big-step semantics of [2] in order to count the number of *function unfoldings*. The step-counting (state transition) semantics for lazy functional logic programs is defined in Figure 2. Our rules obey the following naming conventions:

$$\Gamma, \Delta, \Theta \in Heap \;=\; Var \to Exp \qquad v \in Value \;::=\; x \mid c(\overline{e_n})$$

A *heap* 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 $\Gamma[x]$. $\Gamma[x \mapsto e]$ denotes a heap with $\Gamma[x] = e$, i.e., we use this notation either as a condition on a heap Γ or as a modification of Γ. In a heap Γ, a logical variable x is represented by a circular binding of the form $\Gamma[x] = x$. A *value* is a term in so called *head normal form*, i.e., a constructor-rooted term or a logical variable (w.r.t. the associated heap).

We use judgments of the form "$\Gamma : e \ \Downarrow_k \ \Delta : v$" which are interpreted as "the expression e in the context of the heap Γ evaluates to the value v with the (possibly modified) heap Δ by unfolding k function calls". We briefly explain the rules of our semantics:

(VarCons) In order to evaluate a variable which is bound to a constructor-rooted term in the heap, we simply reduce the variable to this term. The heap remains unchanged and the associated cost is trivially zero.

(VarExp) If the variable to be evaluated is bound to some expression in the heap, then this expression is evaluated and, then, this value is returned as the result. The cost is not affected by the application of this rule.
It is important to notice that this rule is different from the one presented in [2]. In particular, in the original big-step semantics, it is defined as follows:

$$\frac{\Gamma[x \mapsto e] : e \ \Downarrow_k \ \Delta : v}{\Gamma[x \mapsto e] : x \ \Downarrow_k \ \Delta[x \mapsto v] : v}$$

which means that the heap is updated with the computed value v for e. This is essential to correctly model sharing (trivially, our modification does not affect to the computed results, only to the number of function unfoldings). The generation of time-equations that correctly model sharing is a difficult topic which is out of the scope of this paper (see Section 4.2).

(Val) For the evaluation of a value, we return it without modifying the heap, with associated cost zero.

(Fun) This rule 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. Trivially, if the evaluation of the unfolded right-hand side requires k function unfoldings, the original function call will require $k + 1$ function calls.

(Let) In order to reduce a let construct, we add the bindings to the heap and proceed with the evaluation of the main argument of *let*. Note that we rename the variables introduced by the let construct with fresh names in order to avoid variable name clashes. The number of function unfoldings is not affected by the application of this rule.

(Select) This rule 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 by applying the corresponding matching substitution. The number of function unfoldings is obtained by adding the number of function unfoldings required to evaluate the case argument with those required to evaluate the expression in the selected branch.

(Guess) This rule considers the evaluation of a flexible case expression whose argument reduces to a logical variable. It non-deterministically binds this variable to one of the patterns and proceeds with the evaluation of the corresponding branch. Renaming of pattern variables is also necessary to avoid variable name clashes. Additionally, we update the heap with the (renamed) logical variables of the pattern. Similarly to the previous rule, the number of

function unfoldings is the addition of the function unfoldings in the evaluation of the case argument and the (non-deterministically) selected branch.

A proof of a judgment corresponds to a derivation sequence using the rules of Figure 2. Given a program P and an expression e (to be evaluated), the *initial configuration* has the form "$[\,] : e$". If the judgment "$[\,] : e \;\Downarrow_k\; \Gamma : v$" holds, then the computed *answer* can be extracted from the final heap Γ by a simple process of *dereferencing* in order to obtain the values associated to the logical variables of the initial expression e.

In the following, we denote by a judgment of the form $\Gamma : e \;\Downarrow\; \Delta : v$ a derivation with the standard semantics (without cost), i.e., the semantics of Figure 2 by ignoring the cost annotations.

3 The Program Transformation

In this section, we present our program transformation to compute the time complexity of lazy functional logic programs. First, let us informally describe the approach of [17] in order to motivate our work and its advantages (the approach of [14] is basically equivalent except for the use of a neededness analysis rather than a strictness analysis). In this method, for each function definition $f(x_1, \ldots, x_n) = e$, a new rule of the form

$$cf(x_1, \ldots, x_n, \alpha) \;=\; \alpha \;\hookrightarrow\; 1 + \mathcal{T}[\![e]\!]\alpha$$

is produced, where cf is a *time function* and α denotes an evaluation *context* which indicates whether the evaluation of a given expression is required. In this way, transformed functions are parameterized by a given context. The right-hand side is "guarded" by the context so that the corresponding cost is not added when its evaluation is not required. A key element of this approach is the *propagation* of contexts through expressions. For instance, the definition of function $\mathcal{T}$ for function calls is as follows:

$$\mathcal{T}[\![f(e_1, \ldots, e_n)]\!]\alpha = \mathcal{T}[\![e_1]\!](f^{\sharp 1}\alpha) + \ldots + \mathcal{T}[\![e_n]\!](f^{\sharp n}\alpha) + cf(e_1, \ldots, e_n, \alpha)$$

The basic idea is that the cost of evaluating a function application can be obtained from the cost of evaluating the *strict* arguments of the function plus the cost of the outermost function call. This information is provided by a strictness analysis. Recall that a function is *strict* in some argument i iff $f(\ldots, \bot_i, \ldots) = \bot$, where $\bot$ denotes an undefined expression (i.e., the result of function f cannot be computed when the argument i cannot be evaluated to a value).

Functions $f^{\sharp i}$ are used to propagate strictness information through function arguments. To be more precise, they are useful to determine whether argument i of function f is strict in the context α. Let us illustrate this approach with an example. Consider the following function definitions:[4]

```
head xs = case xs of {(y : ys) → y}
foo xs  = let ys = head xs in head ys
```

[4] Although we consider a first-order language, we use a curried notation in examples, as it is common practice in functional programming.

$$
\begin{array}{rl}
\mathcal{C}[\![x]\!] & = x \\
\mathcal{C}[\![c(x_1,\ldots,x_n)]\!] & = (c(x_1,\ldots,x_n),0) \\
\mathcal{C}[\![f(x_1,\ldots,x_n)]\!] & = f_c(x_1,\ldots,x_n) \\
\mathcal{C}[\![(f)case\ x\ of\ \{\overline{p_n \to e_n}\}]\!] & = (f)case\ x\ of\ \{\overline{(p_n,k_n) \to \mathcal{C}[\![e_n]\!] \mathbin{\underline{+}} k_n}\} \\
\mathcal{C}[\![let\ \overline{y_n = e_n}\ in\ e]\!] & = let\ \overline{y_n = \mathcal{C}[\![e_n]\!]}\ in\ \mathcal{C}[\![e]\!]
\end{array}
$$

Fig. 3. Cost Function $\mathcal{C}$

According to [17], these functions are transformed into

$$
\begin{array}{l}
\texttt{chead xs}\ \alpha = \alpha \hookrightarrow \texttt{1} \\
\texttt{cfoo xs}\ \alpha\ \ = \alpha \hookrightarrow \texttt{1} + \texttt{chead xs}\ (\texttt{head}^{\sharp 1}\alpha) \\
\qquad\qquad\qquad\quad + \texttt{let ys} = \texttt{head xs in chead}\ (\texttt{ys}\ \alpha)
\end{array}
$$

If the strictness analysis has a good precision, then whenever this function is called within a strict context, $(\texttt{head}^{\sharp 1}\alpha)$ should indicate that the head normal form of the argument of `head` is also needed.

Unlike the previous transformation, we do not use the information of a strictness (or neededness) analysis. A novel idea of our method is that we *delay* the computation of the cost of a given function up to the point where its evaluation is actually required. This contrasts to [14, 17] where the costs of the arguments of a function call are computed *a priori*, hence the need of the strictness information. The following definition formalizes our program transformation:

Definition 1 (time-equations). *Let P be a program. For each program rule of the form $f(x_1,\ldots,x_n) = e \in P$, we produce a transformed rule:*

$$f_c(x_1,\ldots,x_n) = \mathcal{C}[\![e]\!] \mathbin{\underline{+}} 1$$

where the definition of the cost function $\mathcal{C}$ is shown in Figure 3. Additionally, we add the following function "$\underline{+}$" to the transformed program:

$$
\begin{array}{rll}
(c,\ k_1) \mathbin{\underline{+}} k_2 = (c,\ k_1 + k_2) & & \textit{for all constructor } c/0 \textit{ in } P \\
(c(\overline{x_n}),\ k_1) \mathbin{\underline{+}} k_2 = (c(\overline{x_n}),\ k_1 + k_2) & & \textit{for all constructor } c/n \textit{ in } P
\end{array}
$$

Basically, for each constructor term $c(\overline{x_n})$ computed in the original program, a pair of the form $(c(\overline{x'_n}), k)$ is computed in the transformed program, where k is the number of function unfoldings which are needed to produce the constructor "c". For instance, a constructor term—a list—like $(\texttt{x} : \texttt{y} : [])$ will have the form $(\texttt{x} : (\texttt{y} : ([], \texttt{k}_3), \texttt{k}_2), \texttt{k}_1)$ in the transformed program, where $\texttt{k}_1$ is the cost of producing the outermost constructor ":", $\texttt{k}_2$ the cost of producing the innermost constructor ":" and $\texttt{k}_3$ the cost of producing the empty list. The auxiliary function "$\underline{+}$" is introduced to correctly increment the current cost.

The definition of the cost function $\mathcal{C}$ distinguishes the following cases depending on the structure of the expression in the right-hand side of the program rule, as it is depicted in Figure 3:

- Variables are not modified. Nevertheless, observe that now their domain is different, i.e., they denote pairs of the form $(c(\overline{x_n}), k)$.

– For constructor-rooted terms, their associated cost is always zero, since no function unfolding is needed to produce them.
– Each function call $f(x_1, \ldots, x_k)$ is replaced by its counterpart in the transformed program, i.e., a call of the form $f_c(x_1, \ldots, x_n)$.
– Case expressions are transformed by leaving the case structure and then replacing each pattern p_i by the pair (p_i, k_i), where k_i denotes the cost of producing the pattern p_i (i.e., the cost of evaluating the case argument to some head normal form). Then, this cost is added to the transformed expression in the selected branch.
– Finally, let bindings are transformed by applying recursively function $\mathcal{C}$ to all the expressions.

Theorem 1 states the correctness of our approach. In this result, we only consider *ground* expressions—without free variables—as initial calls, i.e., we only consider "functional" computations. The use of non-ground calls will be discussed in Section 4.1. Below, we denote by Δ^c the heap obtained from Δ by replacing every binding $x \mapsto e$ in Δ by $x \mapsto \mathcal{C}[\![e]\!]$.

Theorem 1. *Let P be a program and let P_{cost} be its transformed version obtained by applying Definition 1. Then, given a ground expression e, we have $[\,] : e \Downarrow_k \Delta : v$ w.r.t. P iff $[\,] : \mathcal{C}[\![e]\!] \Downarrow \Delta^c : (\mathcal{C}[\![v]\!], k)$ w.r.t. P_{cost}.*

Roughly speaking, we obtain the same cost by evaluating an expression w.r.t. the original program and the cost-augmented semantics defined in Figure 2 and by evaluating it w.r.t. the transformed program and the standard semantics.

4 The Transformation in Practice

In this section, we illustrate the usefulness of our approach by means of some selected examples and point out possible applications of the new technique.

4.1 Complexity Analysis

The generated time-equations are simple and, thus, they are appropriate to reason about the cost of evaluating expressions in our lazy language. We consider an example which shows the behavior of lazy evaluation:

```
hd xs   = case xs of {(y : ys) → y}
from x  = let z = S x
              y = from z
          in x : y
nth n xs = case n of {Z → hd xs; (S m) → case xs of {(y : ys) → nth m ys}}
```

where natural numbers are built from `Z` and `S`. Function `hd` returns the first element of a list, function `from` computes an infinite list of consecutive numbers (starting at its argument), and function `nth` returns the element in the n-th

position of a given list. The transformed program is as follows:

$$\begin{array}{l}
\mathtt{hd_c\ xs} \quad = \mathtt{case\ xs\ of}\ \{(\mathtt{y:ys}, \mathtt{k_{xs}}) \to \mathtt{y} \mathbin{\underline{+}} \mathtt{k_{xs}}\} \mathbin{\underline{+}} 1 \\
\mathtt{from_c\ x} \quad = \mathtt{let\ z} = (\mathtt{S\ x}, 0),\ \mathtt{y} = \mathtt{from_c\ z}\ \ \mathtt{in}\ (\mathtt{x:y}, 0) \mathbin{\underline{+}} 1 \\
\mathtt{nth_c\ n\ xs} = \mathtt{case\ n\ of} \\
\qquad\qquad \{(\mathtt{Z}, \mathtt{k_n}) \to \mathtt{hd_c\ xs} \mathbin{\underline{+}} \mathtt{k_n}; \\
\qquad\qquad\ (\mathtt{S\ m}, \mathtt{k_n}) \to \mathtt{case\ xs\ of} \\
\qquad\qquad\qquad\qquad \{(\mathtt{y:ys}, \mathtt{k_{xs}}) \to \mathtt{nth_c\ m\ ys}\ \mathbin{\underline{+}} \mathtt{k_{xs}}\} \mathbin{\underline{+}} \mathtt{k_n}\} \mathbin{\underline{+}} 1
\end{array}$$

Consider, for instance, the function call[5] "`nth (S (S (S Z))) (from Z)`", which returns the value `(S (S (S Z)))` with 9 function unfoldings. Consequently, the transformed call "$\mathtt{nth_c}\ (\mathtt{S}\ (\mathtt{S}\ (\mathtt{S}\ (\mathtt{Z}, 0), 0), 0), 0)\ (\mathtt{from}\ (\mathtt{Z}, 0))$" returns the pair $((\mathtt{S}\ (\mathtt{S}\ (\mathtt{S}\ (\mathtt{Z}, 0), 0), 0), 0), 9)$. Let us now reason about the complexity of the original program. First, the equations of the transformed program can easily be transformed as follows:[6]

$$\begin{array}{ll}
\mathtt{hd_c}\ (\mathtt{y:ys},\ \mathtt{k_{xs}}) & = \mathtt{y} \mathbin{\underline{+}} (\mathtt{k_{xs}} + 1) \\
\mathtt{from_c\ x} & = (\mathtt{x} : \mathtt{from_c}\ (\mathtt{S\ x}, 0),\ 1) \\
\mathtt{nth_c}\ (\mathtt{Z},\ \mathtt{k_n})\ \mathtt{xs} & = (\mathtt{hd_c\ xs}) \mathbin{\underline{+}} (\mathtt{k_n} + 1) \\
\mathtt{nth_c}\ (\mathtt{S\ m},\ \mathtt{k_n})\ (\mathtt{y:ys},\ \mathtt{k_{xs}}) & = (\mathtt{nth_c\ m\ ys}) \mathbin{\underline{+}} (\mathtt{k_{xs}} + \mathtt{k_n} + 1)
\end{array}$$

Here, we basically used inlining (for let expressions) and have moved the arguments of the case expressions to the left-hand sides of the rules. Now, we consider a generic call of the form $\mathtt{nth_c}\ \mathtt{v_1}\ (\mathtt{from}\ \mathtt{v_2})$, where $\mathtt{v_1}$ and $\mathtt{v_2}$ are constructor terms (hence all the associated costs are zero). By unfolding the call to $\mathtt{from_c}$ and replacing $\mathtt{k_n}$ by 0 (since $\mathtt{v_1}$ is a constructor term), we get:

$$\begin{array}{ll}
\mathtt{nth_c}\ (\mathtt{Z},\ 0)\ (\mathtt{x} : \mathtt{from_c}\ (\mathtt{S\ x},\ 0),\ 1) & = (\mathtt{hd_c}\ (\mathtt{x} : \mathtt{from_c}\ (\mathtt{S\ x},\ 0),\ 1)) \mathbin{\underline{+}} 1 \\
\mathtt{nth_c}\ (\mathtt{S\ m},\ 0)\ (\mathtt{x} : \mathtt{from_c}\ (\mathtt{S\ x},\ 0),\ 1) & = (\mathtt{nth_c\ m}\ (\mathtt{from_c}\ (\mathtt{S\ x},\ 0))) \mathbin{\underline{+}} 2
\end{array}$$

By unfolding the call to $\mathtt{hd_c}$ in the first equation, we have:

$$\begin{array}{ll}
\mathtt{nth_c}\ (\mathtt{Z},\ 0)\ (\mathtt{x} : \mathtt{from_c}\ (\mathtt{S\ x},\ 0),\ 1) & = \mathtt{x} \mathbin{\underline{+}} 3 \\
\mathtt{nth_c}\ (\mathtt{S\ m},\ 0)\ (\mathtt{x} : \mathtt{from_c}\ (\mathtt{S\ x},\ 0),\ 1) & = (\mathtt{nth_c\ m}\ (\mathtt{from_c}\ (\mathtt{S\ x},\ 0))) \mathbin{\underline{+}} 2
\end{array}$$

Therefore, each recursive call to $\mathtt{nth_c}$ requires 2 steps, while the base case requires 3 steps, i.e., the total cost of a call $\mathtt{nth}\ \mathtt{v_1}\ (\mathtt{from}\ \mathtt{v_2})$ is $n * 2 + 3$, where n is the natural number represented by $\mathtt{v_1}$.

A theorem prover may help to perform this task, this is an interesting (and difficult) problem for further research. Nevertheless, we think that *logical variables* may help to perform this task. For instance, if one executes the function call $\mathtt{nth_c\ x}\ (\mathtt{from_c}\ (\mathtt{Z},\ 0))$, where `x` is a logical variable, we get the following (infinite) sequence (computed bindings for `x` are applied to the initial call):

$$\begin{array}{ll}
\mathtt{nth_c}\ (\mathtt{Z},\ \mathtt{n_1})\ (\mathtt{from_c}\ (\mathtt{Z},\ 0)) & \Longrightarrow (\mathtt{Z},\ 3 + \mathtt{n_1}) \\
\mathtt{nth_c}\ (\mathtt{S}\ (\mathtt{Z}, \mathtt{n_1}),\ \mathtt{n_2})\ (\mathtt{from_c}\ (\mathtt{Z},\ 0)) & \Longrightarrow (\mathtt{S}\ (\mathtt{Z},\ 0),\ 5 + \mathtt{n_1} + \mathtt{n_2}) \\
\mathtt{nth_c}\ (\mathtt{S}\ (\mathtt{S}\ (\mathtt{Z}, \mathtt{n_1}),\ \mathtt{n_2}),\ \mathtt{n_3})\ (\mathtt{from_c}\ (\mathtt{Z},\ 0)) & \Longrightarrow (\mathtt{S}\ (\mathtt{S}\ (\mathtt{Z},\ 0), 0),\ 7 + \mathtt{n_1} + \mathtt{n_2} + \mathtt{n_3}) \\
\ldots &
\end{array}$$

---

[5] Here, we *inline* the let bindings to ease the reading of the function calls.

[6] Since we only consider ground initial calls—and program rules have no extra-variables—we know that case expressions will never suspend.

If one considers that $\mathtt{n}_1 = \mathtt{n}_2 = \mathtt{n}_3 = \mathtt{0}$ (i.e., that the first input argument is a constructor term), then it is fairly easy to check that 2 steps are needed for each constructor "S" of the first input argument of $\mathtt{nth}_\mathtt{c}$, and 3 more steps for the final constructor "Z". Therefore, we have $2 * n + 3$, where n is the natural number represented by the first input argument of $\mathtt{nth}_\mathtt{c}$ (i.e., we obtain the same result as with the previous analysis).

On the other hand, there are already some practical systems, like `ciaopp` [9], which are able to solve certain classes of equations. They basically try to match the produced recurrence with respect to some predefined "patterns of equations" which represent classes of recurrences whose solution is known. These ideas could be also adapted to our context.

4.2 Other Applications and Extensions

Here we discuss some other practical applications which could be done by extending the developments in this work.

There are several approaches in the literature to *symbolic profiling* (e.g., [5, 15]). Here, the idea is to extend the standard semantics with the computation of cost information (similarly to our step-counting semantics, but including different cost criteria like case evaluations, function applications, etc). A drawback of this approach is that one should modify the language interpreter or compiler in order to implement it. Unfortunately, this possibility usually implies a significant amount of work. An alternative could be the development of a meta-interpreter extended with the computation of symbolic costs, but the execution of such a meta-interpreter would involve a significant overhead at runtime.

In this context, our program transformation could be seen as a preprocessing stage (during compilation), which produces an *instrumented* program whose execution (in a standard environment) returns not only the computed values but also the costs of computing such values. However, in this approach, the profiling results should take into account *sharing*, since almost all implementations of lazy functional (logic) languages allow the sharing of common variables. In this case, we can still use our time-equations by performing an additional phase of *linearization* for the right-hand sides of those rules with multiple (free) occurrences of the same variable. Consider, for instance, the following simple program:

$$\begin{array}{lcl} \mathtt{main} & = & \mathtt{let\ x = f,\ y = g\ x\ in\ h\ x\ y} \\ \mathtt{h\ x\ y} & = & \mathtt{case\ x\ of\ \{Z \to case\ y\ of\ \{Z \to Z\}\}} \\ \mathtt{g\ x} & = & \mathtt{x} \end{array} \qquad \mathtt{f} \ = \ \mathtt{Z}$$

The (slightly simplified) time-equations for this program are as follows:

$$\begin{array}{lcl} \mathtt{main_c} & = & \mathtt{let\ x = f_c,\ y = g_c\ x\ in\ (h_c\ x\ y) \mathbin{+\!\!+} 1} \\ \mathtt{h_c\ x\ y} & = & \mathtt{case\ x\ of\ \{(Z, k_x) \to case\ y\ of\ \{(Z, k_y) \to (Z, k_x + k_y + 1)\}\}} \\ \mathtt{g_c\ x} & = & \mathtt{x \mathbin{+\!\!+} 1} \end{array} \qquad \mathtt{f_c} \ = \ \mathtt{(Z, 1)}$$

If we do not consider sharing, the evaluation of function `main` requires 5 function unfoldings ($\mathtt{main_c}$, $\mathtt{g_c}$, $\mathtt{h_c}$, and $\mathtt{f_c}$ twice). Our equations correctly model this cost. However, under a sharing-based implementation, the evaluation of `main`

only requires 4 function unfoldings, since function $\mathtt{f_c}$ is only unfolded once. We can tackle this situation by introducing new logical variables for shared variables and then imposing the constraint that all shared variables *but one* must have an associated cost of zero. For instance, the rule for $\mathtt{main_c}$ is now transformed to:

$$\mathtt{main_c} \;=\; \mathtt{let}\;\; \mathtt{x} = \mathtt{f_c},\;\; \mathtt{y} = \;\mathtt{g_c}\;\mathtt{x'}\;\;\mathtt{in}\;\;(\mathtt{h'_c}\;\mathtt{x'}\;\mathtt{x}\;\mathtt{y}) \mathbin{+\!\!+} 1$$

where x' is a new logical variable and function $\mathtt{h'_c}$ is defined in the following way:

$$\mathtt{h'_c}\;\mathtt{x'}\;\mathtt{x}\;\mathtt{y} \mid \mathtt{one}\;\mathtt{x}\;\mathtt{x'} \;=\; \mathtt{h}\;\mathtt{x}\;\mathtt{y}$$

This is a guarded rule, where the right-hand side is only evaluated if the constraint $\mathtt{one}\;\mathtt{x}\;\mathtt{x'}$ succeed. The definition of `one` should ensure that there is (at most) one cost which is not zero:

$$\begin{array}{lcl} \mathtt{one}\;(\mathtt{Z},\;0)\;(\mathtt{Z},\;0) & = & \mathtt{success} \\ \mathtt{one}\;(\mathtt{Z},\;0)\;(\mathtt{Z},\;\mathtt{k}) & = & \mathtt{k} > 0 \end{array} \qquad \mathtt{one}\;(\mathtt{Z},\;\mathtt{k})\;(\mathtt{Z},\;0) \;=\; \mathtt{k} > 0$$

This idea can be generalized so that the generated time-equations are still a good basis to perform symbolic profiling of lazy functional logic languages which model sharing. Of course, this will make the transformed program much harder to understand and analyze. Nevertheless, it will be only used internally by the language environment to generate profiling results, so this is not a problem.

Finally, the proposed transformation could also be useful in the context of *program debugging*. In particular, some abstract interpretation based systems, like `ciaopp` [9], allows the user to include annotations in the program with the predicted cost. This cost will be then compared with the one automatically generated by the compiler (here, our program transformation can be useful to compute this cost). If both cost functions are not equivalent, this might indicate a programmer error; for instance, the program may contain undesired loops which directly affect the time complexity.

5 Conclusions

In this paper we introduced a novel program transformation to measure the cost of lazy functional (logic) computations. This is a difficult task mainly due to the *on-demand* nature of lazy evaluation. Indeed, previous approaches to this problem only compute an approximation of the result or give exact time equations which cannot be mechanically solved. Our approach is simple, precise—i.e., it computes exact costs rather than upper/lower bounds—, and fully automatic. In order to check the applicability of the ideas presented in this paper, a prototype implementation of the program transformation has been undertaken.

There are several possibilities for future work. In this work, we sketched the use of logical variables for reasoning about the complexity of a program. This idea can be further developed and, perhaps, combined with partial evaluation so that we get a simpler representation of the time-equations for a given expression. The extension of our developments to cope with higher-order functions seems essential to be able to analyze typical lazy functional (logic) programs. For instance, this can be done by *defunctionalization*, i.e., by including an explicit

(first-order) application operator. Finally, time-equations can be augmented with more cost information so that they can be used for profiling. In this case, avoiding the multiple computation of costs for shared variables is required. As we mentioned before, this can be done by linearization of the right-hand sides with new logical variables and, then, by introducing appropriate constraints on these logical variables.

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. In *Proc. of WRS 2002*, volume 70(6) of *ENTCS*. Elsevier Science Publishers, 2002.
3. E. Albert, M. Hanus, and G. Vidal. A Practical Partial Evaluation Scheme for Multi-Paradigm Declarative Languages. *Journal of Functional and Logic Programming*, 2002(1), 2002.
4. E. Albert and G. Vidal. The Narrowing-Driven Approach to Functional Logic Program Specialization. *New Generation Computing*, 20(1):3–26, 2002.
5. E. Albert and G. Vidal. Symbolic Profiling of Multi-Paradigm Declarative Languages. In *Proc. of LOPSTR'01*, pages 148–167. Springer LNCS 2372, 2002.
6. B. Bjerner and S. Holmström. A Compositional Approach to Time Analysis of First-Order Lazy Functional Programs. In *Proc. of FPCA'89*. ACM Press, 1989.
7. Yoshihiko Futamura. Partial Evaluation of Computation Process—An Approach to a Compiler-Compiler. *Higher-Order and Symbolic Computation*, 12(4):381–391, 1999. Reprint of article in *Systems, Computers, Controls* 1971.
8. M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, 19&20:583–628, 1994.
9. M. Hermenegildo, G. Puebla, F. Bueno, and P. López-García. Program Development Using Abstract Interpretation (and The Ciao System Preprocessor). In *10th International Static Analysis Symposium (SAS'03)*. Springer-Verlag, June 2003.
10. N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Englewood Cliffs, NJ, 1993.
11. J. Launchbury. A Natural Semantics for Lazy Evaluation. In *Proc. of POPL'93*, pages 144–154. ACM Press, 1993.
12. Y.A. Liu and G. Gómez. Automatic accurate time-bound analysis for high-level languages. In *Proc. of the ACM SIGPLAN 1998 Workshop on Languages, Compilers, and Tools for Embbeded Systems*, pages 31–40. Springer LNCS 1474, 1998.
13. D. Le Métayer. Analysis of Functional Programs by Program Transformations. In *Proc. of 2nd France-Japan Artificial Intelligence and Computer Science Symposium*. North-Holland, 1988.
14. D. Sands. Complexity Analysis for a Lazy Higher-Order Language. In *Proc. of ESOP'90*. Springer LNCS 432, 1990.
15. P.M. Sansom and S.L. Peyton-Jones. Formally Based Profiling for Higher-Order Functional Languages. *ACM TOPLAS*, 19(2):334–385, 1997.
16. G. Vidal. Cost-Augmented Narrowing-Driven Specialization. In *Proc. of PEPM'02*, pages 52–62. ACM Press, 2002.
17. P. Wadler. Strictness Analysis aids Time Analysis. In *Proc. of POPL'88*. ACM Press, 1988.