# Measuring the Effectiveness of Partial Evaluation in Functional Logic Languages

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

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

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

## 1 Introduction

Partial evaluation is a source-to-source program transformation technique for specializing programs w.r.t. parts of their input (hence also called *program specialization*). This technique has been studied, among others, in the context of functional [12, 21], logic [14, 25], and functional logic [6, 22] programming languages. A common motivation of all partial evaluation techniques is to improve the efficiency of a program while preserving its meaning. Rather surprisingly, relatively little attention has been paid to the development of formal methods for reasoning about the effectiveness of this program transformation; usually, only experimental tests on particular languages and compilers are undertaken. Clearly, a machine-independent way of measuring the effectiveness of partial evaluation would be useful to both users and developers of partial evaluators.

Predicting the speedup achieved by partial evaluators is generally undecidable. We mention below some approaches to this problem. Andersen and Gomard's *speedup analysis* [8] predicts a relative interval of the speedup achieved by a program specialization. Nielson's type system [29] formally expresses when a partial evaluator is better than another. Other interesting efforts investigate *cost analyses* for logic and functional programs which may be useful for determining the effectiveness of program transformations. For instance, Debray and Lin's method [13] for the semiautomatic analysis of the worst-case cost of a large class of logic programs and Sands's theory of cost equivalence [31] for reasoning about the computational cost of *lazy* functional programs. Laziness introduces a considerable difficulty since the cost of lazy (call-by-name) computations is not *compositional* [31]. Although both the above cost analyses can be used to study the effectiveness of partial evaluation, their authors did not address this issue.

All these efforts mainly base the cost of executing a program on the number of steps performed in a computation. However, simple experiments show that the number of steps and the computation time are not easily correlated. Consider, for instance, the positive supercompiler described in [35]. As noted by Sørensen in [34, Chapter 11], the residual program obtained by positive supercompilation —without the *postunfolding* phase—performs *exactly* the same number of steps as the original one. This does not mean that the process is useless. Rather, all intermediate data structures are gone, and such structures take up space and garbage collection time in actual implementations. Furthermore, the reduction in number of steps of a computation does not imply a proportional reduction in its execution time. The following example illustrates this point. In this work we consider a first-order language. However, we use a curried notation in the examples as usual in functional languages.

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

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

and the following partial evaluation w.r.t. app x y obtained by the partial evaluator INDY [2]:

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

Note that no input data have been provided for the specialization. In spite of this, INDY can still improve programs by shortening computations and removing intermediate data structures. In particular, this residual program computes the same function as the original one but in approximately half the number of steps. This might suggest that the execution time of app2s (for sufficiently large inputs) should be about one half the execution time of app. However, executions of function app2s in several environments (e.g., in the lazy functional language Hugs [19] and the functional logic language Curry [17]) show that speedup is only around 10%.

In order to reason about these counterintuitive results, we introduce several formal criteria to measure the efficiency of a functional logic computation. We consider *inductively sequential* rewrite systems as programs. Inductive sequentiality ensures strong desirable properties of evaluations. In particular, if a term has a value, there is a sound, complete and efficient algorithm to find this value. Inductive sequentiality is not a limiting condition for programming. In fact, the first order components of many functional programs, e.g., Haskell and ML, are inductively sequential. Essentially, a rewrite system is *inductively sequential* when all its operations are defined by rewrite rules that, recursively, make on their arguments a case distinction analogous to a data type (or structural) induction.

The strategy that determines what to evaluate in a term is based on this case distinction and is called *needed narrowing* [10]. Needed narrowing, which extends the usual call-by-name semantics of functional computations, has been proved *optimal* for functional logic computations as well. We formally define the cost of evaluating an expression in terms of the number of steps, the number of function applications, and the complexity of pattern-matching or unification involved in the computation. Similar criteria are taken into account (though experimentally) in traditional profiling approaches (e.g., [33]). Let us remark that our aim is not to define a complex cost analysis for functional logic programs, but to introduce some representative cost criteria and then investigate their variations by the application of a particular partial evaluation method. The above criteria seem specially well-suited to estimate the speedup achieved by the narrowing-driven partial evaluation scheme of [6]. Nevertheless, our technique can be easily adapted to other related partial evaluation methods, like partial deduction [25] and positive supercompilation [35]. In particular, we use *recurrence equations* to compare the cost of executing the original and residual programs. These equations can be automatically derived from the partial evaluation process itself and are parametric w.r.t. the considered cost criteria. Unlike traditional recurrence equations used to reason about the complexity of programs, our equations are defined on data structures rather than on natural numbers. This complicates the computation of their solutions, although in some cases useful statements about the improvements achieved by partial evaluation can be made by a simple inspection of the sets of equations. In other cases, these equations can be transformed into traditional recurrence equations and then solved by well-known mathematical methods.

The remainder of the paper is organized as follows. Section 2 introduces some preliminary definitions. Section 3 defines several formal criteria to measure the cost of a computation. Section 4 addresses the problem of determining the improvement achieved by the partial evaluation process in the context of a functional logic language and Sect. 5 illustrates its usefulness by showing some possible applications. Section 6 discusses some related work and Sect. 7 concludes. An extended version of this paper can be found in [3].

## 2 Preliminaries

For the sake of completeness, we recall in this section some basic notions of term rewriting [11] and functional logic programming [16]. We consider a (*many-sorted*) *signature* $\Sigma$ partitioned into a set $\mathcal{C}$ of *constructors* and a set $\mathcal{F}$ of (defined) *functions* or *operations*. We write $c/n \in \mathcal{C}$ and $f/n \in \mathcal{F}$ for $n$-ary constructor and operation symbols, respectively. There is at least one sort *Bool* containing the constructors `True` and `False`. The set of *terms* and *constructor terms* with *variables* (e.g., $x, y, z$) from $\mathcal{V}$ are denoted by $\mathcal{T}(\mathcal{C} \cup \mathcal{F}, \mathcal{V})$ and $\mathcal{T}(\mathcal{C}, \mathcal{V})$, respectively. The set of variables occurring in a term $t$ is denoted by $\mathcal{V}ar(t)$. A term is *linear* if it does not contain multiple occurrences of one variable.

A *pattern* is a term of the form $f(d_1, \ldots, d_n)$ where $f/n \in \mathcal{F}$ and $d_1, \ldots, d_n \in \mathcal{T}(\mathcal{C}, \mathcal{V})$. A term is *operation-rooted* if it has an operation symbol at the root. A *position* $p$ in a term $t$ is represented by a sequence of natural numbers ($\Lambda$ denotes the empty sequence, i.e., the root position). $t|_p$ denotes the *subterm* of $t$ at position $p$, and $t[s]_p$ denotes the result of *replacing the subterm* $t|_p$ by the term $s$. We denote a *substitution* $\sigma$ by $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ with $\sigma(x_i) = t_i$ for $i = 1, \ldots, n$ (with $x_i \neq x_j$ if $i \neq j$), and $\sigma(x) = x$ for all other variables $x$. The set $\mathcal{D}om(\sigma) = \{x \in \mathcal{V} \mid \sigma(x) \neq x\}$ is called the *domain* of $\sigma$. A substitution $\sigma$ is *constructor*, if $\sigma(x)$ is a constructor term for all $x$. The identity substitution is denoted by $\{\,\}$. A substitution $\theta$ is more general than $\sigma$, in symbols $\theta \leq \sigma$, iff there exists a substitution $\gamma$ such that $\gamma \circ \theta = \sigma$. A term $t'$ is a (constructor) *instance* of $t$ if there is a (constructor) substitution $\sigma$ with $t' = \sigma(t)$.

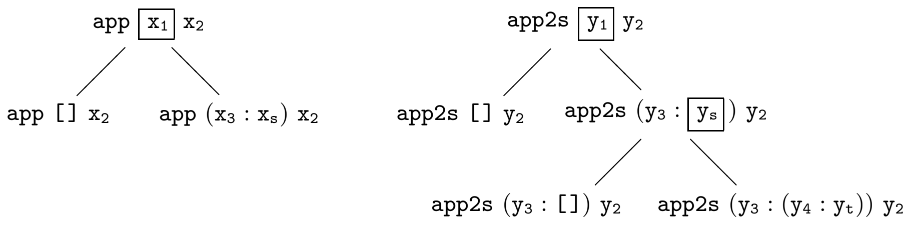
A set of rewrite rules $l \to r$ such that $l \notin \mathcal{V}$, and $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$ is called a *term rewriting system* (TRS). Terms $l$ and $r$ are called the *left-hand side* and the *right-hand side* of the rule, respectively. A TRS $\mathcal{R}$ is *left-linear* if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

To evaluate terms containing variables, *narrowing* non-deterministically instantiates these variables so that a rewrite step is possible. Formally, $t \leadsto_{(p,R,\sigma)} t'$ is a *narrowing step* if $p$ is a non-variable position in $t$ and $\sigma(t) \to_{p,R} t'$. We often write $t \leadsto_\sigma t'$ when the position and the rule are clear from the context. We denote by $t_0 \leadsto^n_\sigma t_n$ a sequence of $n$ narrowing steps $t_0 \leadsto_{\sigma_1} \ldots \leadsto_{\sigma_n} t_n$ with $\sigma = \sigma_n \circ \cdots \circ \sigma_1$. (If $n = 0$ then $\sigma = \{\,\}$.) Due to the presence of free variables, an expression may be reduced to different values after instantiating free variables to different terms. Given a narrowing derivation $t_0 \leadsto^*_\sigma t_n$, we say that $t_n$ is a computed *value* and $\sigma$ is a computed *answer* for $t_0$. To avoid unnecessary narrowing computations and to provide computations with infinite data structures, as well as a demand-driven generation of the search space, the most recent work has advocated *lazy* narrowing strategies (e.g., [15, 26, 28]). In this paper we consider *needed* narrowing [10], an evaluation strategy which is based on the idea of evaluating only subterms that are needed, in a precise technical sense, to obtain a result.

**Needed Narrowing.** Needed narrowing is an optimal evaluation strategy w.r.t. both the length of derivations and the independence of computed solutions [10]. It extends the Huet and Lévy notion of a needed rewrite step [18] to functional logic programming. Following [10], a narrowing step $t \leadsto_{(p,R,\sigma)} t'$ is called *needed* iff, for every substitution $\theta$ such that $\sigma \leq \theta$, $p$ is the position of a needed redex of $\theta(t)$ in the sense of [18]. A narrowing derivation is called *needed* iff every step of the derivation is needed.

An efficient implementation of needed narrowing exists for *inductively sequential* programs. The formal definition of inductive sequentiality is rather technical. In this paper, for the sake of completeness, we give a more intuitive account of this concept. The complete technical details are in [9].

A rewrite system is *inductively sequential* when all its operations are defined by rewrite rules that, recursively, make on their arguments a case distinction analogous to a data type (or structural) induction. Both definitions of Example 1 show this point. In fact, operation `app` makes a case distinction on its first argument. The type of this argument is *list*. A structural induction on *list* needs to consider two cases, namely `[]` and a list consisting of a head and a tail. The advantage of this disciplined approach is that in any term `app x y`, it is necessary and sufficient to evaluate argument `x` to a head normal form in order to fire a rule of `app`. Operations defined according to this principle are called *inductively sequential* as well and their rules can be organized in a hierarchical structure called a *definitional tree*. We show below these trees for the operations of Example 1.



app $\boxed{x_1}$ $x_2$
app [] $x_2$
app ($x_3$ : $x_s$) $x_2$
app2s $\boxed{y_1}$ $y_2$
app2s [] $y_2$
app2s ($y_3$ : $\boxed{y_s}$) $y_2$
app2s ($y_3$ : []) $y_2$
app2s ($y_3$ : ($y_4$ : $y_t$)) $y_2$

The leaves of these trees are (modulo a renaming of variables) the left-hand sides of the operations' rewrite rules. It is easy to see that operation `app2s` makes an initial case distinction on its first argument. Then, it makes a second case distinction that leads to its three rewrite rules. The arguments' target of a case distinction are shown in a box and are usually called the *inductive positions* of the tree. Thus, to evaluate an expression `app2s` $y_1$ $y_2$, we first evaluate $y_1$ to a head normal form. If the result of this evaluation is `[]`, we apply the first rule. If the result of this evaluation is of the form $y_3$ : $y_s$, we evaluate $y_s$ to a head normal form which will eventually determine which rule, if any, to fire.

There exists a needed narrowing strategy, denoted by $\lambda$ in [10, Def. 13], which determines what to evaluate in a term based on this case distinction. Following [10], to compute needed narrowing steps for an operation-rooted term $t$, we take a definitional tree $\mathcal{P}$ for the root of $t$ and compute $\lambda(t, \mathcal{P})$. Then, for all $(p, l \to r, \sigma) \in \lambda(t, \mathcal{P})$, we say that $t \leadsto_{(p,l\to r,\sigma)} t'$ is a *needed narrowing step*, where $t' = \sigma(t[r]_p)$. Informally speaking, given an operation-rooted term and an associated definitional tree for the root of this term, needed narrowing applies a rule to the entire term, if possible, or checks the subterm corresponding to an inductive position of the tree: if it is a variable, it is instantiated to the constructor of a child; if it is already a constructor, we proceed with the corresponding child;

finally, if it is a function, we evaluate it by recursively applying needed narrowing. The extension of function $\lambda$ to constructor-rooted terms is straightforward. Essentially, to compute a needed narrowing step of a constructor-rooted term, it suffices to compute a needed narrowing step of any of its maximal operation-rooted subterms. We call $\lambda$-*derivation* a narrowing derivation computed by $\lambda$.

*Example 2.* Consider the following rules which define the less-or-equal relation "$\leqslant$" and the addition on natural numbers which are represented by terms built from `0` and `Succ`:

$$\begin{array}{rcllrclcl} \mathtt{0} & \leqslant & \mathtt{n} & \to \mathtt{True} & \mathtt{0} & + & \mathtt{n} & \to & \mathtt{n} \\ (\mathtt{Succ}\ \mathtt{m}) & \leqslant & \mathtt{0} & \to \mathtt{False} & (\mathtt{Succ}\ \mathtt{m}) & + & \mathtt{n} & \to & \mathtt{Succ}\ (\mathtt{m}+\mathtt{n}) \\ (\mathtt{Succ}\ \mathtt{m}) & \leqslant & (\mathtt{Succ}\ \mathtt{n}) & \to \mathtt{m} \leqslant \mathtt{n} & & & & & \end{array}$$

Then the function $\lambda$ computes the following set for the initial term $\mathtt{x} \leqslant (\mathtt{x}+\mathtt{x})$:

$$\{(\Lambda, \mathtt{0} \leqslant \mathtt{n} \to \mathtt{True}, \{\mathtt{x} \mapsto \mathtt{0}\}),\ (2, (\mathtt{Succ}\ \mathtt{m}) + \mathtt{n} \to \mathtt{Succ}\ (\mathtt{m}+\mathtt{n}), \{\mathtt{x} \mapsto \mathtt{Succ}\ \mathtt{m}\})\}$$

These steps yield the $\lambda$-derivations:

$$\begin{array}{l} \mathtt{x} \leqslant (\mathtt{x}+\mathtt{x}) \leadsto_{\{\mathtt{x}\mapsto\mathtt{0}\}} \quad \mathtt{True} \\ \mathtt{x} \leqslant (\mathtt{x}+\mathtt{x}) \leadsto_{\{\mathtt{x}\mapsto\mathtt{Succ}\ \mathtt{m}\}} (\mathtt{Succ}\ \mathtt{m}) \leqslant (\mathtt{Succ}\ (\mathtt{m}+(\mathtt{Succ}\ \mathtt{m}))) \end{array}$$

Needed narrowing derivations can be represented by a (possibly infinite) finitely branching *tree*. Formally, given an inductively sequential program $\mathcal{R}$ and an operation-rooted term $t$, a *needed narrowing tree* for $t$ in $\mathcal{R}$ is a tree satisfying the following conditions:

– Each node of the tree is a term.
– The root node is $t$.
– Let $s$ be a node in the tree and assume that $\mathcal{P}$ is a definitional tree for the root of $s$. Then, for each tuple $(p, l \to r, \sigma) \in \lambda(s, \mathcal{P})$, the node has a child $\sigma(s[r]_p)$.
– Nodes which are constructor terms have no children.

Each branch of the needed narrowing tree is a $\lambda$-derivation for $t$ in $\mathcal{R}$.

## 3 Formal Criteria for Measuring Computational Cost

The cost criteria that we introduce below are independent of the particular implementation of the language. Rather, they are formulated for a rewrite system, which we intend as a program, and are based on operations that are, in one form or another, performed by likely implementations of rewriting and narrowing.

The first cost criterion that we consider has been widely used in the literature. This is the *number of steps*, or *length*, of the evaluation of a term. The following trivial definition is presented only for uniformity with the remaining costs.

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

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

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

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

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

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

We note that $\mathcal{P}$ gives a worst-case measure of the pattern matching effort. In particular, if a ground expression $e$ is evaluated using rule $R$, then $\mathcal{P}(R)$ returns exactly the number of constructor symbols of $e$ whose inspection is required. However, whenever $e$ contains free variables, the value returned by $\mathcal{P}(R)$ represents an upper bound.

For simplicity, we do not consider non-deterministic computations, although our cost measures could be extended along the lines of [13]. On the other hand, many partial evaluation methods do not change the non-determinism of computations, i.e., although some paths become shorter, the *search spaces* of a given goal in the original and residual programs have essentially the same structure. In particular, this is the case of the narrowing-driven partial evaluation method (see discussion in Sect. 5.2). Therefore, a cost measure which quantifies the amount of non-determinism would not be affected by our partial evaluation method.

In the remainder of this paper, we denote by $C$ any cost criterion, i.e., $C$ stands for $\mathcal{S}$, $\mathcal{A}$ or $\mathcal{P}$. Furthermore, most of the developments in the following sections are independent of the considered criteria; thus, $C$ could also denote more elaborated criteria, like the number of variable bindings, the “size” of the reduced expression, etc.

The previous definitions allow us to define the cost of a derivation as the total cost of its steps.

**Definition 4 (cost of a derivation).** *Let $\mathcal{R}$ be a program and $t_0$ a term. Let $C$ denote a cost criterion. We overload function $C$ by partially defining it on derivations as follows. If $D : t_0 \leadsto_{(p_1,R_1,\sigma_1)} t_1 \leadsto_{(p_2,R_2,\sigma_2)} \cdots \leadsto_{(p_n,R_n,\sigma_n)} t_n$ is a derivation for $t_0$ in $\mathcal{R}$, then $C(D) = \sum_{i=1}^{n} C(R_i)$.*

For the developments in the next section, it is more convenient to reason about the efficiency of programs when a cost measure is defined over terms rather than entire *computations*. We use "computation" as a generic word for the *evaluation* of a term, i.e., a narrowing derivation ending in a constructor term. In general, different strategies applied to a same term may produce evaluations of different lengths and/or fail to terminate. For instance, if term $t$ contains uninstantiated variables, there may exist distinct evaluations of $t$ obtained by distinct instantiations of $t$'s variables. Luckily, needed narrowing gives us some leeway. We allow uninstantiated variables in a term $t$ as long as these variables are not instantiated during its evaluation, i.e., we have a derivation of the form $t \leadsto^*_{\{\}} d$. In this case, there is a concrete implementation of needed narrowing, i.e., that denoted by $\lambda$ in Sect. 2, in which $t \leadsto^*_{\{\}} d$ is the only possible derivation computed from $t$ (see Lemma 1 in [3]). Therefore, to be formal, we consider only $\lambda$-derivations in the technical results of this paper. Note that this is not a practical restriction in our context, since $\lambda$-derivations are efficient, easy to compute, and used in the implementations of modern functional logic languages such as Curry [17] and Toy [27]. The above property allows us to define the cost of a term as the cost of its $\lambda$-derivation when the computed substitution is empty (since it is unique).

**Definition 5 (cost of a term).** *Let $\mathcal{R}$ be a program and $t$ a term. Let $C$ denote a cost criterion. We overload function $C$ by partially defining it on terms and programs as follows. If $t \leadsto_{(p_1,R_1,\sigma_1)} \ldots \leadsto_{(p_k,R_k,\sigma_k)} d$ is a $\lambda$-derivation, where $d$ is a constructor term and $\sigma_1 = \cdots = \sigma_k = \{\}$ (i.e., $\sigma_i = \{\}$, for $i = 1, 2, \ldots, k$), we define $C(t, \mathcal{R}) = \sum_{i=1}^{k} C(R_i)$.*

In the following, we often write $\mathrm{C}(t)$ to denote the cost of a term when the program $\mathcal{R}$ is clear from the context.

We apply the previous definitions to Example 1. The next table summarizes (with minor approximations to ease understanding) the cost of computations with both functions when the arguments are constructor terms:

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

Here $n$ represents the *size* of the inputs to the functions, i.e., the number of elements in the first argument of `app` and `app2s` (the second argument does not affect the cost of computations). The first observation is that not all cost criteria have been improved. In fact, the number of constructor applications and the pattern matching effort remain unchanged. To obtain a "global" value of the improvement achieved by a partial evaluation, one might assume an equal unit cost for all criteria. In this way, the total cost of a computation using `app` is $4\,n$. Likewise, the cost of a computation using `app2s` is $3\,n$. The speedup is only 25%.

In general, one should determine the appropriate weight of each cost criterion for a specific language environment. For instance, if we increase the unit cost of $\mathcal{A}_c$ and decrease that of $\mathcal{S}$—a more realistic choice in our environment—the improvement of `app2s` over `app` estimated by our criteria closely explains the lower speedup measured experimentally (10%).

## 4 Measuring the Effectiveness of a Partial Evaluation

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

### 4.1 Narrowing-driven Partial Evaluation

We briefly recall the partial evaluation scheme of [6] which is based on needed narrowing. An intrinsic feature of this approach is the use of the same operational mechanism—needed narrowing—for both execution and partial evaluation. Informally speaking, a *partial evaluation* for a term $s$ in a program $\mathcal{R}$ is computed by constructing a finite (possibly incomplete) needed narrowing tree for this term, and then extracting the *resultants* associated to the root-to-leaf derivations of this tree. Resultants are defined as follows:

**Definition 6 (resultant).** *Let $\mathcal{R}$ be a program and $s$ be a term. Given a needed narrowing derivation $s \leadsto^{+}_{\sigma} t$, its associated resultant is the rewrite rule $\sigma(s) \to t$.*

The potential value of resultants is that they compute in a single step a non-null derivation of the original program.

*Example 3.* Consider the function `app` of Example 1 and the following needed narrowing derivations:

$$
\begin{array}{lll}
\mathtt{app\ (app\ x_s\ y_s)\ z_s} & \leadsto_{\{\mathtt{x_s} \mapsto []\}} & \mathtt{app\ y_s\ z_s} \\
\mathtt{app\ (app\ x_s\ y_s)\ z_s} & \leadsto_{\{\mathtt{x_s} \mapsto \mathtt{x'} : \mathtt{x'_s}\}} & \mathtt{app\ (x' : app\ x'_s\ y_s)\ z_s} \\
 & \leadsto_{\{\,\}} & \mathtt{x' : app\ (app\ x'_s\ y_s)\ z_s}
\end{array}
$$

Then, the associated resultants are:

$$
\begin{array}{ll}
\mathtt{app\ (app\ [\,]\ y_s)\ z_s} & \to \mathtt{app\ y_s\ z_s} \\
\mathtt{app\ (app\ (x' : x'_s)\ y_s)\ z_s} & \to \mathtt{x' : app\ (app\ x'_s\ y_s)\ z_s}
\end{array}
$$

We note that, whenever the specialized call $s$ is not a linear pattern, the left-hand sides of resultants may not be linear patterns either and hence resultants may not be legal program rules (as the above example shows). To overcome this problem, we introduce a post-processing of renaming which also eliminates redundant structures from residual rules. Informally, the renaming transformation proceeds as follows. First, an *independent renaming* $\rho$ for a set of terms $S$ is constructed, which consists of a mapping from terms to terms such that for all $s \in S$, we have $\rho(s) = f(x_1, \ldots, x_n)$, where $x_1, \ldots, x_n$ are the distinct variables in $s$ in the order of their first occurrence and $f$ is a *fresh* function symbol. We also let $\rho(S)$ denote the set $S' = \{\rho(s) \mid s \in S\}$. While the independent renaming suffices to rename the left-hand sides of resultants (since they are constructor instances of the specialized calls), right-hand sides are renamed by means of the auxiliary function $ren_\rho$, which *recursively* replaces each call in the expression by a call to the corresponding renamed function (according to $\rho$).

Given an independent renaming $\rho$ for a set of terms, the auxiliary function $ren_\rho$ is formally defined as follows.

**Definition 7.** *Let $S$ be a set of terms, $t$ a term, and $\rho$ an independent renaming of $S$. The partial function $ren_\rho(S, t)$ is defined inductively as follows:*

$$ren_\rho(S,t) \;=\; \begin{cases} t & \text{if } t \in \mathcal{V} \\ c(t'_1, \ldots, t'_n) & \text{if } t = c(t_1, \ldots, t_n),\ c \in \mathcal{C},\ n \geq 0, \text{ and} \\ & \quad t'_i = ren_\rho(S, t_i),\ i = 1, \ldots, n \\ \theta'(\rho(s)) & \text{if } \exists\theta, \exists s \in S \text{ such that } t = \theta(s) \text{ and} \\ & \quad \theta' = \{x \mapsto ren_\rho(S, \theta(x)) \mid x \in \mathcal{D}om(\theta)\} \end{cases}$$

The above mapping is non-deterministic. An operation-rooted term can be possibly renamed by different symbols obtained using different sequences of terms in $S$ (according to the third case). The mapping can be made deterministic by some simple heuristics (as it is done in the INDY system [2]).

*Example 4.* Consider the set of terms:

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

A possible independent renaming $\rho$ for $S$ is the mapping:

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

By using this independent renaming, we can rename arbitrary expressions by means of function $ren_\rho$. Consider, for instance:

$$\begin{array}{ll} ren_\rho(\texttt{app [] (x : x}_\texttt{s}\texttt{))} & = \texttt{app1s [] (x : x}_\texttt{s}\texttt{)} \\ ren_\rho(\texttt{app (app [] y}_\texttt{s}\texttt{) (z : z}_\texttt{s}\texttt{))} & = \texttt{dapp [] y}_\texttt{s}\texttt{ (z : z}_\texttt{s}\texttt{)} \\ ren_\rho(\texttt{app (app [] y}_\texttt{s}\texttt{) (app z}_\texttt{s}\texttt{ []))} & = \texttt{dapp [] y}_\texttt{s}\texttt{ (app1s z}_\texttt{s}\texttt{ [])} \end{array}$$

Observe that the renaming of the term `app (app [] y`$_\texttt{s}$`) (z : z`$_\texttt{s}$`)` could be `app1s (app1s [] y`$_\texttt{s}$`) (z : z`$_\texttt{s}$`)` as well.

Following [25], in this work we adopt the convention that any derivation is potentially *incomplete* in the sense that at any point we are allowed to simply not select any redex and terminate the derivation; a branch thus can be failed, incomplete, successful, or infinite. A *failing derivation* is a needed narrowing derivation ending in an expression that is neither a constructor term nor can be further narrowed. Given a needed narrowing tree $\mathcal{N}$ for a term $t$ in program $\mathcal{R}$, a *partial* (or *incomplete*) needed narrowing tree $\mathcal{N}'$ for $t$ in $\mathcal{R}$ is obtained by considering only the narrowing derivations from $t$ down to some terms $t_1, \ldots, t_n$ such that $t_1, \ldots, t_n$ appear in $\mathcal{N}$ and each non-failing branch of $\mathcal{N}$ contains exactly one of them. Partial evaluation can be formally defined as follows.

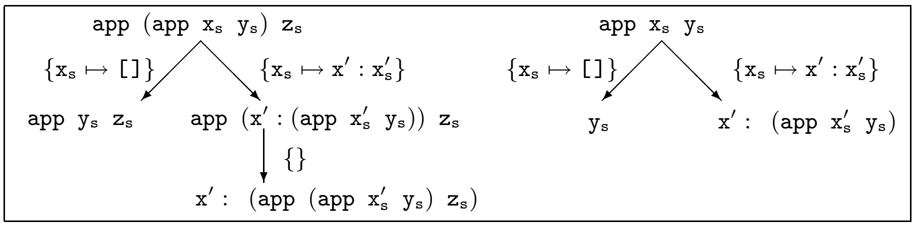


**Fig. 1.** Needed narrowing trees for "app (app $x_s$ $y_s$) $z_s$" and "app $x_s$ $y_s$"

**Definition 8 (partial evaluation).** *Let $\mathcal{R}$ be a program, $S = \{s_1, \ldots, s_n\}$ a finite set of operation-rooted terms, and $\rho$ an independent renaming of $S$. Let $\mathcal{N}_1, \ldots, \mathcal{N}_n$ be finite (possibly incomplete) needed narrowing trees for $s_i$ in $\mathcal{R}$, $i = 1, \ldots, n$. A partial evaluation of $S$ in $\mathcal{R}$ (under $\rho$) is obtained by constructing a renamed resultant, $\sigma(\rho(s)) \to ren_\rho(t)$, for each non-failing needed narrowing derivation $s \leadsto^+_\sigma t$ in $\mathcal{N}_1, \ldots, \mathcal{N}_n$.*

We note that [6] requires, additionally, that no constructor-rooted term is evaluated at partial evaluation time. This restriction is necessary in order to preserve the correctness of the partial evaluation transformation in the context of lazy functional logic languages (see [6] for details). We now illustrate this definition with an example.

*Example 5.* Consider again the function app together with the set of calls:

$$S = \{\texttt{app (app } \mathtt{x_s}\ \mathtt{y_s}\texttt{) } \mathtt{z_s},\ \texttt{app } \mathtt{x_s}\ \mathtt{y_s}\}$$

Given the (incomplete) needed narrowing trees of Figure 1, we produce the following resultants:

$$\begin{array}{ll}
\texttt{app (app [] } \mathtt{y_s}\texttt{) } \mathtt{z_s} & \to \texttt{app } \mathtt{y_s}\ \mathtt{z_s} \\
\texttt{app (app (x : } \mathtt{x_s}\texttt{) } \mathtt{y_s}\texttt{) } \mathtt{z_s} & \to \texttt{x : app (app } \mathtt{x_s}\ \mathtt{y_s}\texttt{) } \mathtt{z_s} \\
\texttt{app [] } \mathtt{y_s} & \to \mathtt{y_s} \\
\texttt{app (x : } \mathtt{x_s}\texttt{) } \mathtt{y_s} & \to \texttt{x : app } \mathtt{x_s}\ \mathtt{y_s}
\end{array}$$

Now, if we consider the independent renaming $\rho$ for $S$ of Example 4:

$$\{\texttt{app (app } \mathtt{x_s}\ \mathtt{y_s}\texttt{) } \mathtt{z_s} \mapsto \texttt{dapp } \mathtt{x_s}\ \mathtt{y_s}\ \mathtt{z_s},\ \texttt{app } \mathtt{x_s}\ \mathtt{y_s} \mapsto \texttt{app1s } \mathtt{x_s}\ \mathtt{y_s}\}$$

we compute the following partial evaluation of $\mathcal{R}$ w.r.t. $S$ (under $\rho$):

$$\begin{array}{ll}
\texttt{dapp [] } \mathtt{y_s}\ \mathtt{z_s} & \to \texttt{app1s } \mathtt{y_s}\ \mathtt{z_s} \\
\texttt{dapp (x : } \mathtt{x_s}\texttt{) } \mathtt{y_s}\ \mathtt{z_s} & \to \texttt{x : dapp } \mathtt{x_s}\ \mathtt{y_s}\ \mathtt{z_s} \\
\texttt{app1s [] } \mathtt{y_s} & \to \mathtt{y_s} \\
\texttt{app1s (x : } \mathtt{x_s}\texttt{) } \mathtt{y_s} & \to \texttt{x : app1s } \mathtt{x_s}\ \mathtt{y_s}
\end{array}$$

We will not discuss in details this transformation since it is not essential for the forthcoming sections, where we mainly deal with the notion of resultant. Nevertheless, a full description of the narrowing-driven approach to partial evaluation, as well as a comparison to related partial evaluation techniques, can be found in [5, 6].

### 4.2 Automatic Generation of Recurrence Equations

In this section we propose the use of recurrence equations to analyze how a cost criterion is affected by the partial evaluation process. Although we will illustrate our proposal over the cost criteria introduced in the previous section, our developments are parametric w.r.t. the considered cost criteria. Our approach is inspired by the standard use of recurrence equations to analyze the complexity of algorithms in terms of their inputs (see, e.g., [1] for imperative, [31] for functional, and [13] for logic programs).

**Definition 9 (recurrence equation).** *Let $\mathcal{R}$ be a program and $s$ be a term. Given a needed narrowing derivation $s \leadsto^{+}_{\sigma} t$, its associated recurrence equation is: $C(\sigma(s)) = C(t) + k$, where $k = C(s \leadsto^{+}_{\sigma} t)$.*

However, we are not interested in arbitrary recurrence equations, or sets of equations, since in general they would not be useful. Rather, we present a technique for deriving sets of recurrence equations tightly associated with the partial evaluation process. These equations, specifically the form in which we present them, are informative about the effectiveness of partial evaluation even in the absence of an explicit solution of the equations.

**Definition 10.** *Let $\mathcal{R}$ be a program, $S$ a finite set of operation-rooted terms, and $\rho$ an independent renaming for $S$. Let $\mathcal{R}'$ be a partial evaluation of $\mathcal{R}$ w.r.t. $S$ (under $\rho$) computed from the finite (possibly incomplete) needed narrowing trees $\mathcal{N}_1, \ldots, \mathcal{N}_n$. We produce a pair of equations:*

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

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

Informally, for each needed narrowing derivation used to construct the partial evaluation, we generate a pair of recurrence equations representing the computational cost of executing goals in the original and residual programs, respectively. We use the notation $equation_1$ / $equation_2$ to emphasize that they represent a kind of *ratio* between the cost of the original and residual programs, as we will discuss later in Sect. 5.2. We note that there is no risk of ambiguity in using the same symbol, $C$, for the (cost) equations associated to both the original and residual programs, since the signatures of $\mathcal{R}$ and $\mathcal{R}'$ are disjoint by definition of partial evaluation. Observe also that if $C = \mathcal{S}$, then $k \geq 1$ and $k' = 1$, i.e., the application of each resultant corresponds to a sequence of one or more rules in the original program.

*Example 6.* Consider the operation `app` of Example 1 and the needed narrowing derivation:

$$\overbrace{\texttt{app (app x y) z}}^{s} \rightsquigarrow_{\{\texttt{x}\mapsto\texttt{x}':\texttt{x}_\texttt{s}\}} \texttt{app (x}' : \texttt{app x}_\texttt{s}\texttt{ y) z} \rightsquigarrow_{\{\,\}} \overbrace{\texttt{x}' : \texttt{app (app x}_\texttt{s}\texttt{ y) z}}^{t}$$

Then, we produce the associated resultant:

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

with $\sigma = \{\texttt{x} \mapsto \texttt{x}' : \texttt{x}_\texttt{s}\}$ and $\rho = \{\texttt{app (app x y) z} \mapsto \texttt{dapp x y z}\}$. According to Def. 10, we produce the following equations for the cost criteria of Sect. 3:

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

which represent the cost of performing the above narrowing derivation in the original / residual program. Note that some useful conclusions about the improvement achieved by this residual rule can be easily inferred from the equations. For instance, we can see that all the cost criteria have been halved.

The following result establishes the *local* correctness of the equations generated according to Def. 9, i.e., each single equation is correct w.r.t. the definition of the different cost measures.

**Theorem 1 (local correctness).** *Let $\mathcal{R}$ be an inductively sequential program and $u$ a term such that $C(u) = n$ in $\mathcal{R}$. If there exists an equation $C(s) = C(t)+k$ (associated to a $\lambda$-derivation in $\mathcal{R}$) with $u = \theta(s)$, then $C(\theta(t)) = n - k$.*

In particular, the above result applies to the sets of equations constructed according to Def. 10. However, reasoning about recurrence equations of the above kind is not easy. The problem comes from the laziness of the computation model, since interesting cost criteria are not compositional for non-strict semantics [31]. In particular, the cost of evaluating an expression of the form $f(e)$ will depend on how much function $f$ needs argument $e$. In eager (call-by-value) languages, the cost of $f(e)$ can be obtained by first computing the cost of evaluating $e$ to some normal form $d$ and, then, adding the cost of evaluating $f(d)$. Trivially, this procedure can be used to compute an *upper-bound* for $f(e)$ under a lazy semantics. Nevertheless, we have identified a class of recurrence equations for which we can state a stronger result.

**Definition 11 (closed set of recurrence equations).** *Let $\mathcal{R}$ be a program and $S = \{s_1, \ldots, s_n\}$ a finite set of operation-rooted terms. Let $\mathcal{N}_1, \ldots, \mathcal{N}_n$ be finite (possibly incomplete) needed narrowing trees for $s_i$ in $\mathcal{R}$, $i = 1, \ldots, n$. Let $E$ be the set of recurrence equations associated to the narrowing derivations in $\mathcal{N}_1, \ldots, \mathcal{N}_n$. We say that $E$ is $S$-closed iff for each equation in $E$ of the form:*

$$C(s) = C(t) + k$$

*$t$ is either a constructor term or a constructor instance of some term in $S$.*

The relevance of closed recurrence equations stems from their use to *compute* the cost of a term:

**Theorem 2 (cost computation).** *Let $\mathcal{R}$ be an inductively sequential program and $S = \{s_1, \ldots, s_k\}$ a finite set of operation-rooted terms. Let $\mathcal{N}_1, \ldots, \mathcal{N}_k$ be finite (possibly incomplete) needed narrowing trees (using $\lambda$-derivations) for $s_i$ in $\mathcal{R}$, $i = 1, \ldots, k$. Let $E$ be a set of $S$-closed recurrence equations associated to the $\lambda$-derivations in $\mathcal{N}_1, \ldots, \mathcal{N}_k$. If $t$ is a constructor instance of some term in $S$ and $C(t) = n$, then there is a rewrite sequence $C(t) \to^* n$ using the (oriented) equations in $E$ and the definition of "+".*

Roughly speaking, this result states that, by considering the set of recurrence equations as a rewrite system (implicitly oriented from left to right) and performing additions as usual, we have all the necessary information for computing the associated cost of a term (whenever the cost of this term is defined, i.e., it is narrowable to a constructor term with the empty substitution). Therefore, if the cost of term $t$ is $C(t) = n$ in program $\mathcal{R}$, then there exists a rewrite sequence which *computes* this cost: $C(t) \to^* n$ in $E$.

This result shows that, under appropriate conditions, a finite set $E$ of recurrence equations captures the cost of an infinite set of derivations or terms. For example, $E$ could be used to mechanically compute the cost of a term according to the considered cost criteria. Observe that the rewrite strategy for $E$ is irrelevant, since in $E$ the only defined functions are $C$ and $+$ (any function defined in the original and residual programs plays now the role of a constructor), and $+$ is strict in its both arguments.

Theorem 2 becomes useful when the sets of recurrence equations associated to a concrete partial evaluation are closed. Indeed, the characterization of closed recurrence equations is related to the "closedness" condition employed in some partial evaluation methods. In particular, given a partial evaluation $\mathcal{R}'$ of $\mathcal{R}$ w.r.t. $S$ (under $\rho$), we compute a set of recurrence equations $E = E_{\mathcal{R}} \cup E_{\mathcal{R}'}$ (according to Def. 10), where $E_{\mathcal{R}}$ (resp. $E_{\mathcal{R}'}$) is the subset of recurrence equations associated to program $\mathcal{R}$ (resp. $\mathcal{R}'$). Then, Theorem 2 can be applied whenever $E_{\mathcal{R}}$ is $S$-closed (and, thus, $E_{\mathcal{R}'}$ is $\rho(S)$-closed). Note that this is always ensured if one considers the *perfect* ("$\alpha$-identical") closedness test of [35] or the *basic* notion of closedness of [5] during partial evaluation. However, this property is not guaranteed when stronger notions of closedness are considered during partial evaluation (e.g., the recursive closedness of [5]) or when some partitioning techniques are used (as in conjunctive partial deduction [23]).

According to Def. 11, the equations of Example 6 are not closed due to calls like $\mathcal{S}(\texttt{x : app (app x}_\texttt{s}\texttt{ y) z})$. However, this is not a problem. We can simplify constructor-rooted calls using the following (straightforward) properties:

$$C(x) = 0, \text{ for all } x \in \mathcal{V} \quad (1)$$
$$C(t) = C(t_1) + \ldots + C(t_n), \text{ if } t = c(t_1, \ldots, t_n),\ c \in \mathcal{C},\ n \geq 0 \quad (2)$$

This amounts to say that the cost of reducing constructor constants and variables is zero. Furthermore, the cost to evaluate a constructor-rooted term is the total cost to evaluate all its arguments. In the following, we assume that recurrence equations are possibly simplified using the above properties.

## 5 Usefulness of Recurrence Equations

The previous section presents a formal approach for assessing the effectiveness of partial evaluation. Here, we illustrate its usefulness by showing several possible applications of our recurrence equations.

### 5.1 Recurrence Equations over Natural Numbers

Our recurrence equations provide a formal characterization of the computational cost of executing a program w.r.t. a class of goals (those which are constructor instances of the left-hand sides). Therefore, as a first approach to analyze the improvement achieved by a concrete partial evaluation, one can *solve* the recurrence equations associated to both the original and partially evaluated programs and determine the speedup. In general, it is hard to find explicit solutions of these equations. Nevertheless, we can use a *size* function that maps the arguments of an expression to natural numbers. In some cases, using this function one can transform a cost $C$ over terms into a cost $T$ over natural numbers. Intuitively, for each recurrence equation

$$C(\sigma(s)) = C(t) + k$$

we define an associated equation

$$T_s(n_1, \ldots, n_i) = T_t(m_1, \ldots, m_j) + k$$

where $n_1, \ldots, n_i$ is a sequence of natural numbers representing the sizes of arguments $\sigma(x_1), \ldots, \sigma(x_i)$, with $x_1, \ldots, x_i$ the distinct variables of $s$. Similarly, $m_1, \ldots, m_j$ denote the sizes of the different variables in $t$. Note that the subscript $s$ of $T_s$ is only a device to uniquely identify the recurrence equations associated to term $s$.

*Example 7.* Consider the following recurrence equations defining $\mathcal{S}$:

$$\begin{array}{lcll}
\mathcal{S}(\sigma(\texttt{app (app x y) z})) & = & \mathcal{S}(\texttt{app y z}) + 1 & \text{with } \sigma = \{\texttt{x} \mapsto \texttt{[]}\} \\
\mathcal{S}(\sigma(\texttt{app (app x y) z})) & = & \mathcal{S}(\texttt{app (app x}_\texttt{s}\texttt{ y) z}) + 2 & \text{with } \sigma = \{\texttt{x} \mapsto \texttt{x}' : \texttt{x}_\texttt{s}\} \\
\mathcal{S}(\sigma(\texttt{app x y})) & = & 1 & \text{with } \sigma = \{\texttt{x} \mapsto \texttt{[]}\} \\
\mathcal{S}(\sigma(\texttt{app x y})) & = & \mathcal{S}(\texttt{app x}_\texttt{s}\texttt{ y}) + 1 & \text{with } \sigma = \{\texttt{x} \mapsto \texttt{x}' : \texttt{x}_\texttt{s}\}
\end{array}$$

We can transform them into the following standard recurrence equations over natural numbers:

$$\begin{array}{lcll}
\mathtt{T}_1(0, \mathtt{n}_2, \mathtt{n}_3) & = & \mathtt{T}_2(\mathtt{n}_2, \mathtt{n}_3) + 1 & \\
\mathtt{T}_1(\mathtt{n}_1, \mathtt{n}_2, \mathtt{n}_3) & = & \mathtt{T}_1(\mathtt{n}_1 - 1, \mathtt{n}_2, \mathtt{n}_3) + 2 & \mathtt{n}_1 > 0 \\
\mathtt{T}_2(0, \mathtt{n}_3) & = & 1 & \\
\mathtt{T}_2(\mathtt{n}_2, \mathtt{n}_3) & = & \mathtt{T}_2(\mathtt{n}_2 - 1, \mathtt{n}_3) + 1 & \mathtt{n}_2 > 0
\end{array}$$

where $\mathtt{n_1}$, $\mathtt{n_2}$ and $\mathtt{n_3}$ denote the length of the corresponding lists $\sigma(\mathtt{x})$, $\sigma(\mathtt{y})$ and $\sigma(\mathtt{z})$, respectively. Here, $\mathtt{T_1}$ stands for $\mathtt{T_{app\ (app\ (x\ y)\ z)}}$ and $\mathtt{T_2}$ for $\mathtt{T_{app\ x\ y}}$. The *explicit solutions* of these equations are the functions:

$$\begin{aligned} \mathtt{T_1(n_1, n_2, n_3)} &= 2\ \mathtt{n_1} + \mathtt{n_2} + 2 \\ \mathtt{T_2(n_2, n_3)} &= \mathtt{n_2} + 1 \end{aligned}$$

Generalizing and formalizing a useful notion of size does not seem to ease the understanding or manipulation of recurrence equations because one must carefully distinguish different occurrences of a same constructor. In practice, the occurrences to be counted in sizing a constructor term depend on specific computations. For example, if x is a list of lists, only the “top-level” occurrences of the constructors of list x affect to the length of the evaluation of app x y. The number of occurrences of constructors in an element of x is irrelevant. However, these additional occurrences should be counted in sizing an argument of operation flatten defined below:

```
flatten []              → []
flatten ([] : y)        → flatten y
flatten ((x1 : xs) : y) → x1 : flatten (xs : y)
```

Furthermore, a solution of arbitrary recurrence equations does not always exist. In particular, *non-linear* recurrence equations, which might arise in some cases, do not always have a mathematical explicit solution (i.e., a solution in terms of some size-measure of the arguments).

Nevertheless, in the following we present some alternative approaches. Basically, we are interested in computing the improvement achieved by partial evaluation. Therefore, it may suffice to compute a speedup interval from the corresponding sets of equations, rather than their exact solutions.

## 5.2 Bounds for the Effectiveness of Partial Evaluation

In this section, we are concerned with the estimation of the speedup (or slowdown) produced by partial evaluation from the associated recurrence equations.

Let us denote by $|s, \mathcal{P}|$ the execution time of evaluating a term $s$ in the program $\mathcal{P}$. Consider a program $\mathcal{R}$, a set of terms $S$, and an independent renaming $\rho$ for $S$. Let $\mathcal{R}_S$ be a partial evaluation of $\mathcal{R}$ w.r.t. $S$ (under $\rho$). Then, for a given term $t$, the speedup achieved by partial evaluation is:

$$\frac{|t, \mathcal{R}|}{|ren_\rho(t), \mathcal{R}_S|}$$

Following [8], we say that partial evaluation accomplishes *linear speedup* on $\mathcal{R}$ if for all $S$ there exists a constant $k$ such that for all term $t$

$$k \leq \frac{|t, \mathcal{R}|}{|ren_\rho(t), \mathcal{R}_S|}$$

Let, for each $S$, $k_S$ be the least upper bound of the possible values for $k$. We call $k_S$ the *speedup on $\mathcal{R}$ for $S$*. Jones [20] posed as an open question the possibility of accomplishing superlinear speedups; equivalently: "does there exist a set $S$ for which $k_S$ is not defined?".

It is well-known that, in general, partial evaluation cannot accomplish superlinear speedups; intuitively, the assumption that partial evaluation terminates can be used to place a bound on $k_S$ (see, e.g., [8]). Only a *constant* speedup is usually achieved, i.e., the complexity of the original and partially evaluated programs differs by a constant factor (or, equivalently, the worst-case complexity —"big O" notation— is the same); see, for instance, [7, 8, 21] for traditional partial evaluation and [34] for positive supercompilation. This is also true in partial deduction if the same execution model is also used for performing computations during partial evaluation. Of course, if one uses a different computational mechanism at partial evaluation time, superlinear speedup becomes possible. For instance, one can use call-by-name evaluation at partial evaluation time when specializing call-by-value functional languages, or a refined selection rule when specializing Prolog programs (see, e.g., the discussion in [7]).

In our case, where we use the same mechanism both for execution and for partial evaluation, it is obvious from Def. 10 that the recurrence equations associated to the original and residual programs have exactly the same structure, i.e., they are identical except for the renaming of terms and the associated costs. Hence, it is straightforward to conclude that narrowing-driven partial evaluation cannot achieve superlinear speedups, as well.

An important observation is that the overall speedup of a program is determined by the speedups of loops, since sufficiently long runs will consume most of the execution time inside loops. In our method, loops are represented by recurrence equations. Since the recurrence equations associated to the original and residual programs have the same structure, as justified in the above paragraph, they constitute by themselves a useful aid to the user for determining the speedup (or slowdown) associated to each loop. Moreover, this information is inexpensive to obtain since the recurrence equations can be generated from the same partial derivations used to produce residual rules.

In principle, we can easily modify existing partial evaluators for functional logic programs [2, 4] to provide rules decorated with the associated cost improvement. Each resultant rule can be augmented with a pair of integers, $(k, k')$, for each cost criterion. This pair describes a cost variation (according to Def. 10) of the resultant rule in the original and in the residual program.

For instance, given the partial derivation of Example 6, we produce the (decorated) residual rule:

$$\texttt{dapp}\ (\texttt{x}' : \texttt{x}_\texttt{s})\ \texttt{y}\ \texttt{z}\ \rightarrow\ \texttt{x}' : \texttt{dapp}\ \texttt{x}_\texttt{s}\ \texttt{y}\ \texttt{z} \qquad /\!*\ \{(2,1),(4,2),(2,1)\}\ */$$

From this information, we immediately see that all cost criteria have been improved and, moreover, we can quantify this improvement.

However, as residual programs grow larger, it becomes more difficult to estimate the effectiveness of a particular partial evaluation from the decorated rules. In this case, it would be valuable to design an automatic speedup analysis tool to determine (at least in some cases) the improvement achieved by the whole program w.r.t. each cost criterion. For instance, we could define a simple speedup analysis along the lines of [8]. For this purpose, it suffices to consider a speedup interval $\langle l, u \rangle$ where $l$ and $u$ are, respectively, the smallest and largest ratios $k'/k$ among all the computed recurrence equations.

## 6 Related Work

A considerable effort has been devoted to reason about the complexity of imperative programs (see, e.g., [1]). However, relatively little attention has been paid to the development of methods for reasoning about the computational cost of declarative programs. For logic programs, [13] introduces a method for the (semi-)automatic analysis of the worst-case cost of a large class of logic programs, including nondeterminism and the generation of multiple solutions via backtracking. Regarding eager (call-by-value) functional programs, [24] describes a general approach for time-bound analysis of programs. Essentially, it is based on the construction of time-bound functions which mimic the original functions but compute the associated cost of evaluations. The techniques of [24] cannot be easily adapted to lazy (call-by-name) functional languages since cost criteria are not usually compositional. A similar approach can be found in [30], where a *step-counting* version of a functional program is produced automatically. This version, when called with the same arguments as the original program, returns the computation time for the original program. In a second phase, a *time bound* function (or worst-case complexity) is expressed as an abstract interpretation of the step-counting version. Rosendahl's work [30] is defined for (first-order) call-by-value functional languages, although it contains some hints about how to adapt the method to different languages. We note that, in contrast to [24] and [30], our (cost) recurrence equations are tightly associated to the partial evaluation process and, thus, they allow us to assess more easily the effectiveness achieved by the partial evaluation transformation. On the other hand, [31] develops a theory of cost equivalence to reason about the cost of lazy functional programs. Essentially, [31] introduces a set of *time rules* extracted from a suitable operational semantics, together with some equivalence laws. The aim of this calculus is to reveal enough of the "algorithmic structure" of operationally opaque lazy functional programs to permit the use of more traditional techniques developed in the context of imperative programs.

None of the above references apply the introduced analyses to predict the improvement achieved by partial evaluation techniques. Indeed, we found very little work directed to the formal study of the cost variation due to partial evaluation techniques. For instance, [29] introduces a type system to formally express when a partial evaluator is better than another, i.e., when a residual program is more *efficient* than another. The aim of [7] is on the definition of a general framework to study the effects of several unfolding-based transformations over logic programs. The framework is applied to partial evaluation, but the considered measures are very simple (e.g., unification is not taken into account). A well-known technique appears in [8] and [21, Chapter 6]. They introduce a simple *speedup analysis* which predicts a relative interval of the speedup achieved by a partial evaluation. Finally, [34, Chapter 11] presents a theoretical study of the efficiency of residual programs by positive supercompilation, a partial evaluation technique closely related to narrowing-driven partial evaluation. It proves that the number of steps is not improved by positive supercompilation alone (a post-unfolding phase is necessary). This would be also true in our context if resultants were constructed only from one-step derivations. However, as discussed in [34], this does not mean that the process is useless, since intermediate data structures are frequently eliminated.

All the cited references study the effectiveness of partial evaluation by taking into account only the number of steps in evaluations. Although this is an important measure, we think that other criteria should also be considered (as in traditional experimental profiling approaches [33]). The discussion in [34] (see above) as well as situations like that in Example 1 justify this position.

## 7 Conclusions and Future Work

To the best of our knowledge, this is the first attempt to formally measure the effectiveness of partial evaluation with cost criteria different from the number of evaluation steps. Our characterization of cost enables us to estimate the effectiveness of a partial evaluation in a precise framework. We also provide an automatic method to infer some recurrence equations which help us to reason about the improvement achieved. The combination of these contributions helps us to reconcile theoretical results and experimental measures. Although the introduced notions and techniques are specialized to narrowing-driven partial evaluation, they could be adapted to other related partial evaluation methods (e.g., positive supercompilation [35] or partial deduction [25]).

There are several possible directions for further research. On the theoretical side, we plan to study several properties of partial evaluation within the formal framework presented so far. Two well-known facts motivate our interest: (1) partial evaluation techniques cannot, in general, increase the number of steps required to perform a particular computation (under a call-by-name evaluation model, see [32]); and (2) partial evaluation sometimes degrades program efficiency. By experimenting with a preliminary implementation of the analysis outlined in the present paper, we discovered that some cost criteria can be actually degraded (producing a slowdown in the residual program). The formal study of the conditions under which an improvement is guaranteed for each cost criteria is the subject of ongoing research. On the practical side, we plan to develop an analytical tool for estimating the improvements achieved by residual programs.

## Acknowledgements

We gratefully acknowledge the anonymous referees and the participants of LOPSTR'2000 for their comments on this paper. We are also grateful to Mark Jones for helping us to benchmark Hugs programs. Part of this research was done while Elvira Albert and Germán Vidal were visiting the Department of Computer Science at Portland State University. They gratefully acknowledge this department's hospitality.

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

## References

1. A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA, 1974.
2. E. Albert, M. Alpuente, M. Falaschi, and G. Vidal. INDY User's Manual. Technical Report DSIC-II/12/98, UPV, 1998.
3. E. Albert, S. Antoy, and G. Vidal. A Formal Approach to Reasoning about the Effectiveness of Partial Evaluation. Technical Report DSIC, UPV, 2000. Available from URL: `http://www.dsic.upv.es/users/elp/papers.html`.
4. E. Albert, M. Hanus, and G. Vidal. Using an Abstract Representation to Specialize Functional Logic Programs. In *Proc. of the 7th Int'l Conf. on Logic for Programming and Automated Reasoning (LPAR'2000)*, pages 381–398. Springer LNAI 1955, 2000.
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. M. Alpuente, M. Hanus, S. Lucas, and G. Vidal. Specialization of Functional Logic Programs Based on Needed Narrowing. *ACM Sigplan Notices*, 34(9):273–283, 1999.
7. T. Amtoft. Properties of Unfolding-based Meta-level Systems. In *Proc. of PEPM'91*, pages 243–254. ACM Press, 1991.
8. L.O. Andersen and C.K. Gomard. Speedup Analysis in Partial Evaluation: Preliminary Results. In *Proc. of PEPM'92*, pages 1–7. Yale University, 1992.
9. S. Antoy. Definitional trees. In *Proc. of the 3rd Int'l Conference on Algebraic and Logic Programming (ALP'92)*, pages 143–157. Springer LNCS 632, 1992.
10. S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. *Journal of the ACM*, 47(4):776–822, 2000.
11. F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.
12. C. Consel and O. Danvy. Tutorial notes on Partial Evaluation. In *Proc. of the Annual Symp. on Principles of Programming Languages (POPL'93)*, pages 493–501. ACM, New York, 1993.
13. S.K. Debray and N.W. Lin. Cost Analysis of Logic Programs. *ACM Transactions on Programming Languages and Systems*, 15(5):826–975, 1993.
14. J. Gallagher. Tutorial on Specialisation of Logic Programs. In *Proc. of PEPM'93*, pages 88–98. ACM, New York, 1993.
15. E. Giovannetti, G. Levi, C. Moiso, and C. Palamidessi. Kernel Leaf: A Logic plus Functional Language. *Journal of Computer and System Sciences*, 42:363–377, 1991.
16. M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, 19&20:583–628, 1994.
17. M. Hanus (ed.). Curry: An Integrated Functional Logic Language. Available at `http://www-i2.informatik.rwth-aachen.de/~hanus/curry`, 2000.

18. G. Huet and J.J. Lévy. Computations in orthogonal rewriting systems, Part I + II. In J.L. Lassez and G.D. Plotkin, editors, *Computational Logic – Essays in Honor of Alan Robinson*, pages 395–443, 1992.
19. M.P. Jones and A. Reid. The Hugs 98 User Manual. Available at `http://haskell.cs.yale.edu/hugs/`, 1998.
20. N.D. Jones. Partial Evaluation, Self-Application and Types. In M.S. Paterson, editor, *Proc. of 17th Int'l Colloquium on Automata, Languages and Programming (ICALP'90)*, pages 639–659. Springer LNCS 443, 1990.
21. N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Englewood Cliffs, NJ, 1993.
22. Laura Lafave. *A Constraint-based Partial Evaluator for Functional Logic Programs and its Application*. PhD thesis, University of Bristol, 1999.
23. M. Leuschel, D. De Schreye, and A. de Waal. A Conceptual Embedding of Folding into Partial Deduction: Towards a Maximal Integration. In M. Maher, editor, *Proc. of the Joint Int'l Conf. and Symp. on Logic Programming (JICSLP'96)*, pages 319–332. The MIT Press, Cambridge, MA, 1996.
24. Y. A. Liu and G. Gomez. Automatic Accurate Time-Bound Analysis for High-Level Languages. In *Proc. of ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 31–40. Springer LNCS 1474, 1998.
25. J.W. Lloyd and J.C. Shepherdson. Partial Evaluation in Logic Programming. *Journal of Logic Programming*, 11:217–242, 1991.
26. R. Loogen, F. López-Fraguas, and M. Rodríguez-Artalejo. A Demand Driven Computation Strategy for Lazy Narrowing. In *Proc. of 5th Int'l Symp. on Programming Language Implementation and Logic Programming (PLILP'93)*, pages 184–200. Springer LNCS 714, 1993.
27. F.J. 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.
28. J.J. Moreno-Navarro and M. Rodríguez-Artalejo. Logic Programming with Functions and Predicates: The language Babel. *Journal of Logic Programming*, 12(3):191–224, 1992.
29. F. Nielson. A Formal Type System for Comparing Partial Evaluators. In *Proc. of the Int'l Workshop on Partial Evaluation and Mixed Computation*, pages 349–384. N-H, 1988.
30. M. Rosendahl. Automatic Complexity Analysis. In *Proc. of the Int'l Conf. on Functional Programming Languages and Computer Architecture*, pages 144–156, New York, NY, 1989. ACM.
31. D. Sands. A Naive Time Analysis and its Theory of Cost Equivalence. *Journal of Logic and Computation*, 5(4):495–541, 1995.
32. D. Sands. Total Correctness by Local Improvement in the Transformation of Functional Programs. *ACM Transactions on Programming Languages and Systems*, 18(2):175–234, March 1996.
33. 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.
34. M.H. Sørensen. Turchin's Supercompiler Revisited: An Operational Theory of Positive Information Propagation. Technical Report 94/7, Master's Thesis, DIKU, University of Copenhagen, Denmark, 1994.
35. M.H. Sørensen, R. Glück, and N.D. Jones. A Positive Supercompiler. *Journal of Functional Programming*, 6(6):811–838, 1996.