Cost-Augmented Narrowing-Driven Specialization[*]

Germán Vidal
DSIC, Technical University of Valencia
Camino de Vera s/n
E-46022 Valencia, Spain
gvidal@dsic.upv.es

ABSTRACT

The aim of many program transformers is to improve efficiency while preserving program meaning. Correctness issues have been dealt with extensively. However, very little attention has been paid to formally establish the improvements achieved by these transformers. In this work, we introduce the scheme of a narrowing-driven partial evaluator enhanced with abstract costs. They are "abstract" in the sense that they measure the number of *basic* operations performed during a computation rather than actual execution times. Thus, we have available a setting in which one can discuss the effects of the program transformer in a precise framework and, moreover, to quantify these effects. Our scheme may serve as a basis to develop speedup analyses and cost-guided transformers. An implementation of the cost-augmented specializer has been undertaken, which demonstrates the practicality of our approach.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Optimization*; I.2.2 [**Artificial Intelligence**]: Automatic Programming—*Program transformation*

General Terms

Languages, Performance

1. INTRODUCTION

Program transformation techniques include different methods whose common goal is to derive correct and efficient programs. This paper concerns an automatic program transformation called *partial evaluation* (PE). Basically, a partial evaluator is a source-to-source program transformer which takes a program and *part* of its input data—the so-called *static* data—and returns a specialized version of the original program for the given data. The new, residual program hopefully runs more efficiently than the original program since those computations that depend only on the static data have been performed once and for all at PE time.

While correctness issues have been dealt with extensively in PE, very little effort has been devoted to formally study the efficiency improvements achieved by this technique. In this work, we present a novel approach which combines ideas from PE and (abstract) profiling. In particular, we propose a PE method enhanced with *abstract* costs. They are "abstract" in the sense that they count the number of *basic* operations (e.g., function unfoldings, pattern matchings, etc) performed during a computation rather than actual execution times. The output of the enhanced partial evaluator is a set of residual rules together with the costs of the partial computations which produced such residual rules. This allows us to determine whether an improvement has been achieved and, moreover, to quantify this improvement.

To center the discussion, we apply these ideas to a particular PE framework: the narrowing-driven approach to the PE of functional logic programs [4, 6]. This framework has been recently extended to cope with modern implementations of functional logic programming in [2, 3] by translating source programs into an intermediate form—called *flat* representation [20, 22, 23, 27]—where all the features of these language implementations can be represented at an adequate level of abstraction. A partial evaluator which follows the ideas presented in [2, 3] has been incorporated into the PAKCS [20] compiler for the multi-paradigm language Curry [22], which has become the standard in functional logic programming.

The syntax of functional logic programs is closer to that of pure functional programs (e.g., the syntax of Curry follows mainly that of the lazy functional language Haskell). Despite this, the narrowing-driven PE framework shares more similarities with unification-based methods for the PE of logic programs (also known as *partial deduction* [26]) than to traditional partial evaluators for functional programs based on constant propagation [24]. In narrowing-driven PE, the so-called static/dynamic distinction is hardly present. Indeed, the "known data" are presented in the form of a partially instantiated call. Another significant feature of this framework is its ability to improve programs even when no input data are provided, e.g., when the input to the PE procedure is a function call of the form $f(x_1, \ldots, x_n)$ where all the arguments are variables. In this case, narrowing-driven PE is often able to produce a new, residual function f' which is equivalent to f but more efficient, since all computations

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

that are independent of the values of the input variables can be precomputed in f'. In fact, narrowing-driven PE is able to achieve some form of composition and tupling automatically [6]. These remarks also apply to several PE methods for logic programs (e.g., partial deduction [26] and conjunctive partial deduction [14]) as well as to some (on-line) PE methods for functional programs (like supercompilation [31], generalized partial computation [15], and positive supercompilation [30], the closest to our framework [4, 6]).

As a starting point, we extend the standard semantics for functional logic programs in flat form—the LNT calculus [21]—with abstract costs. Our abstract costs are based on the cost criteria introduced in [1, 11] to measure the cost of functional logic computations: the number of steps, the number of pattern matching operations, and the number of applications. The resulting cost-augmented semantics is similar to the one introduced in [5] for performing source-level abstract profiling. Nevertheless, here we do not consider the attribution of costs to so-called "cost centers" but attribute all costs to a single cost center associated to the entire computation. Thus, our calculus is notably simplified. Basically, the enhanced semantics mimics the LNT calculus but additionally computes the abstract costs attributed to a particular computation. In contrast to [1, 11], we define our cost semantics for *flat programs*. This makes our approach more practically applicable, since actual implementations of functional logic languages use a flat representation as an intermediate language and, thus, realistic costs should be gathered at this level.

Narrowing-driven PE constructs residual rules by means of a *residualizing* variant of the standard semantics: the RLNT calculus [2]. Hence, we also define a cost-augmented version of the RLNT calculus. The relation, in terms of cost, between the standard and the residualizing semantics (augmented with costs) is established. Then, we present the scheme of a narrowing-driven partial evaluator which uses the cost-augmented RLNT calculus to perform computations at PE time. Unlike the original framework, the new partial evaluator returns a set of pairs (r, k), where r is a residual rule and k is the associated cost. By using the cost equivalence between the standard and the residualizing semantics, we can reason about the relation, in terms of cost, between applying the residual rule r and an equivalent computation in the original program. Thus, we can analyze the cost *improvement* associated to a particular specialization.

An implementation of the cost-augmented PE scheme has been undertaken by extending an existing partial evaluator for Curry programs [3]. Preliminary experiments indicate that our approach is both practical and useful. We also sketch the implementation of a simple *speedup analysis* to obtain a global measure of the improvement achieved by a concrete specialization. It is based on detecting the program loops and computing their associated costs (either in the original and residual programs).

The structure of the paper is as follows. Section 2 presents the syntax of the flat representation for programs and Section 3 introduces a cost-augmented semantics for these programs. Then, in Section 4, we define a cost-augmented version of the residualizing semantics used to perform computations at PE time. The overall PE scheme enhanced with cost information is presented in Section 5. Some details of the implemented system, together with some preliminary results, are shown in Section 6. Several related works are discussed in Section 7 before we conclude in Section 8.

$$
\begin{array}{lcll}
\mathcal{R} & ::= & D_1 \ldots D_m & \text{(program)} \\
\mathcal{D} & ::= & f(\overline{x_n}) = e & \text{(rule)} \\
e & ::= & x & \text{(variable)} \\
 & \mid & c(\overline{e_n}) & \text{(constructor)} \\
 & \mid & f(\overline{e_n}) & \text{(function call)} \\
 & \mid & \mathit{case}\ e\ \mathit{of}\ \{\overline{p_n \to e_n}\} & \text{(rigid case)} \\
 & \mid & \mathit{fcase}\ e\ \mathit{of}\ \{\overline{p_n \to e_n}\} & \text{(flexible case)} \\
p & ::= & c(\overline{x_n}) & \text{(flat pattern)}
\end{array}
$$

Figure 1: Syntax of Flat Programs

2. THE FLAT LANGUAGE

Recent proposals for multi-paradigm declarative programming (including features from the functional, logic and concurrent paradigms) usually consider inductively sequential term rewriting systems [9] as *source programs* and a combination of needed narrowing—the counterpart of call-by-name evaluation—and residuation as operational semantics [17]. In actual implementations (e.g., the PAKCS [20] compiler for Curry [22]), programs may also include a number of additional features: calls to external (built-in) functions, concurrent constraints, higher-order functions, overlapping left-hand sides, guarded expressions, etc. In order to ease the compilation of programs as well as to provide a common interface for connecting different tools working on source programs, a *flat representation* for programs has been recently introduced. This representation is based on the formulation of [21] to express pattern-matching by case expressions; the complete flat representation is called FlatCurry [20, 22, 23, 27] and is used as an intermediate language during the compilation of source programs.

To keep things simple, we only present the *core* of the flat representation. Extending the developments in this paper to the remaining features is not difficult and, indeed, the implementation reported in Section 6 covers all the additional features. The syntax is presented in Figure 1, where we write $\overline{o_n}$ to denote the *sequence of objects* $o_1, \ldots, o_n$. We use $x, y, z, \ldots$ for denoting *variables*, $a, b, c, \ldots$ for *constructors*, and $f, g, h, \ldots$ for defined functions or *operations*. A program $\mathcal{R}$ consists of a sequence of function definitions D such that each function is defined by one rule whose left-hand side contains only different variables as parameters. The right-hand side is an expression e composed by variables, constructors, function calls, and case expressions for pattern-matching. The general form of a case expression is:

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

where e is an expression, $c_1, \ldots, c_k$ are different constructors of the type of e, and $e_1, \ldots, e_k$ are expressions (possibly containing nested $(f)\mathit{case}$'s). The variables $\overline{x_{n_i}}$ are local variables which occur only in the corresponding subexpression e_i. The difference between *case* and *fcase* only shows up when the argument e is a free variable: *case* suspends (which corresponds to *residuation*, i.e., pure functional reduction) whereas *fcase* nondeterministically binds this variable to a pattern in a branch of the case expression (which corresponds to narrowing). Note that our functional logic language mainly differs from typical (lazy) functional languages in the presence of flexible case expressions. Functions defined only by *fcase* (resp. *case*) expressions are called *flexible* (resp. *rigid*). Thus, flexible functions act as generators (like predicates in logic programming) and rigid functions act as consumers.

Example 1. Consider the well-known function `app` to concatenate two lists. It can be defined in our flat representation by the following definition:[1]

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

where [ ] denotes the empty list and `x : xs` (alternatively, [x|xs]) a list with first element `x` and tail `xs`.

An automatic transformation from inductively sequential programs [9] to programs using case expressions is introduced in [21].

3. COST-AUGMENTED SEMANTICS

In this section, we present an operational semantics for flat programs enhanced with cost information. The standard semantics for flat programs is defined in terms of the LNT calculus (Lazy Narrowing with definitional Trees [21]). The rules of the LNT calculus basically coincide with the rules depicted in Figure 2 by ignoring cost information.

Our cost-augmented semantics is based on the cost criteria introduced in [1, 11] to measure the cost of functional logic computations. However, as we mentioned before, we define our abstract costs at the level of the flat representation rather than at the level of source programs. The abstract costs that we consider are: the number of function unfoldings, the number of case evaluations (to measure the pattern matching effort), and the number of applications (to measure the number of allocated cells). In fact, the resulting cost-augmented semantics is similar to the one introduced in [5] for performing source-level abstract profiling. There are, though, two main differences:

- Firstly, the attribution of costs to so-called *cost centers* is ignored; here, all costs are attributed to one single cost center representing the entire computation.
- Secondly, there is no abstract cost to measure the number of *nondeterministic* branching points (which result from the evaluation of flexible case expressions with a variable argument). This is justified by the fact that, at PE time, computations are performed with *incomplete* information. Thus, counting the number of branching points would not be fair since, at execution time, this number may vary depending on the degree of instantiation of actual calls (see the discussion in Section 5.1).

The cost-augmented semantics is formalized by the state-transition rules of Figure 2. The *state* consists of a tuple $\langle k, e\rangle$, where k is the accumulated cost and e is an expression. An initial state has the form $\langle K_0, e\rangle$, where K_0 is the *empty* cost and e is the expression to be evaluated. Costs are represented by a set of cost variables $\in \{S, C, A\}$. Thus, S records the number of steps, C the number of case evaluations (or basic pattern matching operations), and A the number of applications. The one-step transition relation $\Rightarrow_\sigma$ is labeled with the substitution σ computed in the step. Let us briefly describe the state-transition rules.

The *hnf* rule can be applied to evaluate expressions in head normal form (i.e., rooted by a constructor symbol). It proceeds by recursively evaluating any argument (e.g., the leftmost one) which is not a *constructor term* (i.e., a term containing only constructor symbols and variables). Note that there is no rule applicable to evaluate constructor terms. In this case, the computation stops *successfully.*

The *case_select* rule simply selects the appropriate branch of a case expression and continues with the evaluation of this branch. This step is labeled with the identity substitution *id*. The current cost is updated by adding one to cost variable C.

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

The *case_eval* rule can be only applied when the argument of the case construct is a function call or another case construct. Then, it tries to evaluate this expression recursively. If an evaluation step is possible, we return the original expression with the argument and the associated cost updated. The step is labeled with the same substitution computed from the evaluation of the case argument.

Finally, the *fun_eval* rule performs the unfolding of a function call. As in proof procedures for logic programming, we assume that we take a program rule with fresh variables in each such evaluation step. The current cost k is updated by adding one to cost variable S and by adding *size*(e) to cost variable A, where e is the right-hand side of the applied rule. Function *size* counts the number of applications in an expression and it is useful to quantify memory usage. Following [11], given an expression e, a call to *size*(e) returns the total number of occurrences of n-ary symbols, with $n > 0$, in e, plus their arities. This is coherent, e.g., with the implementation described in [10]. Within the flat syntax, we consider that a case expression with n branches has arity $2n + 1$. For instance, the case expression in the right-hand side of function `app` (Example 1) has arity 5; indeed, it could be written in prefix notation as follows: `case(x, [ ], y, z : zs, x : app zs y)`.

Arbitrary *derivations* are denoted by

$$\langle K_0, e\rangle \Rightarrow^*_\sigma \langle k, e'\rangle$$

which is a shorthand for the sequence of steps

$$\langle K_0, e\rangle \Rightarrow_{\sigma_1} \ldots \Rightarrow_{\sigma_n} \langle k, e'\rangle$$

with $\sigma = \sigma_n \circ \cdots \circ \sigma_1$ (if $n = 0$ then $\sigma = id$). We say that a derivation $\langle K_0, e\rangle \Rightarrow^*_\sigma \langle k, e'\rangle$ is *successful* when e' contains no defined function calls (i.e., it is a constructor term). Then, we say that e evaluates to e' with computed answer σ and associated cost k.

Example 2. Consider the function `app` of Example 1. Given the initial state

$$\langle \mathtt{K_0},\ \mathtt{app}\ [1,2|\mathtt{x}]\ [3]\rangle$$

[1]Although we consider in this work a first-order representation for programs, we use a curried notation in concrete examples (as usual in functional languages).

rule	$\langle$cost,	expression$\rangle$	$\Rightarrow$	$\langle$cost,	expression$\rangle$
hnf	$\langle k,$	$c(e_1,\ldots,e_i,\ldots,e_n)\rangle$	$\Rightarrow_\sigma$	$\langle k',$	$\sigma(c(e_1,\ldots,e'_i,\ldots,e_n))\rangle$ if e_i is not a constructor term and $\langle k,e_i\rangle \;\Rightarrow_\sigma\; \langle k',e'_i\rangle$
case_select	$\langle k,$	$(f)case\ c(\overline{e_n})\ of\ \{\overline{p_m \to e'_m}\}\rangle$	$\Rightarrow_{id}$	$\langle k',$	$\sigma(e'_i)\rangle$ if $p_i = c(\overline{x_n})$, $\sigma = \{\overline{x_n \mapsto e_n}\}$, and $k' = k[C \leftarrow C+1]$
case_guess	$\langle k,$	$fcase\ x\ of\ \{\overline{p_m \to e_m}\}\rangle$	$\Rightarrow_\sigma$	$\langle k',$	$\sigma(e_i)\rangle$ if $\sigma = \{x \mapsto p_i\}$, and $k' = k[C \leftarrow C+1]$
case_eval	$\langle k,$	$(f)case\ e\ of\ \{\overline{p_m \to e_m}\}\rangle$	$\Rightarrow_\sigma$	$\langle k',$	$(f)case\ e'\ of\ \{\overline{p_m \to e_m}\}\rangle$ if e is a nonvariable expression which is not rooted by a constructor symbol and $\langle k,e\rangle \;\Rightarrow_\sigma\; \langle k',e'\rangle$
fun_eval	$\langle k,$	$f(\overline{e_n})\rangle$	$\Rightarrow_{id}$	$\langle k',$	$\sigma(e)\rangle$ if $f(\overline{x_n}) = e \in \mathcal{R}$ is a rule with fresh variables, $\sigma = \{\overline{x_n \mapsto e_n}\}$, and $k' = k[S \leftarrow S+1, A \leftarrow A + size(e)]$

Figure 2: Cost-augmented LNT calculus

the enhanced LNT calculus computes, among others, the value $[1,2,3]$, with computed answer $\{\mathtt{x} \mapsto [\,]\}$ and associated costs $S = 3$, $C = 3$, and $A = 45$ (with $size(e) = 15$, where e is the right-hand side of function `app`).

Trivially, the cost-augmented semantics of Figure 2 is a conservative extension of the original LNT calculus. The search space and the computed values and answers never depend on the current costs, so the new semantics does not change the results produced by a computation. To be precise, given a cost-augmented LNT derivation $\langle K_0, e\rangle \Rightarrow^*_\sigma \langle k, e'\rangle$, we have that $e \Rightarrow^*_\sigma e'$ is a standard LNT derivation, and vice versa.

4. COST-AUGMENTED PE

The narrowing-driven PE scheme was originally designed to use the standard operational semantics to perform computations at PE time [4, 6]. Within this scheme, residual rules are constructed from partial computations as follows. Given an expression e and a (possibly incomplete) standard evaluation $e \;\Rightarrow^*_\sigma\; e'$ computing the substitution σ, we derive a residual rule—a *resultant*—of the form: $\sigma(e) \;=\; e'$. However, the backpropagation of bindings to the left-hand sides of residual rules (i.e., the instantiation of e by σ), introduces several problems [2]; for instance, the left-hand sides of resultants may become instantiated, which is not allowed by the syntax of Figure 1 (where only variable arguments are accepted in the left-hand sides of the rules). Therefore, [2, 3] proposed the use of a non-standard, *residualizing* semantics to perform partial computations. The main particularity of the new semantics, the RLNT calculus (which stands for Residualizing LNT calculus), is that bindings are not propagated backwards but represented by *residual case expressions with a variable argument.*

For instance, given the expression:

$$\mathtt{fcase\ x\ of\ \{0 \to 0;\ 1 : h\ x\ y\}}$$

the standard semantics (nondeterministically) computes either "0" (with associated binding $\{\mathtt{x} \mapsto \mathtt{0}\}$) or "`h 1 y`" (with associated binding $\{\mathtt{x} \mapsto \mathtt{1}\}$). On the other hand, the residualizing semantics leaves the expression unchanged and proceeds with the evaluation of "`h 1 y`." Thus, the residualizing semantics is deterministic.

In general, our cost-augmented semantics may produce infinite derivations. At PE time, however, we are only interested in performing *finite* (possibly incomplete) derivations. Thus, some mechanism to ensure termination is required. In our case, this is the task of the *unfolding rule*, as we will see in Section 5.

The enhanced semantics is based on the RLNT calculus of [2, 3] extended to compute abstract costs. Let us describe the state transition rules of the enhanced calculus (depicted in Figure 3). Let us note that the same considerations of Section 3 about states, derivations, etc., apply here. The only exception is that the relation $\Rightarrow$ is not labeled with a substitution, since the new calculus does not compute bindings, but encode them by means of residual case expressions.

Firstly, note that there is no rule to evaluate terms in head normal form. This is necessary not to encumber the formulation of the calculus; otherwise, additional rules to propagate bindings—in the form of case expressions with a variable argument—and to accumulate costs between the arguments of a head normal form become necessary. Let us mention, though, that this is not a real restriction, since the calculus is applied iteratively and, thus, the arguments of a head normal form will be evaluated in the next iteration of the algorithm (see the PE procedure in Section 5).

While the *case_select* rule remains unchanged, the original *case_guess* rule is replaced by two new rules. Rule *case_guess1* proceeds as in the standard semantics, but residualizes the case structure—the "bindings"—rather than performing a nondeterministic branching. Moreover, it applies the corresponding substitutions to the different alternatives of the case expression in order to propagate bindings forward in the computation. As for the cost, we use the construction $alt(\overline{k_m})$ to denote an *alternative* between different branches. For instance, given a state of the form:

$$\langle alt(k_1,k_2),\ case\ x\ of\ \{p_1 \to e_1;\ p_2 \to e_2\}\rangle$$

k_1 denotes the costs attributed to e_1 while k_2 denotes the costs attributed to e_2. Of course, we could follow a simpler

rule	⟨cost,	expression⟩	⇒ ⟨cost, expression⟩
case_select	$\langle k,$	$(f)\mathit{case}\ c(\overline{e_n})\ \mathit{of}\ \{\overline{p_m \to e'_m}\}\rangle$	$\Rightarrow \langle k', \quad \sigma(e'_i)\rangle$ if $p_i = c(\overline{x_n})$, $\sigma = \{\overline{x_n \mapsto e_n}\}$, and $k' = k[C \leftarrow C+1]$
case_guess1	$\langle k,$	$(f)\mathit{case}\ x\ \mathit{of}\ \{\overline{p_m \to e_m}\}\rangle$	$\Rightarrow \langle k', \quad (f)\mathit{case}\ x\ \mathit{of}\ \{\overline{p_m \to \sigma_m(e_m)}\}\rangle$ if $k \neq \mathit{alt}(\ldots)$, $\sigma_i = \{x \mapsto p_i\}$, $k_i = k[C \leftarrow C+1]$, for all $i = 1, \ldots, m$, and $k' = \mathit{alt}(\overline{k_m})$
case_guess2	$\langle \mathit{alt}(\overline{k_m}),$	$(f)\mathit{case}\ x\ \mathit{of}\ \{\overline{p_m \to e_m}\}\rangle$	$\Rightarrow \langle k', \quad (f)\mathit{case}\ x\ \mathit{of}\ \{\overline{p_m \to e'_m}\}\rangle$ if $\langle k_i, e_i\rangle \Rightarrow \langle k'_i, e'_i\rangle$, for some $i \in \{1, \ldots, m\}$, $k' = \mathit{alt}(\overline{k_{i-1}}, k'_i, k_{i+1}, \ldots, k_m)$, and $e'_j = e_j$ for $j \neq i$
case_of_case	$\langle k,$	$(f)\mathit{case}\ ((f)\mathit{case}\ x\ \mathit{of}\ \{\overline{p'_n \to e'_n}\})\ \mathit{of}\ \{\overline{p_m \to e'_m}\}\rangle$	$\Rightarrow \langle k, \quad (f)\mathit{case}\ x\ \mathit{of}\ \{\overline{p'_n \to (f)\mathit{case}\ e'_n\ \mathit{of}\ \{\overline{p_m \to e_m}\}}\}\rangle$
case_eval	$\langle k,$	$(f)\mathit{case}\ e\ \mathit{of}\ \{\overline{p_m \to e_m}\}\rangle$	$\Rightarrow \langle k', \quad (f)\mathit{case}\ e'\ \mathit{of}\ \{\overline{p_m \to e_m}\}\rangle$ if e is neither a variable, a constructor-rooted term, nor of the form $(f)\mathit{case}\ x\ \mathit{of}\ \{\ldots\}$, and $\langle k, e\rangle \Rightarrow \langle k', e'\rangle$
fun_eval	$\langle k,$	$f(\overline{e_n})\rangle$	$\Rightarrow \langle k', \quad \sigma(e)\rangle$ if $f(\overline{x_n}) = e \in \mathcal{R}$ is a rule with fresh variables, $\sigma = \{\overline{x_n \mapsto e_n}\}$, and $k' = k[S \leftarrow S+1, A \leftarrow A + \mathit{size}(e)]$

Figure 3: Cost-augmented RLNT calculus

strategy and mark each branch with the current cost, i.e.,

$$\mathit{case}\ x\ \mathit{of}\ \{p_1 \to \langle k_1, e_1\rangle;\ p_2 \to \langle k_2, e_2\rangle\}$$

However, this will only postpone the creation of *alt* constructs since, at the end, we need to produce *standard* expressions with no cost information (in order to produce *executable* residual programs!). Let us notice that we assign the same cost increment to both the *case_select* and *case_guess1* rules. Basically, the reason is that the application of rule *case_guess1* during PE may correspond to the application of rule *case_select* at execution time, since runtime expressions are usually more instantiated.

After one application of the previous rule, the new rule *case_guess2* applies. It is used to recursively evaluate the different branches of a case expression with a variable argument. Depending on the selection strategy, we can simulate either a depth-first or a breadth-first inspection of the search space.

Due to the residualization of case structures with a variable argument, a new rule to evaluate nested case expressions (where the inner case has a variable argument) becomes necessary. For this purpose, we introduce the rule *case_of_case*, which moves the outer case inside the branches of the inner one. Similar rules can be found, e.g., in Wadler's deforestation [32] and in the driving mechanism [30]. Note that the current costs remain the same, since no expression is reduced and the order of evaluation is not changed. Rigorously speaking, this rule can be expanded into four rules (with the different combinations for *case* and *fcase* expressions), but we keep the above (less formal) presentation for simplicity. Observe that the outer case expression may be duplicated several times, but each copy is now (possibly) scrutinizing a known value, and so the *case_select* rule can be applied to eliminate some case constructs.

The remaining rules, *case_eval* and *fun_eval*, are not modified, except for the fact that the former rule can be only applied when the argument of the case construct is a function call or another case construct *with a nonvariable argument* (since the case where the inner case has a variable argument is dealt with in the *case_of_case* rule). Of course, the application of the *fun_eval* rule may duplicate computations under a graph-based implementation of narrowing. In this case, an additional constraint to fire this rule should be the linearity of the right-hand side e. Although this is not the case in the calculus of Figure 2, several functional logic implementations (e.g., the PAKCS environment [20]) allow the sharing of variables and, thus, we should impose this restriction in practical partial evaluators.

Example 3. Consider again the function `app` of Example 1. Given the initial state

$$\langle \mathtt{K_0},\ \mathtt{app\ (app\ x\ y)\ z}\rangle$$

the enhanced RLNT calculus computes, for instance, the following incomplete derivation:

$$\langle \mathtt{K_0},\ \mathtt{app\ (app\ x\ y)\ z}\rangle$$

$$\Rightarrow \quad (\mathit{fun\_eval})$$

$$\begin{array}{l} \langle\{S \mapsto 1, C \mapsto 0, A \mapsto 15\}, \\ \quad \mathtt{fcase\ (app\ x\ y)\ of\ \{\ [\,]} \qquad\quad \to\ \mathtt{z}\ ; \\ \qquad\qquad\qquad\qquad\quad \mathtt{(y' : ys')}\ \to\ \mathtt{y' : app\ ys'\ z\ \}}\ \rangle \end{array}$$

$$\Rightarrow \quad (\mathit{case\_eval/fun\_eval})$$

$$\begin{array}{l} \langle\{S \mapsto 2, C \mapsto 0, A \mapsto 30\}, \\ \quad \mathtt{fcase\ (fcase\ x\ of\ \{[\,] \to y; (x' : xs') \to x' : app\ xs'\ y\})} \\ \qquad\quad \mathtt{of\ \{\ \ [\,]} \qquad\quad \to\ \ \mathtt{z}\ ; \\ \qquad\qquad\quad \mathtt{(y' : ys')}\ \ \to\ \ \mathtt{y' : app\ ys'\ z\ \}}\ \rangle \end{array}$$

$\Rightarrow \quad (\mathit{case\_of\_case})$

$\langle \{S \mapsto 2, C \mapsto 0, A \mapsto 30\},$
```
 fcase x of {
      [ ]        →   fcase y of {
                         [] → z;
                         (y′ : ys′) → y′ : app ys′ z } ;
      (x′ : xs′)  →   fcase (x′ : app xs′ y) of {
                         [ ] → z;
                         (y′ : ys′) → y′ : app ys′ z } } ⟩
```

$\Rightarrow \quad (\mathit{case\_guess1})$

$\langle \mathit{alt}(\{S \mapsto 2, C \mapsto 1, A \mapsto 30\}, \{S \mapsto 2, C \mapsto 1, A \mapsto 30\}),$
```
 fcase x of {
      [ ]        →   fcase y of {
                         [] → z;
                         (y′ : ys′) → y′ : app ys′ z } ;
      (x′ : xs′)  →   fcase (x′ : app xs′ y) of {
                         [ ] → z;
                         (y′ : ys′) → y′ : app ys′ z } } ⟩
```

$\Rightarrow \quad (\mathit{case\_guess2/case\_select})$

$\langle \mathit{alt}(\{S \mapsto 2, C \mapsto 1, A \mapsto 30\}, \{S \mapsto 2, C \mapsto 2, A \mapsto 30\}),$
```
 fcase x of {
      [ ]        →   fcase y of {
                         [] → z;
                         (y′ : ys′) → y′ : app ys′ z } ;
      (x′ : xs′)  →   x′ : app (app xs′ y) z } ⟩
```

The following result establishes a precise equivalence between the cost-augmented standard semantics (Fig. 2) and its residualizing version (Fig. 3). First, we need the auxiliary state transition relation $\hookrightarrow$, which is defined by the transition rule:

$$\langle \mathit{alt}(\overline{k_m}),\ \mathit{fcase}\ x\ \mathit{of}\ \{\overline{p_m \to e_m}\}\rangle \ \hookrightarrow_\sigma\ \langle k_i, e_i\rangle$$

where $\sigma = \{x \mapsto p_i\}$ for some $i \in \{1, \ldots, m\}$. This nondeterministic relation is needed to extract the bindings encoded by residualized case expressions.

THEOREM 1. *Let e be an expression, e' a head normal form, and $\mathcal{R}$ a program in the flat representation. For each cost-augmented LNT derivation $\langle K_0, e\rangle \Rightarrow^*_\sigma \langle k', e'\rangle$ in $\mathcal{R}$, there exists a cost-augmented RLNT derivation $\langle K_0, e\rangle \Rightarrow^* \langle k'', e''\rangle$ in $\mathcal{R}$ such that $\langle k'', e''\rangle \hookrightarrow^*_\sigma \langle k', e'\rangle$, and vice versa.*

Informally speaking, for each cost-augmented LNT derivation from e to a head normal form e', computing σ and k', there is a cost-augmented RLNT derivation from e to some e'' in which the computed substitution σ' is encoded in e'' by case expressions and can be obtained by a (finite) sequence of $\hookrightarrow$ steps (deriving the same expression e' and cost k').

5. THE ENHANCED PE SCHEME

This section describes the development of a narrowing-driven PE scheme enhanced with cost information. The narrowing-driven PE framework was first adapted to the flat syntax of Fig. 1 in [2] and later extended to cover all the additional features of the flat representation in [3]. These ideas gave rise to the first, purely declarative, partial evaluator for a realistic functional logic language like Curry [22]. Let us first recall some basic notions about the narrowing-driven PE procedure for flat programs.

```
Input: a program $\mathcal{R}$ and a set of expressions $E$
Output: a residual program $\mathcal{R}'$
Initialization: $i := 0$;  $E_0 := E$
Repeat
    $\mathcal{R}' := \mathit{unfold}(E_i, \mathcal{R})$;
    $E_{i+1} := \mathit{add\_exps}(E_i, \mathcal{R}'_{calls})$;
    $i := i + 1$;
Until $E_i = E_{i-1}$ (modulo renaming)
Return:
    $\mathcal{R}' = \mathit{post\_process}(\mathit{unfold}(E_i, \mathcal{R}))$
```

Figure 4: Narrowing-Driven PE Procedure

Essentially, narrowing-driven PE proceeds by iteratively unfolding a set of function calls, testing the *closedness* of the unfolded expressions, and adding to the current set those calls which are not closed. This process is repeated until all the unfolded expressions are closed (which guarantees the correctness of the transformation process [6]). Basically, an expression is *closed* whenever its maximal operation-rooted subterms are *constructor instances*[2] of the already partially evaluated terms (a more relaxed definition of closedness can be found in [4, 6]). This iterative style of performing PE was first described by Gallagher [16] for the PE of logic programs. Several (on-line) transformations, though formulated in a different style, can be easily recast in terms of Gallagher's algorithm.

The basic PE procedure can be seen in Figure 4. The operator *unfold* takes a set of expressions $E_i = \{\overline{e_n}\}$, computes a *finite* set of RLNT derivations, $e_j \Rightarrow^* e'_j$, and returns the set of residual rules ($e_j \ = \ e'_j$), for $j = 1, \ldots, n$. In order to ensure the finiteness of RLNT derivations, there exist a number of well-known techniques in the literature, e.g., depth-bounds, loop-checks, well-founded (or well-quasi) orderings (see, e.g., [12, 25, 29]). For instance, an unfolding rule based on the use of the homeomorphic embedding ordering was used in the INDY partial evaluator [6].

Function *add_exps* is used to add those expressions in the right-hand sides $\overline{e'_n}$ of the residual rules which are not closed w.r.t. E_i to the current set of (to be) partially evaluated expressions E_i. Here, we denote by $\mathcal{R}'_{calls}$ the expressions in the right-hand sides of the rules of $\mathcal{R}'$. Obviously, the more expensive part of the algorithm lies in the repeat-until loop. Finding a closed set of expressions is not an easy task and, in some cases, it could even be impossible (when the partially evaluated terms always contain new expressions which are not closed). More refined procedures, which use an *abstraction* operator to ensure the generation of a closed set of expressions in a finite number of iterations, can be found in [4, 6].

Therefore, the main loop of the algorithm can be seen as a *pre-processing* stage whose aim is to find a closed set of expressions. Note that no residual rules are actually constructed during this phase. Only when a closed set of expressions has been found, the unfolding operator is applied one more time in order to construct the associated residual program. Finally, a post-processing transformation is used to rename expressions—thus, some useless constructor symbols and repeated variables disappear—and to remove un-

[2]An expression e is a constructor instance of e' if $\sigma(e) = e'$ and, for all x, $\sigma(x)$ is a constructor term.

necessary (intermediate) functions—a typical post-unfolding *compression* phase [24].

Luckily, cost information need not be propagated along the whole PE process of Figure 4, but only during the last stage (the less expensive part of the algorithm). Since the original RLNT calculus and its cost-augmented version compute the same expressions, we can delay the application of the cost-augmented calculus to the point where residual rules are actually constructed. Hence, the main modification is to replace the last call to *unfold* by a new call to some *unfold′* which uses the cost-augmented RLNT calculus instead of the original one. As for the post-processing phase, it is basically extended as follows:

- the renaming of expressions does not affect to the computed costs, thus no extension is necessary;
- the post-unfolding process is managed by properly integrating the costs of the function used to perform an unfolding step into the costs of the expression to be unfolded.

The precise definitions are rather technical but easy (indeed, they have been incorporated into the prototype implementation shown in Section 6).

In this way, we obtain a PE scheme which returns a pair:

$$(\mathcal{R}', \mathcal{K}) = \mathit{post\_process}'(\mathit{unfold}'(E_i, \mathcal{R}))$$

where $\mathcal{R}'$ is a sequence of residual rules—a program—and $\mathcal{K}$ is a sequence of costs, one associated to each rule. In particular, for each rule $r_i \in \mathcal{R}'$, there is a corresponding cost $k_i \in \mathcal{K}$ which represents the cost of performing the RLNT computation which produced the residual rule r_i. Moreover, by using Theorem 1, it can be shown that this cost also coincides with the cost of performing an equivalent LNT derivation in the original program. The proof scheme is sketched as follows. Consider an expression $\sigma(e_1)$ and a residual rule $r_i = (e_1 = e_2)$. Consider also the LNT step

$$\sigma(e_1) \;\Rightarrow_{id}\; \sigma(e_2)$$

using the residual rule r_i, whose associated cost is k_i. From the correctness of the PE scheme, we know that the following cost-augmented RLNT derivation:

$$\langle K_0, \sigma(e_1)\rangle \;\Rightarrow^*\; \langle k_i, \sigma(e_2)\rangle$$

can be computed in the original program; indeed, it performs the same sequence of steps as the RLNT derivation used to construct the residual rule r_i, hence its associated cost is k_i too. Then, by Theorem 1, we know that there exists a corresponding cost-augmented LNT derivation

$$\langle K_0, \sigma(e_1)\rangle \;\Rightarrow^*_\theta\; \langle k'_i, e'\rangle$$

in the original program, such that $\langle k_i, \sigma(e_2)\rangle \;\hookrightarrow_\theta\; \langle k'_i, e'\rangle$. This means that k_i correctly represents the costs of all possible evaluations for $\sigma(e_1)$ in the original program (recorded with *alt* constructs; namely, it contains k'_i).

Now, in order to compare the cost of performing computations in the original and residual programs, we introduce the function *cost_rule*. It allows us to compute the cost associated to the application of a residual rule. Given a residual rule r whose associated cost is k, we compute $\mathit{cost\_rule}(k, r)$ as follows:

$$\mathit{cost\_rule}(k, r) = \mathit{cost}(k, e', e', 0)$$

where $r = (e = e')$ and $\mathit{cost}(k, e_1, e_2, c)$ is equal to

- $\{S \mapsto 1, C \mapsto c, A \mapsto \mathit{size}(e_2)\}$, if $k \neq \mathit{alt}(\ldots)$, and
- $\mathit{alt}(\overline{\mathit{cost}(k_m, e'_m, e_2, c+1)})$, if $k = \mathit{alt}(\overline{k_m})$, and $e_1 = (f)\mathit{case}\ x\ \{\overline{p_m \to e'_m}\}$.

Let us informally explain how this function computes the cost associated to a residual rule:

- the cost structure is the same as that of the cost k, i.e., there is the same number of *alt* constructs and in the same position;
- trivially, the number of steps is always one;
- the number of case evaluations (C) coincides with the number of *residual* case expressions;[3]
- the number of applications is equal to the size of the entire right-hand side of the residual rule.

Therefore, for each residual rule r, we have an associated pair of costs, (k, k'), where k is the cost of the RLNT derivation which produced r and $k' = \mathit{cost\_rule}(k, r)$. The intended meaning of (k, k') is as follows: the application of the residual rule r has an associated cost k', while an "equivalent" computation in the original program has an associated cost k. The computation of cost variation pairs was originally proposed by [1] for the PE of inductively sequential rewrite systems based on needed narrowing.

It could be argued that, for each residual rule r, the associated cost pair should be $(k, k + k')$, since the costs of performing partial computations should be also attributed to the residual rule. This is true when the specialized program is executed only once. In general, though, we can assume that specialized programs will be executed many times and, thus, we can safely ignore the costs of performing PE.

Example 4. Consider the RLNT derivation shown in Example 3. From this derivation, we get the residual rule r:

```
dapp x y z = fcase x of {
              [ ]        → fcase y of {
                                 [] → z;
                                 (y′ : ys′) → y′ : app ys′ z } ;
              (x′ : xs′) → x′ : dapp xs′ y z
            }
```

where `app (app x y) z` is renamed as `dapp x y z`. Below, we have its associated costs k and k', respectively:

$$\mathit{alt}(\{S \mapsto 2, C \mapsto 1, A \mapsto 30\}, \{S \mapsto 2, C \mapsto 2, A \mapsto 30\})$$

$$\mathit{alt}(\{S \mapsto 1, C \mapsto 1, A \mapsto 31\}, \{S \mapsto 1, C \mapsto 1, A \mapsto 31\})$$

By comparing k and k'—and, more precisely, the second argument of the *alt* construct—, it is easy to see that, for each element of the first input list of `dapp x y z`, the residual rule performs half the number of steps and half the number of case evaluations than the equivalent derivation in the original program.

From the cost variation pairs, we can easily design a simple speedup analysis to estimate the global improvement achieved by a particular specialization. In Section 6, we present an implementation of the enhanced PE scheme, together with a simple speedup analysis.

[3]Note that *non-residual* case expressions are not considered, since they have not been evaluated at PE time. Thus, we obtain a fair comparison by only counting the costs of performing the same evaluation steps in both programs.

5.1 Nondeterminism and PE

The current definitions of the enhanced RLNT calculus and the PE scheme does not take into account the amount of nondeterminism. It is true that current partial evaluators for functional logic programs (e.g., [3]) could reduce the amount of nondeterminism in the residual program, which leads to a reduction in the execution time and space requirements. Unfortunately, estimating the variation in terms of nondeterminism is not an easy task.

On the one hand, the RLNT calculus should be extended in order to properly treat "failed" branches. Consider, for instance, the following expression:

$$\texttt{fcase x of } \{ \quad [\,] \to \texttt{case 0 of } \{\texttt{1} \to \texttt{1}\}; \quad (\texttt{y} : \texttt{ys}) \to \texttt{g ys} \quad \}$$

Here, we can easily see that the first branch of the outer case structure will always fail. Hence, we could produce the equivalent expression:

$$\texttt{fcase x of } \{(\texttt{y} : \texttt{ys}) \to \texttt{g ys } \}$$

In this way, we transform a nondeterministic expression into a deterministic one.

In its present form, the RLNT calculus of Fig. 3 cannot change the nondeterministic behavior of the original program. Thus, it does not include an abstract cost to measure the amount of nondeterminism.[4] Due to this fact, the current RLNT calculus enjoys the following property: for each residual rule r, the associated costs in the original and residual programs, (k, k'), have exactly the same *structure* (i.e., the same number of *alt* constructs and in the same position). If we extend the PE scheme in order to be able to transform nondeterministic expressions into deterministic ones, the above property does not hold anymore. In this case, the comparison between the costs of the original and residual programs would be much more complex.

On the other hand, the current PE scheme also enjoys the following property: given an expression e, if the cost-augmented LNT semantics computes the derivation

$$\langle K_0, e\rangle \Rightarrow^*_\sigma \langle k, e'\rangle$$

then, for each constructor instance $\sigma(e)$ of e, we can also prove the derivation:

$$\langle K_0, \sigma(e)\rangle \Rightarrow^*_\sigma \langle k, \sigma(e')\rangle$$

where the associated cost remains unchanged. In other words, the instantiation of variables (with constructor terms) does not alter the computed costs. Informally speaking, this property is implied by the fact that the cost increment associated to the *case_select* and *case_guess* rules is the same. Moreover, it allows us to say that the costs computed during PE correctly represent the costs of the computations performed at execution time (where more instantiated calls usually occur).

Unfortunately, if we change the current definitions of the *case_guess* rules in order to take into account the amount of nondeterminism, then the above property is no longer true. Therefore, cost variation pairs (k, k') would only hold for the *concrete calls used to perform the specialization.* They would not correctly represent the costs of the computations with more instantiated calls.

[4]A cost-augmented calculus used to profile functional logic programs including nondeterminism can be found in [5].

The extension of the current PE scheme to take into account the amount of nondeterminism is an interesting—but difficult—problem. We consider it a promising subject for further research.

6. EXPERIMENTAL EVALUATION

In order to assess the practicality of the ideas presented in this work, the implementation of a cost-augmented partial evaluator for the multi-paradigm declarative language Curry has been undertaken.[5] Curry [22] integrates features from logic (logic variables, partial data structures, built-in search), functional (higher-order functions, demand-driven evaluation) and concurrent programming (concurrent evaluation of constraints with synchronization on logical variables). Furthermore, Curry is a complete programming language which is able to implement distributed applications (e.g. Internet servers [18]) or graphical user interfaces at a high-level [19]. Our PE tool has been implemented by extending an existing tool for the PE of Curry programs (see [3]). The enhanced PE tool is completely written in Curry itself, so it is easy to modify in order to incorporate future extensions. It takes source programs as input, which are automatically translated to the flat representation.

Let us consider a simple example to illustrate the behavior of the developed tool. Consider the well-known "all_ones" benchmark [13] (a typical example to illustrate the deforestation transformation [32]):

```
allones x = case x of
              { Z     -> [] ;
                (S y) -> 1 : allones y }
length x = case x of
             { []     -> Z ;
               (y:ys) -> S (length ys) }
```

The PE of this program w.r.t. the call "allones (length x)" returns the following residual program:

```
allones_pe x = case x of
                 { []     -> [] ;
                   (y:ys) -> 1 : allones_pe ys }
```

where allones_pe x is a renaming for allones (length x). The associated costs for this residual rule are:

```
Original: Alt (Cost 2 2 28 ) (Cost 2 2 28 )
Residual: Alt (Cost 1 1 15 ) (Cost 1 1 15 )
```

Here, we use expressions of the form (Cost S C A), where S denotes the number of steps, C the number of case evaluations, and A the number of applications. In this example, it is fairly easy to analyze the cost improvement achieved by narrowing-driven PE. Almost all the abstract costs have been halved. However, in more complex examples, it is not so easy to analyze the global improvement from the cost variation achieved by each residual rule. To overcome this problem, we have included a simple speedup analysis in our PE tool. It computes all the possible loops in the residual program, starting from the outermost function symbol of the initial call, and then sum up their associated costs. It proceeds, basically, by constructing a *dependency graph* for the function symbol of the partially evaluated call, thus it gives only approximate results (a higher bound).

Table 1 shows the results obtained from some selected

[5]It is publicly available from
http://www.dsic.upv.es/users/elp/soft.html.

Table 1: **Benchmark Results**

Benchmark	Original			Residual			Speedup
	S	C	A	S	C	A	
`all_ones`	2	2	28	1	1	15	1.29
`app_last`	2	3	36	1	1	13	2.83
`double_app`	2	2	30	1	1	31	1.30
`double_flip`	2	2	44	1	1	22	1.25
`kmp`	4	6	78	1	2	67	12.94
`length_app`	2	2	29	1	1	15	1.49
`loop_6`	6	1	30	1	1	15	1.15
`reverse`	1	1	15	1	1	15	1.00

benchmarks. Some of them are typical from deforestation (the case of `all_ones`, `app_last`, `double_app`, `double_flip`, `length_app`); `kmp` is the well-known "KMP test"; `loop_6` is a simple function with a 6-steps loop; `reverse` is a naive reversing function using `append`. The complete code of these benchmarks can be found within the implemented tool (some of them can be also found, e.g., in [6]). For each benchmark, we show the cost variation of the main loop in the program as well as the actual speedup (i.e., the ratio *original*/*residual*, where *original* is the runtime in the original program and *residual* is the runtime in the partially evaluated program). Runtime input goals were chosen to give a reasonably long overall time.

All benchmarks have been specialized w.r.t. function calls containing no static data, except for the `kmp` example (what explains the larger speedup produced). Although it is not obvious how to relate the variation of abstract costs to actual speedups, some conclusions can be made. Firstly, the most important cost seems to be the number of case evaluations. For instance, in the `loop_6` benchmark, the number of steps is reduced by a factor of 6. However, there is almost no speedup, which can be justified by the fact that the number of case evaluations is not reduced at all. In almost all the deforestation examples, both the steps and the pattern matchings are halved. Thus all of them achieve a similar speedup. The only exception is `app_last` that gets a better speedup, which can be explained by the bigger reduction in the number of pattern matchings (a factor of 3). Finally, `reverse` is not improved and, consequently, the associated abstract costs remain the same.

7. RELATED WORK

We find very little related works in the literature. Among them, let us briefly mention the closest to our approach. [7] established several properties of program transformations based on folding/unfolding in the context of logic programming. In particular, he proved that *superlinear* speedup cannot be accomplished by partial evaluation (this result can be also found in [8]). [8] developed a *speedup analysis* that, for any binding-time annotated program, computes a relative speedup interval such that the specialization of this program will result in a speedup within the predicted interval. The speedup analysis in the previous section is clearly inspired by the work of [8]. Also, [28] introduced a theory of cost equivalence that can be used to reason about the computational cost of *lazy* functional programs. Our cost variation pairs share some similarities with the equations generated in [28]. However, [28] centers the discussion in the number of evaluation steps, while we consider additional cost criteria.

The closest approach, though, is [1], where a formal framework to measure the effectiveness of PE in functional logic languages is introduced. Indeed, the present work can be seen as a natural evolution of the ideas presented in [1]. The main differences are the following:

- Firstly, [1] considers the basic narrowing-driven framework of [6], where programs are inductively sequential systems (with no distinction between flexible and rigid functions) and standard (needed) narrowing is used to perform computations at PE time. In contrast, our developments can be directly applied to practical partial evaluators—e.g., [3], which considers a non-standard, residualizing semantics to perform computations at PE time—for modern functional logic languages based on the combination of narrowing and residuation.

- Secondly, we have developed a formal specification of the cost-augmented semantics. While [1] shows how the cost of a narrowing derivation can be computed, there is no formalization of a "cost-augmented narrowing" relation.

- Finally, [1] considers the construction of (cost) recurrence equations associated to a particular PE. From these equations, one can analyze the cost of the program in several ways: by transforming them to ordinary recurrence equations over natural numbers—and, then, solving them—or by decorating each rule with a cost variation pair (obtained from the cost recurrence equations). In this paper, we developed further this idea in order to implement a simple speedup analysis.

In summary, while [1] presents two *independent* processes: narrowing-driven partial evaluation and generation of recurrence equations (to analyze the cost variation), we have fully integrated the computation of quantitative aspects into the narrowing-driven specialization technique.

8. CONCLUSIONS AND FUTURE WORK

We presented a first step towards the integration of quantitative aspects into the narrowing-driven PE framework. We introduced cost-augmented versions of the standard and residualizing semantics and proved their cost equivalence. We also sketched the scheme of a cost-enhanced partial evaluator. Preliminary experiments are encouraging and show that our ideas are both practical and useful.

A promising application of our developments is the generation of cost-guided partial evaluators. In particular, by experimenting with the developed PE tool, we discovered several common patterns associated with "successful" specializations. For instance, deforestation problems often reduce both the number of steps and the number of pattern matchings in the same factor. Thus, a cost-augmented partial evaluator may take this information into account to dynamically decide when to unfold and when to residualize expressions. The extension of the cost-augmented partial evaluator to consider the amount of nondeterminism is also subject of future work.

9. ACKNOWLEDGEMENTS

We gratefully acknowledge Elvira Albert for many helpful discussions on the topic of this paper. We also acknowledge the anonymous referees for many useful comments and suggestions.

10. REFERENCES

[1] E. Albert, S. Antoy, and G. Vidal. Measuring the Effectiveness of Partial Evaluation in Functional Logic Languages. In *Proc. of the 10th Int'l Workshop on Logic-based Program Synthesis and Transformation (LOPSTR'00)*, pages 103–124. Springer LNCS 2042, 2001.

[2] E. Albert, M. Hanus, and G. Vidal. Using an Abstract Representation to Specialize Functional Logic Programs. In *Proc. of the 7th Int'l Conf. on Logic for Programming and Automated Reasoning (LPAR'00)*, pages 381–398. Springer LNAI 1955, 2000.

[3] E. Albert, M. Hanus, and G. Vidal. A Practical Partial Evaluator for a Multi-Paradigm Declarative Language. In *Proc. of 5th Int'l Symp. on Functional and Logic Programming (FLOPS'01)*, pages 326–342. Springer LNCS 2024, 2001.

[4] E. Albert and G. Vidal. The Narrowing-Driven Approach to Functional Logic Program Specialization. *New Generation Computing*, 2001. To appear.

[5] E. Albert and G. Vidal. Source-Level Abstract Profiling of Multi-Paradigm Declarative Languages. In *Proc. of the 11th Int'l Workshop on Logic-based Program Synthesis and Transformation (LOPSTR'01)*, 2001. Available at `http://www.dsic.upv.es/~gvidal`.

[6] M. Alpuente, M. Falaschi, and G. Vidal. Partial Evaluation of Functional Logic Programs. *ACM TOPLAS*, 20(4):768–844, 1998.

[7] T. Amtoft. Properties of Unfolding-based Meta-level Systems. In *Proc. of the ACM Symp. on Partial Evaluation and Semantics-based Program Transformation (PEPM'91)*, 1991.

[8] L.O. Andersen and C.K. Gomard. Speedup Analysis in Partial Evaluation: Preliminary Results. In *Proc. of the ACM Workshop on Partial Evaluation and Semantics-based Program Transformation (PEPM'92)*, pages 1–7. Yale University, New Haven, CT, 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 and M. Hanus. Compiling Multi-Paradigm Declarative Programs into Prolog. In *Proc. of the Int'l Workshop on Frontiers of Combining Systems (FroCoS'2000)*, pages 171–185. Springer LNCS 1794, 2000.

[11] S. Antoy, B. Massey, and P. Julián. Improving the Efficiency of Non-Deterministic Computations. In *Proc. of the 10th Int'l Workshop on Functional and Logic Programming and 16th Workshop on Logic Programming (WFLP'01)*, Kiel, Germany, 2001.

[12] M. Bruynooghe, D. De Schreye, and B. Martens. A General Criterion for Avoiding Infinite Unfolding. *New Generation Computing*, 11(1):47–79, 1992.

[13] R.M. Burstall and J. Darlington. A Transformation System for Developing Recursive Programs. *Journal of the ACM*, 24(1):44–67, 1977.

[14] D. De Schreye, R. Glück, J. Jørgensen, M. Leuschel, B. Martens, and M.H. Sørensen. Conjunctive Partial Deduction: Foundations, Control, Algorihtms, and Experiments. *Journal of Logic Programming*, 41(2&3):231–277, 1999.

[15] 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.

[16] J. Gallagher. Tutorial on Specialisation of Logic Programs. In *Proc. of the ACM Symp. on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'93)*, pages 88–98. ACM, New York, 1993.

[17] M. Hanus. A unified computation model for functional and logic programming. In *Proc. of ACM Symp. on Principles of Programming Languages*, pages 80–93. ACM, New York, 1997.

[18] M. Hanus. Distributed Programming in a Multi-Paradigm Declarative Language. In *Proc. of the Int'l Conf. on Principles and Practice of Declarative Programming (PPDP'99)*, pages 376–395. Springer LNCS 1702, 1999.

[19] M. Hanus. A Functional Logic Programming Approach to Graphical User Interfaces. In *Proc. of the Int'l Workshop on Practical Aspects of Declarative Languages (PADL'00)*, pages 47–62. Springer LNCS 1753, 2000.

[20] M. Hanus, S. Antoy, J. Koj, R. Sadre, and F. Steiner. PAKCS 1.2: The Portland Aachen Kiel Curry System User Manual. Technical report, University of Kiel, Germany, 2000.

[21] M. Hanus and C. Prehofer. Higher-Order Narrowing with Definitional Trees. *Journal of Functional Programming*, 9(1):33–75, 1999.

[22] M. Hanus (ed.). Curry: An Integrated Functional Logic Language. Available at: `http://www.informatik.uni-kiel.de/~mh/curry/`.

[23] T. Hortalá-González and E. Ullán. An Abstract Machine Based System for a Lazy Narrowing Calculus. In *Proc. of the 5th Int'l Symp. on Functional and Logic Programming (FLOPS 2001)*, pages 216–232. Springer LNCS 2024, 2001.

[24] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, Englewood Cliffs, NJ, 1993.

[25] M. Leuschel. On the Power of Homeomorphic Embedding for Online Termination. In *Proc. of the Int'l Static Analysis Symposium (SAS'98)*, pages 230–245. Springer LNCS 1503, 1998.

[26] J.W. Lloyd and J.C. Shepherdson. Partial Evaluation in Logic Programming. *Journal of Logic Programming*, 11:217–242, 1991.

[27] W. Lux and H. Kuchen. An Efficient Abstract Machine for Curry. In *Proc. of the 8th Int'l Workshop on Functional and Logic Programming (WFLP'99)*, pages 171–181, 1999.

[28] D. Sands. A Naive Time Analysis and its Theory of Cost Equivalence. *Journal of Logic and Computation*, 5(4):495–541, 1995.

[29] M.H. Sørensen and R. Glück. An Algorithm of

Generalization in Positive Supercompilation. In *Proc. of the Int'l Logic Programming Symposium (ILPS'95)*, pages 465–479. The MIT Press, Cambridge, MA, 1995.

[30] M.H. Sørensen, R. Glück, and N.D. Jones. A Positive Supercompiler. *Journal of Functional Programming*, 6(6):811–838, 1996.

[31] V.F. Turchin. Program Transformation by Supercompilation. In H. Ganzinger and N.D. Jones, editors, *Programs as Data Objects, 1985*, pages 257–281. Springer LNCS 217, 1986.

[32] P.L. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.