Cost-Augmented Partial Evaluation of Functional Logic Programs *

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

Abstract. We enhance the narrowing-driven partial evaluation scheme for lazy functional logic programs with the computation of *symbolic* costs. The enhanced scheme allows us to estimate the effects of the program transformer in a precise framework and, moreover, to quantify these effects. The considered costs are "symbolic" in the sense that they measure the number of *basic* operations performed during a computation rather than actual execution times. Our scheme may serve as a basis to develop speedup analyses and cost-guided transformers. A cost-augmented partial evaluator, which demonstrates the usefulness of our approach, has been implemented in the multi-paradigm language Curry.

Keywords: partial evaluation, computational costs, functional logic programming

1. Introduction

Program transformation techniques include different methods whose common goal is to derive correct and efficient programs. This work is focused on a particular, fully automatic, program transformation called *partial evaluation*. 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 program for the given data. The new, residual program hopefully runs more efficiently since those computations that depend only on the static data have been performed once and for all at specialization time.

While correctness issues have been dealt with extensively in partial evaluation, 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 partial evaluation and (symbolic) profiling. We propose the enhancement of partial evaluation with the computation of *symbolic* costs. They are "symbolic" in the sense that they count the number of *basic* operations performed during a computation (e.g., function unfoldings, applications, pattern

* A preliminary version of this work appeared in the Proceedings of PEPM 2002 (Vidal, 2002). This work has been partially supported by MCYT HA2001-0059, by CICYT TIC 2001-2705-C03-01, and by Generalitat Valenciana CTIDIA/2002/205.

matchings, etc.) 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.

In order to center the discussion, we apply these ideas to a particular partial evaluation framework: the narrowing-driven approach to the partial evaluation of (lazy) functional logic programs (Albert and Vidal, 2002a). This framework has recently been extended by Albert et al. (2002b) in order to cope with modern implementations of functional logic languages by translating source programs into an intermediate form—called *flat* representation—where all the features of these language implementations can be represented at an adequate level of abstraction. A partial evaluator for flat programs has been incorporated into the PAKCS compiler (Hanus (ed.) et al., 2003) for the multi-paradigm language Curry (Hanus (ed.), 2003).

The syntax of functional logic programs is closer to the syntax of pure functional programs (e.g., the syntax of Curry is a conservative extension of that of Haskell). In spite of this fact, narrowing-driven partial evaluation shares more similarities with unification-based methods for the partial evaluation of logic programs—also known as *partial deduction* (Lloyd and Shepherdson, 1991)—than to traditional partial evaluators for functional programs based on constant propagation (Jones et al., 1993). In narrowing-driven partial evaluation, 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 partial evaluation procedure is a function call of the form $f(x_1, \ldots, x_n)$ where all the arguments are variables. In this case, it is often able to produce a new, residual function f' which is equivalent to f but more efficient, since all computations that are independent of the values of the input arguments can be precomputed in f'.

The first step in this work consists in instrumenting the standard semantics for functional logic programs in flat form, the LNT calculus, with the computation of symbolic costs. These costs are based on the cost criteria introduced by Albert et al. (2001) to measure the cost of functional logic computations: the number of steps, the pattern matching effort, and the number of applications. Additionally, we also measure the amount of non-determinism—similarly to the approach by Albert and Vidal (2002b) for source-level profiling—and the number of higher-order applications—which has not been considered before.

Narrowing-driven partial evaluation builds residual rules by means of a *residualizing* variant of the standard semantics: the RLNT calculus. 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. This is crucial to ensure the correctness of our approach: costs are computed by using the cost-augmented RLNT calculus, but the they should still correspond to the costs of equivalent computations with the LNT calculus; otherwise, the computation of costs during partial evaluation would be useless.

Finally, we introduce the scheme of a narrowing-driven partial evaluator which uses the cost-augmented RLNT calculus to perform partial computations. 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. This allows us to analyze the cost *improvement* associated to a partial evaluation. For instance, a cost-augmented partial evaluator could return the following residual function (this example is shown in detail in Section 6):

```
allones_pe []     = []                    (2,1)
allones_pe (x:xs) = 1 : allones_pe xs     (2,1)
```

Each rule is decorated with a *cost variation pair* which indicates that computations with the residual function `allones_pe` perform half the number of function unfoldings than equivalent computations in the original program. Similar cost variation pairs can also be shown for the remaining cost criteria. Decorated residual rules may help the user to quantify the improvement achieved by partial evaluation. Observe that we are *not* predicting the improvement. Rather, we estimate the improvement achieved *after* the computation of the residual program. This is a more modest—but still undecidable—problem.

Traditional approaches for measuring the improvement achieved by partial evaluators consider the use of profilers to estimate whether the transformation has been successful. These approaches have several drawbacks: one has to execute an extensive set of benchmarks in order to get reliable results, these results are only valid for a particular language implementation, and the user gets no information about which parts of the program have been improved and which are not. In contrast, our approach gives only an approximation of the global improvement but this estimation is independent of the language implementation and, moreover, the user gets precise information—the cost variation pairs—for each residual rule.

The main contributions of this work can be summarized as follows:

- We introduce a cost-augmented version of the LNT calculus, the standard semantics for flat programs. The considered costs include function unfoldings, applications, pattern matchings, nondeterministic branching points, and higher-order applications. This contrasts with previous approaches where only the number of steps is considered, e.g., (Andersen and Gomard, 1992; Sands, 1995).

- Our cost-augmented semantics covers the combination of needed narrowing and residuation, which forms the basis of the execution mechanism of modern functional logic languages like Curry (Hanus (ed.), 2003). Therefore, our approach is applicable to actual implementations of Curry, which contrasts with (Albert et al., 2001) where only needed narrowing is considered.

- We define a cost-augmented partial evaluation scheme that allows us to estimate the effects of the program transformer in a precise framework. Since the partial evaluator is based on a residualizing version of the standard semantics, the RLNT calculus, we also introduce a cost-augmented version of this calculus. The cost equivalence with the standard semantics is proved.

- Finally, since residual programs are usually executed with instances of the partially evaluated calls, it becomes important to study whether the cost-augmented semantics is closed under instantiation. We show that this is only true for some cost criteria, giving upper bounds for the remaining costs.

A cost-augmented partial evaluator, which demonstrates the usefulness of our approach, has been implemented in the multi-paradigm language Curry. The experiments indicate that our approach is both practical and useful. The enhanced partial evaluator also includes a simple *speedup analysis* which may be helpful to obtain a global measure of the improvement achieved by a concrete specialization.

The structure of the paper is as follows. We first recall, in Section 2, some fundamentals of functional logic programming. Section 3 introduces both a standard and a cost-augmented semantics for flat programs. Then, in Section 4, we define a cost-augmented version of the residualizing semantics and prove its equivalence with the cost-augmented standard semantics. The cost-augmented partial evaluation scheme is presented in Section 5. Some details of the implemented system, together with experimental results, are shown in Section 6. Several related works are discussed in Section 7 before we conclude in Section 8. Proofs of technical results can be found in Appendix A.

2. Functional Logic Programming

For the sake of completeness, we recall in this section some basic notions of term rewriting (Baader and Nipkow, 1998) and functional logic programming (Hanus, 1994).

2.1. Preliminaries

We consider a (*many-sorted*) *signature* Σ 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. A term is *linear* if it does not contain multiple occurrences of one variable. The set of variables occurring in a term t is denoted by $\mathcal{V}ar(t)$. A term t is *ground* if $\mathcal{V}ar(t) = \emptyset$.

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* (constructor-rooted) if it has an operation (constructor) symbol at the root. A *position* p in a term t is represented by a sequence of natural numbers (Λ 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* σ by $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ where $\sigma(x_i) = t_i$ for $i = 1, \ldots, n$ (with $x_i \neq x_j$ if $i \neq j$), and $\sigma(x) = x$ for all other variables x. A substitution σ is *constructor*, if $\sigma(x)$ is a constructor term for all x. The identity substitution is denoted by id. A substitution θ is more general than σ, in symbols $\theta \leq \sigma$, iff there exists a substitution γ such that $\gamma \circ \theta = \sigma$ ("$\circ$" denotes the composition operator). Term t' is a (constructor) *instance* of term t if there is a (constructor) substitution σ with $t' = \sigma(t)$.

A set of rewrite rules (or oriented equations) $l = r$ 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 $l = r \in \mathcal{R}$. 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 $R = (l = r)$ and a substitution σ with $t|_p = \sigma(l)$ and $s = t[\sigma(r)]_p$. The instantiated left-hand side $\sigma(l)$ of a rule $l = r$ is called a *redex* (*red*ucible *ex*pression). Given a relation $\to$, we denote by $\to^*$ its transitive and reflexive closure.

2.2. Narrowing

Functional *logic* programs mainly differ from purely functional programs in that function calls may contain free variables. In order to evaluate terms containing free 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 $t_0 \leadsto_\sigma^n t_n$ 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 $\sigma = id$). Due to the presence of free variables, a term may be reduced to different values after instantiating these variables to different terms. Given a narrowing derivation $t_0 \leadsto_\sigma^* t_n$, we say that t_n is a computed *value* and σ is a computed *answer* for t_0.

As in logic programming, narrowing derivations can be represented by a (possibly infinite) finitely branching *tree*. Formally, given a program $\mathcal{R}$ and an operation-rooted term t, a *narrowing tree* for t in $\mathcal{R}$ is a tree satisfying the following conditions: (a) each node of the tree is a term, (b) the root node is t, (c) if s is a node of the tree then, for each narrowing step $s \leadsto_{p,R,\sigma} s'$, the node has a child s' and the corresponding arc is labeled with (p, R, σ), and (d) nodes which are constructor terms have no children.

In order to avoid unnecessary computations and to deal with infinite data structures, demand-driven generation of the search space has been advocated by a number *lazy* narrowing strategies (Giovannetti et al., 1991; Loogen et al., 1993; Moreno-Navarro and Rodríguez-Artalejo, 1992). Due to its optimality properties w.r.t. the length of derivations and the number of computed solutions, *needed narrowing* (Antoy et al., 2000) is currently the best lazy narrowing strategy.

2.3. Needed Narrowing

Needed narrowing extends the notion of a *needed* rewrite step by Huet and Lévy (1992) to functional logic programming. Following (Antoy et al., 2000), a narrowing step $t \leadsto_{(p,R,\sigma)} t'$ is called *needed* iff, for every substitution θ such that $\sigma \leq \theta$, p is the position of a needed redex of $\theta(t)$ in the sense of (Huet and Lévy, 1992). 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. For the sake of completeness, we give an intuitive account of this concept; see (Antoy, 1992) for a complete definition. A program is *inductively sequential* when all its functions are defined by rewrite rules that, recursively, make on their arguments a case distinction analogous to a data type induction. Functions defined

in this way are also called *inductively sequential* and their rules can be organized in a hierarchical structure called a *definitional tree*. Antoy et al. (2000) define a needed narrowing strategy which determines how to evaluate a term depending on this case distinction. Consider, for instance, the following rules which define the less-or-equal function on natural numbers represented by terms built from `Z` and `Succ`:[1]

```
      Z ⩽ y         = True
(Succ x) ⩽ y        = False
(Succ x) ⩽ (Succ y) = x ⩽ y
```

In a term like $t_1 \leqslant t_2$, it is always necessary to evaluate t_1 to some *head normal form* (i.e., a variable or a constructor-rooted term) since all three rules defining "$\leqslant$" have a non-variable first argument. On the other hand, the evaluation of t_2 is only *needed* if t_1 is of the form `(Succ` t`)`. Thus, if t_1 is a free variable, needed narrowing instantiates it to a constructor, here `Z` or `(Succ x)`. Depending on this instantiation, either the first rule is applied or the second argument t_2 is evaluated.

2.4. Declarative Multi-Paradigm Languages

Functional logic languages have recently evolved to so called declarative *multi-paradigm* languages, e.g., Curry (Hanus (ed.), 2003), Toy (Hortalá-González and Ullán, 2001) and Escher (Lloyd, 1994). In order to make things concrete, we consider in this work the language Curry, a modern multi-paradigm language which integrates features from logic programming (partial data structures, built-in search), functional programming (higher-order functions, demand-driven evaluation) and concurrent programming (concurrent evaluation of constraints with synchronization on logical variables). Curry follows a Haskell-like syntax, i.e., variables and function names start with lowercase letters and data constructors start with an uppercase letter. The application of function f to an argument e is denoted by juxtaposition, i.e., $(f\ e)$. Functions in Curry are defined by a sequence of rules of the form

$$f\ t_1 \ldots t_n\ \texttt{=}\ e$$

where $t_1, \ldots, t_n$ are constructor terms and e is an expression. Functions can also be defined by *conditional equations* which have the form

$$f\ t_1 \ldots t_n\ \texttt{|}\ c\ \texttt{=}\ e$$

where the condition (or *guard*) c can be either a Boolean function or a *constraint*. Elementary constraints are `success`, which is always satisfied, and *equational constraints* e_1 `=:=` e_2 between two expressions. The

[1] We write constructor symbols starting with upper case (except for the list constructors, "[]" and ":", which are a shorthand for `Nil` and `Cons`, respectively).

latter is satisfied if both expressions are reducible to a same ground constructor term, i.e., we consider the so-called *strict equality*. Higher-order features include partial function applications and lambda abstractions. The evaluation of higher-order calls containing free variables as functions is not allowed, i.e., such calls are suspended to avoid the use of higher-order unification. Curry also allows the use of functions which are not defined in the user's program (*external* functions), like arithmetic operators, common higher-order functions (like `map` or `foldr`), monadic input/output, etc.

The basic operational semantics of Curry is based on a combination of needed narrowing and residuation (Hanus, 1997). The *residuation* principle is based on the idea of delaying function calls until they are ready for a deterministic evaluation. Residuation preserves the deterministic nature of functions and naturally supports concurrent computations. The precise mechanism—narrowing or residuation—for each function is specified by *evaluation annotations*. The annotation of a function as *rigid* forces the delayed evaluation by rewriting, while functions annotated as *flexible* can be evaluated in a non-deterministic manner by narrowing. If an explicit annotation is not provided, a default strategy is used: functions with result type "`IO ...`" (i.e., input/output functions) are rigid and all other functions are flexible.

Example 1. Consider the following rules

```
app []     ys = ys
app (x:xs) ys = x : app xs ys
```

which defines the well-known function `app` to concatenate two lists (where `[]` denotes the empty list and `x:xs` a list with first element `x` and tail `xs`). For instance, the equation "`app p s =:= [1,2,3]`" is solved by instantiating the variables `p` and `s` to lists so that their concatenation yields the list `[1,2,3]`. Thus, we can define a constraint which is satisfied if `s` is a suffix of the list `xs` as follows:[2]

```
suffix s xs  =  let ys free in app ys s =:= xs
```

In order to show an example for higher-order programming, we define below the usual higher-order functions `map` and `foldr`:

```
map f []     = []
map f (x:xs) = f x : map f xs

foldr f z []     = z
foldr f z (x:xs) = f x (foldr f z xs)
```

[2] The statement "`let ys free in` ..." is used to introduce a *logical* variable.

3. Cost-Augmented Semantics

In this section, we introduce a formal specification of the operational semantics for *flat* programs enhanced with the computation of costs.

3.1. The Flat Language

In actual implementations of declarative multi-paradigm languages, e.g., the PAKCS environment for Curry (Hanus (ed.) et al., 2003), the Münster Curry Compiler (Lux, 2003), and some implementations of Toy (Hortalá-González and Ullán, 2001), programs are translated into a *flat representation* (in the former two cases, it is called FlatCurry). This intermediate representation—originally introduced by Hanus and Prehofer (1999)—is based on the idea to "compile" definitional trees into specific rewrite rules. This provides more explicit control and leads to a calculus simpler than standard needed narrowing. Similarly to *Core* Haskell (Peyton Jones and Santos, 1998), our flat representation is not tied to any particular language implementation.

In order to ease the presentation, we only consider the *core* of the flat representation. Extending the developments in this work to the remaining features is not difficult and, indeed, the implementation reported in Section 6 covers these additional features. The syntax of flat programs is summarized in Figure 1. We use $x, y, z, \ldots$ for denoting *variables*, $a, b, c, \ldots$ for *constructors*, and $f, g, h, \ldots$ for defined functions. A program $\mathcal{R}$ consists of a sequence of function definitions D such that each function is defined by a single rule whose left-hand side contains only different variables as parameters. The right-hand side is an expression[3] e composed by variables, constructors, function calls, higher-order applications, and case expressions for pattern-matching. The general form of a case expression is as follows:[4]

$$(f)\mathit{case}\ e\ \mathit{of}\ \{c_1(\overline{x_{n_1}}) \to e_1; \ldots; c_m(\overline{x_{n_m}}) \to e_m\}$$

where e is an expression, $c_1, \ldots, c_m$ are different constructors of the type of e, and $e, e_1, \ldots, e_m$ are expressions (possibly containing nested (f)*case*'s). The variables $\overline{x_{n_i}}$ are local variables which occur only in e_i. The difference between *case* and *fcase* shows up when the argument e is a free variable: *case* suspends—which corresponds to residuation—whereas *fcase* non-deterministically binds this variable to a pattern in a branch of the case expression—which corresponds to narrowing.

[3] We use the word *expression*—in contrast to *term*—to indicate that e may contain case structures and higher-order applications.

[4] In the following, we write $\overline{o_n}$ for the *sequence of objects* $o_1, \ldots, o_n$ and (f)*case* for either *fcase* or *case*.

$\mathcal{R}$	::=	$D_1 \ldots D_m$	(program)
$\mathcal{D}$	::=	$f(x_1, \ldots, x_n) = e$	(rule)
e	::=	x	(variable)
	|	$c(e_1, \ldots, e_n)$	(constructor call)
	|	$f(e_1, \ldots, e_n)$	(function call)
	|	*case* e *of* $\{p_1 \to e_1; \ldots; p_m \to e_m\}$	(rigid case)
	|	*fcase* e *of* $\{p_1 \to e_1; \ldots; p_m \to e_m\}$	(flexible case)
	|	$partcall(f, e_1, \ldots, e_k)$	(partial application)
	|	$apply(e_1, e_2)$	(application)
p	::=	$c(x_1, \ldots, x_n)$	(flat pattern)

Figure 1. Syntax of Flat Programs

The flat language is first-order, i.e., source programs are *defunctionalized* (Reynolds, 1998) by using the operators *partcall* and *apply*: *partcall* is used to denote partial function applications and *apply* makes the application of a function to an argument explicit (Warren, 1982). This strategy is followed by many language implementations, e.g., PAKCS (Hanus (ed.) et al., 2003) and Münster Curry (Lux, 2003).

Example 2. Consider again the (inductively sequential) function `app` of Example 1. It can be defined in the flat representation as follows:[5]

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

The higher-order functions `map` and `foldr` are defined with the help of the auxiliary operators *apply* and *partcall*:

```
map f xs = case xs of { []     → [] ;
                        (y:ys) → apply f y : map f ys }
foldr f z xs = case xs of
               {[]    → z ;
               (y:ys)→ apply (apply f y) (foldr f z ys)}
```

An automatic transformation from inductively sequential programs to programs using case expressions is introduced by Hanus and Prehofer (1999). While the definition of an operational semantics for source programs is rather involved—e.g., it requires the computation of definitional trees and an operational mechanism which integrates needed narrowing and residuation—the operational semantics for flat programs is defined by means of the eight simple transition rules of Figure 2.

[5] After defunctionalization, the flat language becomes first-order. Nevertheless, we still use a curried notation in concrete examples.

hnf

$$c(e_1,\ldots,e_i,\ldots,e_n) \Rightarrow_\sigma \sigma(c(e_1,\ldots,e'_i,\ldots,e_n)) \quad \text{if } c \in \mathcal{C},\ e_i \text{ is not a constructor term, and } e_i \Rightarrow_\sigma e'_i$$

case_select

$$(f)case\ c(\overline{e_n})\ of\ \{\overline{p_m \to e'_m}\} \Rightarrow_{id} \sigma(e'_i) \quad \text{if } p_i = c(\overline{x_n}),\ c \in \mathcal{C}, \text{ and } \sigma = \{\overline{x_n \mapsto e_n}\}$$

case_guess

$$fcase\ x\ of\ \{\overline{p_m \to e_m}\} \Rightarrow_\sigma \sigma(e_i) \quad \text{if } \sigma = \{x \mapsto p_i\} \text{ and } i \in \{1,\ldots,m\}$$

case_eval

$$(f)case\ e\ of\ \{\overline{p_m \to e_m}\} \Rightarrow_\sigma \sigma((f)case\ e'\ of\ \{\overline{p_m \to e_m}\}) \quad \text{if } e \text{ is not in head normal form and } e \Rightarrow_\sigma e'$$

fun_eval

$$f(\overline{e_n}) \Rightarrow_{id} \sigma(e) \quad \text{if } f(\overline{x_n}) = e \in \mathcal{R} \text{ and } \sigma = \{\overline{x_n \mapsto e_n}\}$$

apply_total

$$apply(partcall(f,e_1,\ldots,e_k),e) \Rightarrow_{id} f(e_1,\ldots,e_k,e) \quad \text{if } f/n \in \mathcal{F} \text{ and } k+1=n$$

apply_partial

$$apply(partcall(f,e_1,\ldots,e_k),e) \Rightarrow_{id} partcall(f,e_1,\ldots,e_k,e) \quad \text{if } f/n \in \mathcal{F} \text{ and } k+1<n$$

apply_eval

$$apply(e_1,e_2) \Rightarrow_\sigma apply(e'_1,e_2) \quad \text{if } e_1 \neq partcall(\ldots) \text{ and } e_1 \Rightarrow_\sigma e'_1$$

Figure 2. Standard LNT Calculus

Our operational semantics, the LNT (for Lazy Narrowing with definitional Trees) calculus, is an extension of the LNT calculus of Albert et al. (2003) to cope with higher-order applications. The one-step transition relation $\Rightarrow_\sigma$ is labeled with the substitution σ computed in the step. Let us briefly describe the LNT rules:

The hnf rule can be applied to evaluate expressions in head normal form (i.e., a variable or a constructor-rooted term). It proceeds by recursively evaluating some argument (e.g., the leftmost one) that contains some defined function call. There is no rule to evaluate constructor terms; in this case, the computation stops *successfully*.

The case_select rule selects the appropriate branch of a case expression and continues with the evaluation of this branch. This rule implements *pattern matching*.

The case_guess rule applies when the argument of a flexible case expression is a variable. Then, this rule non-deterministically binds this variable to a pattern in a branch of the case expression. The step is labeled with the computed binding. Observe that there is no rule to

evaluate a rigid case expression with a variable argument. This situation produces a *suspension* of the evaluation.

The case_eval rule can be applied when the argument of the case construct is not in head normal form (i.e., it is either a function call, another case construct or a higher-order application). Then, it tries to evaluate this expression recursively.

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 apply_total rule is used to evaluate a higher-order application of a partial function with exactly one missing argument. This rule proceeds by adding the missing argument and continuing with the evaluation of the resulting total function.

The apply_partial rule applies to higher-order applications of partial functions with more than one missing argument. In this case, we simply add the new argument to the partial call.

Finally, the apply_eval rule is used to evaluate the first argument of a higher-order application until it becomes a partial call.

An LNT derivation $e \Rightarrow^*_\sigma e'$ is *successful* when e' is a constructor term. Then, we say that e evaluate to e' with computed answer σ.

Example 3. Consider the function app of Example 2. Given the initial call "app (1 : 2 : x) (3 : [])," the LNT calculus computes, among others, the following successful derivation:

```
app (1 : 2 : x) (3 : [])
⇒id   fcase (1 : 2 : x) of                                   (fun_eval)
        { []       → 3 : [] ;
          (z : zs) → z : app zs (3 : []) }
⇒id   1 : app (2 : x) (3 : [])                               (case_select)
⇒id   1 : (fcase (2 : x) of                                  (hnf/fun_eval)
        { []       → 3 : [] ;
          (z : zs) → z : app zs (3 : []) } )
⇒id   1 : 2 : (app x (3 : []))                               (hnf/case_select)
⇒id   1 : 2 : (fcase x of                                    (hnf/fun_eval)
        { []       → 3 : [] ;
          (z : zs) → z : app zs (3 : []) } )
⇒{x↦[]} 1 : 2 : 3 : []                                       (hnf/case_guess)
```

Therefore, the initial expression evaluates to "1:2:3:[]" with computed answer $\{x \mapsto []\}$.

Example 4. Consider again the function `foldr` of Example 2 together with the following function `sum`:

```
sum x y = fcase x of { Z        → y ;
                       (Succ w) → Succ (sum w y) }
```

The call "`foldr (partcall sum) Z (Succ Z : [])`" is reduced as follows:

```
foldr (partcall sum) Z (Succ Z : [])
⇒id case (Succ Z : []) of                                          (fun_eval)
       { []       → Z ;
         (y : ys) → apply (apply (partcall sum) y)
                          (foldr (partcall sum) Z ys)}
⇒id apply (apply (partcall sum) (Succ Z))                          (case_select)
          (foldr (partcall sum) Z [])
⇒id apply (partcall sum (Succ Z))                  (apply_eval/apply_partial)
          (foldr (partcal sum) Z [])
⇒id sum (Succ Z) (foldr (partcall sum) Z [])       (apply_eval/apply_total)
⇒id fcase (Succ Z) of                                              (fun_eval)
       { Z        → (foldr (partcall sum) Z []) ;
         (Succ w) → Succ (sum w (foldr (partcall sum) Z [])) }
⇒id Succ (sum Z (foldr (partcall sum) Z []))                       (case_select)
⇒id Succ (fcase Z of                                              (hnf/fun_eval)
           { Z        → (foldr (partcall sum) Z []) ;
             (Succ w) → Succ (sum w (foldr (partcall sum) Z [])) }
⇒id Succ (foldr (partcall sum) Z [])                             (hnf/case_select)
⇒id Succ (case [] of                                              (hnf/fun_eval)
           { []       → Z ;
             (y : ys) → apply (apply (partcall sum) y)
                              (foldr (partcall sum) Z ys) } )
⇒id Succ Z                                                       (hnf/case_select)
```

3.2. Cost-Augmented LNT Calculus

In this section, we instrument the standard semantics with the computation of cost information. Our symbolic costs are independent of a particular implementation of the language. Rather, they are are based on operations that are performed by likely implementations of (lazy) functional logic languages. We assume that the number of defined functions in a program—i.e., the *size* of the program—does not affect the execution time. The reason is that, in a first-order language, a reference to the symbol being applied can be resolved at compile-time.

We consider five cost criteria which characterize the cost of a functional logic computation:

Function unfoldings (U): Note that this cost does not correspond with the number of evaluation *steps* in more traditional approaches since, typically, a computation step also includes the time spent to perform pattern matching, the allocation of memory cells to store the right-hand side of the unfolded rule, etc. Here, we consider each source of time consumption separately.

Case evaluations (C): This symbolic cost abstracts the pattern matching effort in a computation. This cost criterion was originally introduced by Albert et al. (2001), where it is defined as the number of constructor symbols in the left-hand side of the unfolded rule. In the flat representation, pattern matching is encoded by case expressions. Therefore, by counting the number of case evaluations we get exactly the same measure in our setting.

Applications (A): This cost is introduced to measure memory allocation. The original formulation by Albert et al. (2001) defines this criterion as the number of non-variable symbols in the right-hand side of each unfolded rule. However, this definition is not appropriate in general. Consider, e.g., the following function definition:

$$\texttt{foo x} \;=\; \texttt{foo x}$$

According to (Albert et al., 2001), we should count one application for each unfolding step. However, in any meaningful implementation, the execution of a function call like "`foo` e" should not increasingly allocate new memory cells, since there is no need to build new data structures with each function unfolding. This is independent of the use of particular optimization strategies (e.g., related to tail recursion). This is coherent, e.g., with the PAKCS environment (Hanus (ed.) et al., 2003) and the Münster Curry Compiler (Lux, 2003), but it also holds in standard implementations of Haskell and Prolog—for a corresponding predicate definition.

In this work, we define the number of applications as the number of symbols in the *non-variable arguments* of the function (or constructor) call that appears in the right-hand side of the rule. The precise definition can be found below.

Higher-order applications (HO): Although this cost has not been considered before, it has a clear impact in the execution time. In fact, transforming a higher-order program into a first-order one involves a significant speedup, as we will see in Section 6.

Non-deterministic branching points (N): This symbolic cost is useful to measure the work needed either to create, update or remove a choice point. It is equal to 0 when the argument of a case expression matches exactly one pattern, and 1 otherwise. The computed cost is independent of the implemented search strategy.

The cost-augmented semantics is formalized by the state-transition rules of Figure 3. 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 $\in \{U, C, A, HO, N\}$. As in the standard semantics, the one-step transition relation $\Rightarrow_\sigma$ is labeled with the substitution σ computed in the step.

Rules hnf, case_eval, and apply_eval are self-explanatory. The current cost is modified according to the recursive computation.

In the case_select rule, the current cost is updated by adding one to cost variable C and by adding the number of allocated cells for the selected branch to cost variable A. Function *alloc* returns the number of memory cells required to allocate an expression. Since the expressions introduced by instantiation have been already allocated in some previous step, we define $alloc(e) = alloc'(e')$, where $e = \sigma(e')$, e' is a (sub)expression of some program rule and σ contains the bindings applied to such (sub)expression in the considered computation.[6] The auxiliary function $alloc'$ is defined as follows:

$$alloc'(e) = \begin{cases} \Sigma_{i=j_1}^{j_m} |e_i| & \text{if } e = \varphi(\overline{e_n}) \text{ and } \overline{e_{j_m}} \text{ are the nonvariable arguments of } \varphi \\ |e_0| & \text{if } e = (f)case\ e_0\ of\ \{\overline{p_m \rightarrow e_m}\} \text{ and } e_0 \text{ is not a variable} \\ 0 & \text{otherwise} \end{cases}$$

where φ denotes either a function symbol, a constructor symbol, *apply*, or *partcall*. Function $|\_|$ is used to measure the *size* of an expression:

$$|e| = \begin{cases} 1 & \text{if } e = x \\ 1 + |e_1| + \ldots + |e_n| & \text{if } e = \varphi(\overline{e_n}) \\ 0 & \text{otherwise} \end{cases}$$

Note that we do not consider the branches of inner case expressions (only the argument) since they will be considered in subsequent steps

[6] A formal treatment could be given by recording the computed bindings into a separate component rather than applying them to the expression being evaluated (i.e., by making explicit the *heap* in the states). The LNT calculus could easily be reformulated in this way but we keep the original formulation for simplicity.

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 $c \in \mathcal{C}$, 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 $k' = k[C \leftarrow C+1, A \leftarrow A + alloc(e'_i)]$

case_guess_det
$$\langle k, fcase\ x\ of\ \{p \to e\}\rangle \Rightarrow_\sigma \langle k', \sigma(e)\rangle$$
if $\sigma = \{x \mapsto p\}$ and $k' = k[C \leftarrow C+1, A \leftarrow A + |p| + alloc(e)]$

case_guess_nondet
$$\langle k, fcase\ x\ of\ \{\overline{p_m \to e_m}\}\rangle \Rightarrow_\sigma \langle k', \sigma(e_i)\rangle$$
if $\sigma = \{x \mapsto p_i\}$, $i \in \{1,\ldots,m\}$, $m > 1$, and $k' = k[C \leftarrow C+1, A \leftarrow A + |p_i| + alloc(e_i), N \leftarrow N+1]$

case_eval
$$\langle k, (f)case\ e\ of\ \{\overline{p_m \to e_m}\}\rangle \Rightarrow_\sigma \langle k', \sigma((f)case\ e'\ of\ \{\overline{p_m \to e_m}\})\rangle$$
if e is not in head normal form 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 $f(\overline{x_n}) = e \in \mathcal{R}$, $\sigma = \{\overline{x_n \mapsto e_n}\}$, and $k' = k[U \leftarrow U+1]$

apply_total
$$\langle k, apply(partcall(f, e_1,\ldots,e_k), e)\rangle \Rightarrow_{id} \langle k', f(e_1,\ldots,e_k,e)\rangle$$
if $f/n \in \mathcal{F}$, $k+1 = n$, and $k' = k[HO \leftarrow HO+1]$

apply_partial
$$\langle k, apply(partcall(f, e_1,\ldots,e_k), e)\rangle \Rightarrow_{id} \langle k', partcall(f, e_1,\ldots,e_k, e)\rangle$$
if $f/n \in \mathcal{F}$, $k+1 < n$, and $k' = k[HO \leftarrow HO+1]$

apply_eval
$$\langle k, apply(e_1, e_2)\rangle \Rightarrow_\sigma \langle k', apply(e'_1, e_2)\rangle$$
if $e_1 \neq partcall(\ldots)$ and $\langle k, e_1\rangle \Rightarrow_\sigma \langle k', e'_1\rangle$

Figure 3. Cost-Augmented LNT Calculus

(provided that their evaluation is actually required). Let us illustrate these definitions with some examples:

$$alloc'(x : xs) = 0$$
$$alloc'(x : y : ys) = |y : ys| = 3$$
$$alloc'(f(x, g(y), x : xs)) = |g(y)| + |x : xs| = 5$$

Observe that, by using these definitions, calls of the form "`foo` e" to the previous function `foo` do not allocate memory cells since $alloc($`foo` $e) = alloc'($`foo x`$) = 0$. However, if we change the definition of `foo` as follows:

```
foo x = foo (x + 1)
```

each call "`foo` e" will require some memory allocations:

$$alloc(\texttt{foo}\ (e+1)) = alloc'(\texttt{foo}\ (\texttt{x}+1)) = |\texttt{x}+1| = 3$$

Rules case_guess_det and case_guess_nondet updates the current cost by adding one to cost variable C and by adding the size of the selected pattern as well as the number of allocated cells for the selected branch to cost variable A. The case_guess_nondet rule additionally adds one to cost variable N since there is more than one branch in the case expression, therefore a choice point is created.

Finally, rule fun_eval increments cost variable U while rules apply_total and apply_partial increment cost variable HO.

In the cost-augmented LNT calculus, *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 $\sigma = id$). As in the standard LNT calculus, a derivation $\langle K_0, e\rangle \Rightarrow^*_\sigma \langle k, e'\rangle$ is *successful* when e' is a constructor term. Then, we say that e evaluates to e' with computed answer σ and associated cost k.

Example 5. Consider again the function app of Example 2 and the successful derivation which is shown in Example 3. The corresponding cost-augmented derivation is as follows:

$\langle \texttt{K}_0,\ \texttt{app}\ (\texttt{1} : \texttt{2} : \texttt{x})\ (\texttt{3} : [\,])\rangle$

$\Rightarrow_{id}$ $\langle \{\texttt{U} \mapsto \texttt{1}, \texttt{C} \mapsto \texttt{0}, \texttt{A} \mapsto \texttt{0}, \texttt{HO} \mapsto \texttt{0}, \texttt{N} \mapsto \texttt{0}\},$ (fun_eval)

```
  fcase (1 : 2 : x) of
           { [ ]       → 3 : [ ] ;
             (z : zs)  → z : app zs (3 : [ ]) }⟩
```

$\Rightarrow_{id}$ $\langle \{\texttt{U} \mapsto \texttt{1}, \texttt{C} \mapsto \texttt{1}, \texttt{A} \mapsto \texttt{3}, \texttt{HO} \mapsto \texttt{0}, \texttt{N} \mapsto \texttt{0}\},$ (case_select)

```
  1 : app (2 : x) (3 : [ ]) ⟩
```

$\Rightarrow_{id}$ $\langle \{\texttt{U} \mapsto \texttt{2}, \texttt{C} \mapsto \texttt{1}, \texttt{A} \mapsto \texttt{3}, \texttt{HO} \mapsto \texttt{0}, \texttt{N} \mapsto \texttt{0}\},$ (hnf/fun_eval)

```
  1 : (fcase (2 : x) of
         { [ ]       → 3 : [ ] ;
           (z : zs)  → z : app zs (3 : [ ]) } ) ⟩
```

$\Rightarrow_{id}$ $\langle \{\texttt{U} \mapsto \texttt{2}, \texttt{C} \mapsto \texttt{2}, \texttt{A} \mapsto \texttt{6}, \texttt{HO} \mapsto \texttt{0}, \texttt{N} \mapsto \texttt{0}\},$ (hnf/case_select)

```
  1 : 2 : (app x (3 : [ ])) ⟩
```

$\Rightarrow_{id}$ $\langle \{\texttt{U} \mapsto \texttt{3}, \texttt{C} \mapsto \texttt{2}, \texttt{A} \mapsto \texttt{6}, \texttt{HO} \mapsto \texttt{0}, \texttt{N} \mapsto \texttt{0}\},$ (hnf/fun_eval)

```
  1 : 2 : (fcase x of
             { [ ]       → 3 : [ ] ;
               (z : zs)  → z : app zs (3 : [ ]) }, ) ⟩
```

$\Rightarrow_{\{\texttt{x} \mapsto [\,]\}}$ $\langle \{\texttt{U} \mapsto \texttt{3}, \texttt{C} \mapsto \texttt{3}, \texttt{A} \mapsto \texttt{7}, \texttt{HO} \mapsto \texttt{0}, \texttt{N} \mapsto \texttt{1}\},$ (hnf/case_guess)

```
  1 : 2 : 3 : [ ] ⟩
```

Example 6. Consider now the the higher-order derivation of Example 4. The corresponding cost-augmented derivation is as follows:

```
⟨K0, foldr (partcall sum) Z (Succ Z : [ ])⟩
 ⇒id ⟨{U ↦ 1, C ↦ 0, A ↦ 0, HO ↦ 0, N ↦ 0},                    (fun_eval)
       case (Succ Z : [ ]) of
          { [ ]       →  Z ;
            (y : ys)  →  apply (apply (partcall sum) y)
                                (foldr (partcall sum) Z ys)} ⟩
 ⇒id ⟨{U ↦ 1, C ↦ 1, A ↦ 7, HO ↦ 0, N ↦ 0},                 (case_select)
       apply (apply (partcall sum) (Succ Z))
             (foldr (partcall sum) Z [ ]) ⟩
 ⇒id ⟨{U ↦ 1, C ↦ 1, A ↦ 7, HO ↦ 1, N ↦ 0},     (apply_eval/apply_partial)
       apply (partcall sum (Succ Z))
             (foldr (partcal sum) Z [ ]) ⟩
 ⇒id ⟨{U ↦ 1, C ↦ 1, A ↦ 7, HO ↦ 2, N ↦ 0},       (apply_eval/apply_total)
       sum (Succ Z) (foldr (partcall sum) Z [ ]) ⟩
 ⇒id ⟨{U ↦ 2, C ↦ 1, A ↦ 7, HO ↦ 2, N ↦ 0},                    (fun_eval)
       fcase (Succ Z) of
         { Z         →  (foldr (partcall sum) Z [ ]) ;
           (Succ w)  →  Succ (sum w (foldr (partcall sum) Z [ ])) } ⟩
 ⇒id ⟨{U ↦ 2, C ↦ 2, A ↦ 10, HO ↦ 2, N ↦ 0},                (case_select)
       Succ (sum Z (foldr (partcall sum) Z [ ])) ⟩
 ⇒id ⟨{U ↦ 3, C ↦ 2, A ↦ 10, HO ↦ 2, N ↦ 0},                 (hnf/fun_eval)
       Succ (fcase Z of
              { Z         →  (foldr (partcall sum) Z [ ]) ;
                (Succ w)  →  Succ (sum w (foldr (partcall sum) Z [ ])) } ⟩
 ⇒id ⟨{U ↦ 3, C ↦ 3, A ↦ 13, HO ↦ 2, N ↦ 0},             (hnf/case_select)
       Succ (foldr (partcall sum) Z [ ]) ⟩
 ⇒id ⟨{U ↦ 4, C ↦ 3, A ↦ 13, HO ↦ 2, N ↦ 0},                 (hnf/fun_eval)
       Succ (case [ ] of
              { [ ]       →  Z ;
                (y : ys)  →  apply (apply (partcall sum) y)
                                   (foldr (partcall sum) Z ys) } ) ⟩
 ⇒id ⟨{U ↦ 4, C ↦ 4, A ↦ 13, HO ↦ 2, N ↦ 0}, Succ Z⟩    (hnf/case_select)
```

Trivially, the cost-augmented semantics of Figure 3 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 Partial Evaluation

Narrowing-driven partial evaluation was originally designed to use the standard semantics to perform computations at specialization time (Albert and Vidal, 2002a; Alpuente et al., 1998). Within this scheme, residual rules are constructed from partial computations as follows. Given an expression e and a (possibly incomplete) computation $e \Rightarrow^*_\sigma e'$, 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; for instance, the left-hand sides of resultants may become instantiated, which is not allowed by the syntax of Figure 1 (only variable arguments are valid). Therefore, Albert et al. (2000) 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 following expression:

$$\texttt{fcase x of } \{\texttt{0} \to \texttt{0};\ \texttt{1} \to \texttt{foo x y}\}$$

the standard semantics non-deterministically computes either "`0`" (with associated binding $\{\texttt{x} \mapsto \texttt{0}\}$) or "`foo 1 y`" (with associated binding $\{\texttt{x} \mapsto \texttt{1}\}$). In contrast, the residualizing semantics leaves the case structure and proceeds with the evaluation of both branches (i.e., "0", which is already evaluated, and "`foo 1 y`"). Thus, the residualizing semantics does not compute bindings but encodes them by means of residualized case expressions with a variable argument.

In this section, we introduce a cost-augmented version of the RLNT calculus. This will form the basis of a cost-augmented partial evaluator, as we will see in Section 5. Roughly speaking, for each partial computation $e \Rightarrow^* e'$ (with the RLNT calculus), we produce an associated residual rule $e = e'$. Now, our aim is to determine whether the residual rule is actually an improvement, i.e., is there any gain in using rule $e = e'$ instead of performing the original computation $e \Rightarrow^* e'$? The answer should be clear: yes! This is indeed the purpose of partial evaluation. Usually, part of the costs of the partial computation will be still present in the residual rule but, hopefully, some operations have been performed once and for all in $e \Rightarrow^* e'$ and, therefore, the application of rule $e = e'$ involves some efficiency gain. For instance, if rule case_select is applied in the partial computation, then the application of the residual rule will involve a gain in the number of case evaluations (since case structures disappear whenever rule case_eval is applied). This point is further discussed in Section 6.

case_select
$\langle k, (f)case\ c(\overline{e_n})\ of\ \{\overline{p_m \to e'_m}\}\rangle \Rightarrow \langle k', \sigma(e'_i)\rangle$
if $p_i = c(\overline{x_n})$, $\sigma = \{\overline{x_n \mapsto e_n}\}$, and $k' = k[C \leftarrow C+1, A \leftarrow A + alloc(e'_i)]$

case_guess_det
$\langle k, (f)case\ x\ of\ \{p \to e\}\rangle \Rightarrow \langle alt(k'), (f)case\ x\ of\ \{p \to \sigma(e)\}\rangle$
if $\sigma = \{x \mapsto p\}$ and $k' = k[C \leftarrow C+1, A \leftarrow A + |p| + alloc(e)]$

case_guess_nondet
$\langle k, (f)case\ x\ of\ \{\overline{p_m \to e_m}\}\rangle \Rightarrow \langle alt(\overline{k_m}), (f)case\ x\ of\ \{\overline{p_m \to \sigma_m(e_m)}\}\rangle$
if $k \neq alt(\ldots)$ and, for all $i = 1, \ldots, m$ $(m > 1)$, $\sigma_i = \{x \mapsto p_i\}$
and $k_i = k[C \leftarrow C+1, A \leftarrow A + |p_i| + alloc(e_i), N \leftarrow N+1]$

case_guess_eval
$\langle alt(\overline{k_m}), (f)case\ x\ of\ \{\overline{p_m \to e_m}\}\rangle \Rightarrow \langle alt(\overline{k'_m}), (f)case\ x\ of\ \{\overline{p_m \to e'_m}\}\rangle$
if $\langle k_i, e_i\rangle \Rightarrow \langle k'_i, e'_i\rangle$, $i \in \{1, \ldots, m\}$, with $k'_j = k_j$ and $e'_j = e_j$ for all $j \neq i$

case_of_case
$\langle k, (f)case\ ((f)case\ x\ of\ \{\overline{p'_n \to e'_n}\})\ of\ \{\overline{p_m \to e_m}\}\rangle$
$\Rightarrow \langle k, (f)case\ x\ of\ \{\overline{p'_n \to (f)case\ e'_n\ of\ \{\overline{p_m \to e_m}\}}\}\rangle$

case_eval
$\langle k, (f)case\ e\ of\ \{\overline{p_m \to e_m}\}\rangle \Rightarrow \langle k', (f)case\ e'\ of\ \{\overline{p_m \to e_m}\}\rangle$
if e is operation-rooted or a case expression with
a non-variable argument, and $\langle k, e\rangle \Rightarrow \langle k', e'\rangle$

fun_eval
$\langle k, f(\overline{e_n})\rangle \Rightarrow_{id} \langle k', \sigma(e)\rangle$
if $f(\overline{x_n}) = e \in \mathcal{R}$, $\sigma = \{\overline{x_n \mapsto e_n}\}$, and $k' = k[U \leftarrow U+1]$

apply_total
$\langle k, apply(partcall(f, e_1, \ldots, e_k), e)\rangle \Rightarrow_{id} \langle k', f(e_1, \ldots, e_k, e)\rangle$
if $f/n \in \mathcal{F}$, $k+1 = n$, and $k' = k[HO \leftarrow HO+1]$

apply_partial
$\langle k, apply(partcall(f, e_1, \ldots, e_k), e)\rangle \Rightarrow_{id} \langle k', partcall(f, e_1, \ldots, e_k, e)\rangle$
if $f/n \in \mathcal{F}$, $k+1 < n$, and $k' = k[HO \leftarrow HO+1]$

apply_eval
$\langle k, apply(e_1, e_2)\rangle \Rightarrow_{\sigma} \langle k', apply(e'_1, e_2)\rangle$
if $e_1 \neq partcall(\ldots)$ and $\langle k, e_1\rangle \Rightarrow_{\sigma} \langle k', e'_1\rangle$

Figure 4. Cost-Augmented RLNT Calculus

The cost-augmented RLNT calculus is shown in Figure 4. The same considerations of Section 3.2 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; this allows us to use the same arrow "$\Rightarrow$" for the formalization of both calculi without confusion.

Note that there is no rule to evaluate terms in head normal form. This is not a restriction of the partial evaluator since the calculus is applied—at partial evaluation time—iteratively and, thus, the arguments of a head normal form will be evaluated in the next iteration of the algorithm (compare the partial evaluation procedure in Section 5).

The main difference w.r.t. the cost-augmented LNT calculus is in the definition of rules case_guess_det and case_guess_nondet. Basically, they proceed as in the previous case but residualize the case structure (the "bindings") rather than performing a non-deterministic branching. Nevertheless, they apply 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 keep track of the costs for the different alternatives. For instance, given the state $\langle alt(k_1, k_2),\ case\ x\ of\ \{p_1 \to e_1;\ p_2 \to e_2\}\rangle$, k_1 denotes the costs attributed to e_1 and k_2 those attributed to e_2. Of course, we could follow a simpler strategy and mark each branch with the current cost, i.e., $case\ x\ of\ \{p_1 \to \langle k_1, e_1\rangle;\ p_2 \to \langle k_2, e_2\rangle\}$. However, this will only postpone the creation of *alt* constructs since, in the end, we should produce pairs (r, k) where r is a legal program rule (in order to produce *executable* residual programs!).

After one application of some of the previous rules, the new rule case_guess_eval 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.

Finally, due to the residualization of case structures, a new rule to evaluate nested case expressions (where the inner case has a variable argument) becomes necessary. For this purpose, we introduce the case_of_case rule, which moves the outer case inside the branches of the inner one. Similar rules can be found, e.g., in deforestation (Wadler, 1990) and driving (Sørensen et al., 1996). The current costs remain unchanged, since no expression is reduced and the order of evaluation is not modified. Rigorously speaking, this rule can be expanded into four rules (with the different combinations for *case* and *fcase*), 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 are not significantly changed. Observe that the application of rule fun_eval may duplicate computations under a graph-based implementation of narrowing. In other words, the partial evaluation scheme of Albert et al. (2002b) may destroy sharing during the transformation process. To overcome this problem, a simple solution

consists in adding the following condition to fire rule fun_eval: the right-hand side of the unfolded rule should be linear (i.e., without repeated occurrences of a same variable). Another solution consists in using a sharing-based calculus to perform partial computations, like, e.g., the operational semantics of Albert et al. (2002a). The definition of such a partial evaluator is subject of ongoing work.

Example 7. Consider again the function app of Example 2. Given the initial state $\langle \mathtt{K}_0,\ \mathtt{app}\ (\mathtt{app}\ \mathtt{x}\ \mathtt{y})\ \mathtt{z} \rangle$, the cost-augmented RLNT calculus computes, for instance, the following partial computation:

$$
\begin{array}{ll}
\langle \mathtt{K}_0,\ \mathtt{app}\ (\mathtt{app}\ \mathtt{x}\ \mathtt{y})\ \mathtt{z} \rangle & \\
\Rightarrow\ \langle \{\mathtt{U} \mapsto 1, \mathtt{C} \mapsto 0, \mathtt{A} \mapsto 0, \mathtt{HO} \mapsto 0, \mathtt{N} \mapsto 0\}, & \text{(fun\_eval)} \\
\quad \mathtt{fcase}\ (\mathtt{app}\ \mathtt{x}\ \mathtt{y})\ \mathtt{of} & \\
\quad\quad \{\ [\,] \ \to\ \mathtt{z}\ ; & \\
\quad\quad\ (\mathtt{y}' : \mathtt{ys}') \ \to\ \mathtt{y}' : \mathtt{app}\ \mathtt{ys}'\ \mathtt{z}\ \}\ \rangle & \\
\Rightarrow\ \langle \{\mathtt{U} \mapsto 2, \mathtt{C} \mapsto 0, \mathtt{A} \mapsto 0, \mathtt{HO} \mapsto 0, \mathtt{N} \mapsto 0\}, & \text{(case\_eval/fun\_eval)} \\
\quad \mathtt{fcase}\ (\mathtt{fcase}\ \mathtt{x}\ \mathtt{of}\ \{[\,] \to \mathtt{y};\ (\mathtt{x}' : \mathtt{xs}') \to\ \mathtt{x}' : \mathtt{app}\ \mathtt{xs}'\ \mathtt{y}\})\ \mathtt{of} & \\
\quad\quad \{\ [\,] \ \to\ \mathtt{z}\ ; & \\
\quad\quad\ (\mathtt{y}' : \mathtt{ys}') \ \to\ \mathtt{y}' : \mathtt{app}\ \mathtt{ys}'\ \mathtt{z}\ \}\ \rangle & \\
\Rightarrow\ \langle \{\mathtt{U} \mapsto 2, \mathtt{C} \mapsto 0, \mathtt{A} \mapsto 0, \mathtt{HO} \mapsto 0, \mathtt{N} \mapsto 0\}, & \text{(case\_of\_case)} \\
\quad \mathtt{fcase}\ \mathtt{x}\ \mathtt{of} & \\
\quad\quad \{\ [\,] \ \to\ \mathtt{fcase}\ \mathtt{y}\ \mathtt{of}\ \{[] \to \mathtt{z}; (\mathtt{y}' : \mathtt{ys}') \to \mathtt{y}' : \mathtt{app}\ \mathtt{ys}'\ \mathtt{z}\}\ ; & \\
\quad\quad\ (\mathtt{x}' : \mathtt{xs}') \ \to\ \mathtt{fcase}\ (\mathtt{x}' : \mathtt{app}\ \mathtt{xs}'\ \mathtt{y})\ \mathtt{of} & \\
\quad\quad\quad\quad\quad\quad \{\ [] \ \to\ \mathtt{z}; & \\
\quad\quad\quad\quad\quad\quad\ (\mathtt{y}' : \mathtt{ys}') \ \to\ \mathtt{y}' : \mathtt{app}\ \mathtt{ys}'\ \mathtt{z}\ \}\ \} \rangle & \\
\Rightarrow\ \langle \mathtt{alt}(\{\mathtt{U} \mapsto 2, \mathtt{C} \mapsto 1, \mathtt{A} \mapsto 1, \mathtt{HO} \mapsto 0, \mathtt{N} \mapsto 1\}, & \text{(case\_guess\_nondet)} \\
\quad\quad \{\mathtt{U} \mapsto 2, \mathtt{C} \mapsto 1, \mathtt{A} \mapsto 6, \mathtt{HO} \mapsto 0, \mathtt{N} \mapsto 1\}), & \\
\quad \mathtt{fcase}\ \mathtt{x}\ \mathtt{of} & \\
\quad\quad \{\ [\,] \ \to\ \mathtt{fcase}\ \mathtt{y}\ \mathtt{of}\ \{[] \to \mathtt{z}; (\mathtt{y}' : \mathtt{ys}') \to \mathtt{y}' : \mathtt{app}\ \mathtt{ys}'\ \mathtt{z}\}\ ; & \\
\quad\quad\ (\mathtt{x}' : \mathtt{xs}') \ \to\ \mathtt{fcase}\ (\mathtt{x}' : \mathtt{app}\ \mathtt{xs}'\ \mathtt{y})\ \mathtt{of} & \\
\quad\quad\quad\quad\quad\quad \{\ [] \ \to\ \mathtt{z}; & \\
\quad\quad\quad\quad\quad\quad\ (\mathtt{y}' : \mathtt{ys}') \ \to\ \mathtt{y}' : \mathtt{app}\ \mathtt{ys}'\ \mathtt{z}\ \}\ \} \rangle & \\
\Rightarrow\ \langle \mathtt{alt}(\{\mathtt{U} \mapsto 2, \mathtt{C} \mapsto 1, \mathtt{A} \mapsto 1, \mathtt{HO} \mapsto 0, \mathtt{N} \mapsto 1\}, & \text{(case\_guess\_eval/case\_select)} \\
\quad\quad \{\mathtt{U} \mapsto 2, \mathtt{C} \mapsto 2, \mathtt{A} \mapsto 9, \mathtt{HO} \mapsto 0, \mathtt{N} \mapsto 1\}), & \\
\quad \mathtt{fcase}\ \mathtt{x}\ \mathtt{of} & \\
\quad\quad \{\ [\,] \ \to\ \mathtt{fcase}\ \mathtt{y}\ \mathtt{of}\ \{[] \to \mathtt{z}; (\mathtt{y}' : \mathtt{ys}') \to \mathtt{y}' : \mathtt{app}\ \mathtt{ys}'\ \mathtt{z}\}\ ; & \\
\quad\quad\ (\mathtt{x}' : \mathtt{xs}') \ \to\ \mathtt{x}' : \mathtt{app}\ (\mathtt{app}\ \mathtt{xs}'\ \mathtt{y})\ \mathtt{z}\ \} \rangle &
\end{array}
$$

The computed cost is an *alt* construct whose arguments denote the costs for the computation of the alternatives of the outer case expression. Note that there is no *alt* construct associated to the inner case expression since it has not been evaluated in the considered derivation.

The cost-augmented semantics of Figure 4 is also a conservative extension of the original RLNT calculus. Formally, given a cost-augmented RLNT derivation $\langle K_0, e\rangle \Rightarrow^* \langle k, e'\rangle$, we have that $e \Rightarrow^* e'$ is a standard RLNT derivation, and vice versa.

For each call e, a partial evaluator based on the cost-augmented RLNT calculus computes a partial derivation $\langle K_0, e\rangle \Rightarrow^* \langle k, e'\rangle$ and, then, outputs a pair $(e = e', k)$. Now, we can compute the cost k' of applying rule r according to the cost-augmented LNT semantics. From these costs, k and k', we can check whether an improvement has been achieved in the transformation process. Observe that this comparison is not fair if the cost-augmented LNT calculus and the cost-augmented RLNT calculus are not equivalent. Therefore, we now prove that they are indeed equivalent. The main difference between these calculi is that the RLNT calculus encodes the computed bindings by means of residualized case expressions. Thus, we need some mechanism to extract the expressions in the branches of residualized case expressions together with their associated bindings. For this purpose, we introduce an auxiliary relation, $\hookrightarrow$, which is defined by the transition rule:

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

where $\sigma = \{x \mapsto p_i\}$ for some $i \in \{1, \ldots, m\}$.

The precise equivalence between the calculi of Figure 3 and Figure 4 can now be stated as follows:[7]

Theorem 1. Let e be an expression, e' a head normal form, and $\mathcal{R}$ a flat program. For each cost-augmented LNT derivation $\langle K_0, e\rangle \Rightarrow^*_\sigma \langle k', e'\rangle$ in $\mathcal{R}$ where rule hnf is not applied, there exists a cost-augmented RLNT derivation $\langle K_0, e\rangle \Rightarrow^* \langle k'', e''\rangle$ in $\mathcal{R}$ such that $\langle k'', e''\rangle \hookrightarrow^*_\sigma \langle k', e'\rangle \not\hookrightarrow$, and vice versa.

Basically, this result guarantees that the costs computed at partial evaluation time are equivalent to those computed at execution time. This allows us to design a cost-augmented partial evaluator based on the instrumented RLNT calculus.

5. The Enhanced Partial Evaluation Scheme

In this section, we describe the integration of cost information into the narrowing-driven partial evaluation scheme. Narrowing-driven partial evaluation was first adapted to the flat syntax of Figure 1 by Albert

[7] The notation $\langle k', e'\rangle \not\hookrightarrow$ is used to indicate that the state $\langle k', e'\rangle$ is irreducible w.r.t. the relation $\hookrightarrow$, i.e., there is no state $\langle k'', e''\rangle$ such that $\langle k', e'\rangle \hookrightarrow \langle k'', e''\rangle$.

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

Figure 5. Narrowing-Driven Partial Evaluation Procedure

et al. (2000) and, then, extended to cover all the additional features of the flat representation (Albert et al., 2002b). These ideas gave rise to the first, purely declarative, partial evaluator for a realistic functional logic language like Curry. Let us first recall some basic notions about the narrowing-driven partial evaluation procedure for flat programs.

Essentially, partial evaluation 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 (Alpuente et al., 1998). An expression is *closed* whenever its maximal operation-rooted subterms are constructor instances of the already partially evaluated calls; a more relaxed definition of closedness can be found in (Alpuente et al., 1998). This iterative style of performing partial evaluation was first described by Gallagher (1993) for the partial evaluation of logic programs. Several (online) transformations, though formulated in a different style, can easily be recast in terms of Gallagher's algorithm.

The basic partial evaluation procedure can be seen in Figure 5. 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., (Bruynooghe et al., 1992; Leuschel, 1998; Sørensen and Glück, 1995). For instance, an unfolding rule based on the use of the homeomorphic embedding ordering is used in the partial evaluator for Curry programs of Albert et al. (2002b).

Function *add_exps* is used to add those expressions in the right-hand sides $e'_1, \ldots, e'_n$ which are not closed w.r.t. E_i to the current set

of (to be) partially evaluated expressions. 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 (e.g., when the unfolded expressions in each iteration always contain new calls 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 (Alpuente et al., 1998).

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. Observe that no residual program is actually built during this phase. Only when a closed set of expressions is eventually 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 unnecessary (intermediate) functions—a typical post-unfolding *compression* phase (Jones et al., 1993). The renaming phase begins by computing an independent renaming for the closed set of expressions. Informally speaking, an *independent renaming* for a set of expressions is a mapping which assigns to each expression a function call $f(x_1, \ldots, x_n)$ where $x_1, \ldots, x_n$ are the distinct variables of this expression and f is a fresh function symbol (Alpuente et al., 1997). Then, residual rules are renamed by *recursively* replacing each call in the given rule by a call to the corresponding renamed function (according to the independent renaming).

Luckily, 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. The main modification is thus to replace the last call to *unfold* by a new call to some *unfold'* which uses the cost-augmented RLNT calculus rather than the original RLNT calculus. 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 adding the costs of the function used to perform an unfolding step to the costs of the expression to be unfolded. This phase may also remove a case expression when its argument does not match any pattern in the corresponding branches. In this case, the costs associated to the deleted case expression are simply removed.

The precise definitions are rather technical but easy (indeed, they have been incorporated into the partial evaluator described in Section 6). In summary, the cost-augmented partial evaluator is equal to the one shown in Figure 5 except for the last line, which is now replaced by

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

where *unfold'* and *post_process'* are the new operators described above. For each rule $r_i \in \mathcal{R}'$, there is an associated cost $k_i \in \mathcal{K}$ which represents the cost of performing the RLNT computation which produced such a residual rule (and, by Theorem 1, it also represents the cost of the corresponding LNT derivation).

As mentioned before, we want to compute the *improvement* achieved by each residual rule. Therefore, the cost of applying residual rules is also needed. For this purpose, we introduce the function *cost_rule*. In particular, given a residual rule $r = (e = e')$ whose associated cost is k, we define $cost\_rule(k, r)$ as follows:

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

where $cost(k, e, case, apps, nd)$ returns

- $\{U \mapsto 1, C \mapsto case, A \mapsto apps, HO \leftarrow 0, N \leftarrow nd\}$ if $k \neq alt(\ldots)$
- $alt(k'_1)$ if $k = alt(k_1)$, $e = (f)case\ x\ \{p_1 \to e_1\}$, and $k'_1 = cost(k_1, e_1, case + 1, apps + |p_1| + alloc(e_1), nd)$
- $alt(\overline{k'_m})$ if $k = alt(\overline{k_m})$, $e = (f)case\ x\ \{\overline{p_m \to e_m}\}$, and $k'_i = cost(k_i, e_i, case + 1, apps + |p_i| + alloc(e_i), nd + 1)$ for all $i = 1, \ldots, m$.

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

U : Trivially, this cost is always 1.

C : In order to perform a fair comparison, we only count *residualized* case expressions, since non-residualized case expressions are not considered in the cost k computed by the partial evaluator. Indeed, function *cost_rule* requires cost k in order to check whether a case expression has been residualized or not—in the former case, there must be an associated *alt* construct.

A : This cost is computed in a similar way as in the cost-augmented semantics.

HO : The cost computed by the partial evaluator only counts the occurrences of *apply* that are actually evaluated. Since these occurrences disappear from the residual rule (see rules apply_total and apply_partial), we always have $HO = 0$.

N : Similarly to cost C, the value of N is equal to the number of *residualized* case expressions with more than one alternative.

Therefore, for each residual rule r, we have an associated pair, (k, k'), where k is the cost of the RLNT derivation which produced r and $k' = cost\_rule(k, r)$. The intended meaning of (k, k') is as follows: the application of the residual rule r has a cost k' while an "equivalent" computation in the original program would have a cost k.

Example 8. Consider the RLNT derivation which is shown in Example 7. 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:

$$k = alt(\{U \mapsto 2, C \mapsto 1, A \mapsto 1, HO \mapsto 0, N \mapsto 1\},\\ \{U \mapsto \boxed{2}, C \mapsto \boxed{2}, A \mapsto 9, HO \mapsto 0, N \mapsto 1\})$$

$$k' = alt(\{U \mapsto 1, C \mapsto 1, A \mapsto 1, HO \mapsto 0, N \mapsto 1\},\\ \{U \mapsto \boxed{1}, C \mapsto \boxed{1}, A \mapsto 7, HO \mapsto 0, N \mapsto 1\})$$

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, computations in the residual program perform half the number of function unfoldings and half the number of case evaluations than equivalent computations in the original program.

In principle, the computed costs (k, k') only correspond to actual costs when the executed call belongs to the set of partially evaluated calls. In general, though, we will execute *instances* of these calls. The next theorem states a useful property to analyze this relation. Given a cost k, we denote by $V(k)$ the value of an arbitrary cost variable V in k.

Theorem 2. Let e be an expression with $\langle K_0, e\rangle \Rightarrow^*_\sigma \langle k_1, e'\rangle$. Let $\theta(e)$ be a constructor instance of e with $\langle K_0, \theta(e)\rangle \Rightarrow^*_{\sigma'} \langle k_2, e''\rangle$, where e'' is a constructor instance of e' and $\sigma' \leq \sigma$. Then $U(k_1) = U(k_2)$, $C(k_1) = C(k_2)$, $A(k_1) \geq A(k_2)$, $HO(k_1) = HO(k_2)$, and $N(k_1) \geq N(k_2)$.

Therefore, costs U, C and HO are closed under instantiation, while A and N may decrease when a call is further instantiated. This means that the cost-augmented partial evaluator gives precise results regarding costs U, C, and HO, but it computes only an *upper bound* for the remaining costs A and N. Consider, for instance, that the partial evaluator returns a residual rule with associated costs (k, k') such that $N(k) = 2$ and $N(k') = 1$. This means that, *for the execution of the partially evaluated call*, we avoid the creation of one non-deterministic branching point each time the residual rule is applied. However, *for a particular instance of the partially evaluated call*, it may happen that no non-deterministic branching point is created neither in the original nor in the residual programs.

6. Experimental Evaluation

In order to assess the practicality of the ideas presented so far, a cost-augmented partial evaluator for the declarative multi-paradigm language Curry has been developed.[8] It is implemented by extending an existing tool for the partial evaluation of Curry programs (Albert et al., 2002b). The enhanced partial evaluator is completely written in Curry itself, so it is purely declarative. It takes Curry programs as input and translates them automatically into the flat representation to apply the cost-augmented partial evaluation process.

In order to produce a global measure of the improvement achieved by a partially evaluated program, we have included a simple speedup analysis in our partial evaluator. It computes all the potential loops in the residual program starting from the outermost function symbol of the initial call, and then sums up their associated costs. Basically, we construct a *dependency graph* for the residual functions and, thus, it only gives an approximation. Besides the computation of such a global measure, we think that it is even more interesting for the user to analyze the cost variation achieved by each residual rule. This gives a precise information regarding the behavior of the partial evaluation process—where it achieved significant optimizations and where it failed to improve the original code, regarding each cost criteria. Surely, this is far more useful for the expert user than a single measure about the global improvement. Nevertheless, a global measure may still help the non-expert to have a rough idea of whether the partial evaluation process achieved a significant improvement in efficiency or it just produced a cost-equivalent variant of the original program.

[8] Publicly available at `http://www.dsic.upv.es/users/elp/german/soft.html`.

The forthcoming sections illustrate through examples the usefulness of the cost-augmented partial evaluator.

6.1. Deforestation

In this section, we examine some typical benchmarks for deforestation (Wadler, 1990). Consider, for instance, the well-known "all_ones" program (Burstall and Darlington, 1977):

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

where natural numbers are represented by terms built from `Z` (zero) and `S` (successor). Cost-augmented partial evaluation w.r.t. the call "`allones (length x)`" returns the following residual program:

```
allones_pe x = case x of { []      → [] ;
                           (2 2 1 0 1) → (1 1 1 0 1)
                           (y:ys) → 1 : allones_pe ys }
                           (2 2 8 0 1) → (1 1 6 0 1)

Speedup Analysis:
Loop1: (2 2 8 0 1) → (1 1 6 0 1)
```

where "`allones_pe x`" is a renaming for "`allones (length x)`". In the examples, rather than showing the computed costs as a single structure (with *alt* constructs), we put the arguments of each *alt* construct in the corresponding branch of the case expression for clarity. Costs are denoted by tuples of the form $(U\ C\ A\ HO\ N)$.

In the first branch of `allones_pe`, both U and C are halved. However, this is not relevant since this case is executed only once. If we look at the second branch—where the recursive call appears—, we observe that U and C are also halved, while A is reduced by a factor of $8/6 = 1.33$. Although small, this is a significant improvement for large input data, since this loop will be executed many times. These numbers represent a quantification of the effect of deforestation. The result of the speedup analysis only shows the cost variation associated to the recursive branch of the case expression (the only program loop).

Another typical example of deforestation is the "double-append" program. Consider again the standard definition of function `app` to concatenate two lists (see Example 2). By computing the concatenation of three lists with a call of the form "`app (app x y) z`" we incur into a serious loss of efficiency, since the elements of list `x` should

be traversed twice. Cost-augmented partial evaluation w.r.t. the call "`app (app x y) z`" returns the following residual program:

```
dapp_pe x y z = case x of
                {[]     → case y of
                           {[]     → z;
                                     (2 2 2 0 2) → (1 2 2 0 2)
                            (w:ws) → w : app ws z };
                                     (2 2 7 0 2) → (1 2 7 0 2)
                 (t:ts) → t : dapp_pe ts y z }
                                     (2 2 9 0 1) → (1 1 7 0 1)

Speedup Analysis:
Loop1: (2 2 9 0 1) → (1 1 7 0 1)
```

where "`dapp_pe x y z`" is a renaming for "`app (app x y) z`" and `app` is a call to the original function (which remains unchanged). If we consider the main loop—the second branch of the outer case expression—, we can observe that both U and C are halved, while A is reduced by a factor of $9/7 = 1.29$. Indeed, many deforestation examples follow a similar pattern (see Table I).

6.2. Removal of Higher-Order Calls

A powerful optimization achieved by the narrowing-driven partial evaluator is the transformation of higher-order functions into first-order functions. This reduces the execution time and space requirements w.r.t. almost all language implementations. Consider again the higher-order functions `map` and `foldr` of Example 2. Cost-augmented partial evaluation w.r.t. the call "`foldr (+) 0 (map (+1) xs)`" returns the following residual program:

```
foldr_pe xs = case xs of
              { []     → 0 ;
                                (2 2 1 0 1) → (1 1 1 0 1)
                (y:ys) → (y + 1) + foldr_pe ys }
                                (3 2 11 3 1) → (1 1 8 0 1)
Speedup Analysis:
Loop1: (3 2 11 3 1) → (1 1 8 0 1)
```

where "`foldr_pe xs`" is a renaming for "`foldr (+) 0 (map (+1) xs)`". Note that the residual program is now first-order. In particular, if we consider the recursive branch of function `foldr_pe`, we can see that U is reduced to a third, C is halved, A is reduced by a factor of $11/8 = 1.37$, and HO is reduced by 3/0 (i.e., all higher-order applications have been removed).

6.3. Removal of Non-Determinism

Another important source of improvement in a functional *logic* language is the removal of unnecessary non-deterministic branching points. Let us consider the following simple program:

```
nondet x y = fcase x of { Z     → foo1 Z ;
                          (S z) → foo2 y }

foo1 x = fcase x of { (Succ y) → Z }

foo2 x = fcase x of {[]     → Z;
                     (y:ys) → nondet w ys where w free}
```

where "`nondet w ys where w free`" means that `w` is a logical variable. Given a call of the form "`nondet x [1..100]`" where `x` is a logical variable and `[1..100]` is a list from `1` to `100`, we perform 101 calls to function `nondet` with a variable first argument. The important point is that each recursive call to `nondet` creates a choice point. This choice point is unnecessary since the call to function `foo1` in the first branch of the case expression always fails. By applying the cost-augmented partial evaluator to the initial call "`nondet x y`", we get the program

```
nondet_pe x y = fcase x of
                {(S z) → fcase y of
                         {[]     → Z;
                              (2 3 4 0 2) → (1 3 4 0 1)
                          (y:ys) → nondet_pe w ys
                                   where w free }
                              (2 3 6 0 2) → (1 3 6 0 1)
Speedup Analysis:
Loop1: (2 3 6 0 2) → (1 3 6 0 1)
```

If we look at the second branch of the inner case expression—where the recursive call to `nondet_pe` appears—we can see that N is halved. This implies a significant improvement since the outer case expression is now deterministic and, thus, given a call like "`nondet x [1..100]`", no choice point is created during the computation. Table I shows the speedup associated to a call of this form.

6.4. Program Specialization

A well-known strength of partial evaluation is its ability to *specialize* a given program w.r.t. some known input data. A classical example is the specialization of a (semi)naive pattern matcher for a fixed pattern into

an efficient algorithm—sometimes called "the KMP-test" (Ager et al., 2002; Consel and Danvy, 1989). The considered program is as follows:

```
match p s = loop p s p s

loop a b op os =
  case a of {[]     → True ;
             (p:ps) → case b of
              {[]     → False;
               (s:ss) → if eq p s then loop ps ss op os
                                  else next op os } }

next op x = case x of { []     → False ;
                        (s:ss) → loop op ss op ss }
```

where function `eq` is an equality test for the finite alphabet $\{\mathtt{A}, \mathtt{B}\}$. Function calls of the form "`match p s`" check whether pattern `p` appears in string `s`. Cost-augmented partial evaluation w.r.t. the call "`match (A:A:B:[]) s`" returns the following specialized program:

```
match_pe s = loop_pe s                (1 0 0 0 0) → (1 0 0 0 0)

loop_pe x = case x of
             {[]     → False;  (1 2 1 0 1) → (1 1 1 0 1)
              (s:ss) → case s of
                {A → case ss of
                   {[]     → False;
                                (4 7 47 0 3) → (1 3 5 0 3)
                    (b:bs) → case b of
                      {A  → case bs of
                         {[]     → False;
                             (7 12 95 0 5) → (1 5 9 0 5)
                          (c:cs) → case c of
                            {B → True;
                           (10 16 144 0 6) → (1 6 12 0 6)
                              A → loop_pe (A:A:cs)}}
                           (10 16 157 0 6) → (1 6 17 0 6)
                       B → loop_pe (B:bs)}}
                            (7 11 105 0 4) → (1 4 11 0 4)
                 B → loop_pe ss}}
                                 (4 6 55 0 2) → (1 2 4 0 2)
Speedup Analysis:
Loop1: (10 16 157 0 6) → (1 6 17 0 6)
Loop2: (7  11 105 0 4) → (1 4 11 0 4)
Loop3: (4   6  55 0 2) → (1 2  4 0 2)
```

Let us briefly analyze the results. As pointed out by the speedup analysis, we have three possible loops in the program (the three branches where recursive calls to `loop_pe` appear). While HO and N do not change—it was not the goal of this benchmark—, U, C and A are significantly improved; namely, U is reduced within the interval $[4-10]$, C within the interval $[2.66-3]$, and A within the interval $[9.24-13.75]$.

6.5. Benchmarks

In order to check experimentally the usefulness of our approach, we show the actual speedups of some selected benchmarks and relate them with the cost variation pairs computed by the partial evaluator. The considered benchmarks are the following: all_ones and double_append, the examples shown in Section 6.1; app_last, double_flip, and length_app, three typical benchmarks for deforestation; app_2steps, the specialization of function app so that the residual function consumes two elements in each recursive call; foldr_map and foldr_map2, two examples containing higher-order functions (the first one appears in Section 6.2); nondet, the example shown in Section 6.3; and, finally, kmp, the KMP-test of Section 6.4. The complete code of these benchmarks can be found within the implemented tool.

For each benchmark, Table I shows the cost variation of the main loop in the program together with the *actual* speedups, i.e., the ratio *original*/*residual*, where *original* is the runtime in the original program and *residual* is the runtime in the residual program. Speedups are shown for two Curry implementations: PAKCS (Hanus (ed.) et al., 2003) and the Münster Compiler (Lux, 2003). 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 symbolic costs to actual speedups—recall that they give only an approximation due to the use of a dependency graph—, some conclusions can be drawn. Deforestation examples usually follow the same pattern: costs U and C are usually halved (except for app_last, where C is reduced to a third). Actual speedups are not so impressive, but it seems a useful optimization. Benchmark app_2steps is only included to check that a reduction of cost U does not imply an actual speedup if the remaining costs—particularly C—do not decrease.

The transformation of higher-order functions into first-order ones gives significant speedups in benchmarks foldr_map and foldr_map2. Here, we can also find a common pattern: U is reduced to a third, C is halved and, more importantly, HO is reduced from 3 to 0.

Table I. Benchmark Results

	Original					Residual					Speedup	
Benchmark	U	C	A	HO	N	U	C	A	HO	N	PAKCS	Münster
all_ones	2	2	8	0	1	1	1	6	0	1	1.18	1.34
double_app	2	2	9	0	1	1	1	7	0	1	1.12	1.26
app_last	2	3	11	0	1	1	1	3	0	1	1.17	1.27
double_flip	2	2	12	0	1	1	1	8	0	1	1.15	1.13
length_app	2	2	8	0	1	1	1	6	0	1	1.11	1.18
app_2steps	2	2	6	0	1	1	2	6	0	1	1.04	0.98
foldr_map	3	2	11	3	1	1	1	8	0	1	2.02	2.33
foldr_map2	3	2	11	3	1	1	1	8	0	1	1.75	1.91
nondet	2	3	6	0	2	1	3	6	0	1	1.22	1.29
kmp	10	16	157	0	6	1	6	17	0	6	3.12	2.76

Runtimes for benchmark nondet are obtained by executing the initial call "`nondet x [1..300000]`". While this call is evaluated deterministically in the residual program, choice points are created for each recursive call in the original program. Although the actual speedups are modest, it can be explained by the fact that choice points contain only a few data (thus its creation is not so expensive). The greater speedup is in example kmp, where U, C and A are significantly reduced.

In summary, although all cost criteria are relevant for the execution time, some of them seem to have a greater impact. In particular, there are some costs whose reduction implies a runtime speedup even if the remaining costs are not reduced. This is the case of costs C, HO and N. In order to check this point, we have manually included some auxiliary functions in the residual programs of benchmarks all_ones, foldr_map and nondet so that U is not reduced anymore. In spite of this, runtime speedups remain the same. In contrast, benchmark app_2steps shows that a reduction of cost U alone gives no speedup (even a slowdown). Similarly, cost A seems to be more relevant for measuring memory allocation than runtime improvement.

7. Related Work

Amtoft (1991) establishes several properties of program transformations based on folding/unfolding in the context of logic programming. In particular, he proves that *superlinear* speedup cannot be accomplished by partial evaluation.[9] Andersen and Gomard (1992) develop a

[9] This result can also be found in (Andersen and Gomard, 1992).

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 Andersen and Gomard (1992). Sands (1995) introduces 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 by Sands (1995). However, he centers the discussion in the number of evaluation steps, while we consider additional cost criteria.

The closest approach, though, is the work by Albert et al. (2001), where a formal framework to measure the effectiveness of partial evaluation in functional logic languages is introduced. Indeed, the present work can be seen as a natural evolution of the ideas presented by Albert et al. (2001). The main differences are the following:

- Albert et al. (2001) consider the partial evaluation framework of Alpuente et al. (1999) based solely of needed narrowing. Therefore, their developments are not applicable to modern multi-paradigm languages like Curry whose operational semantics is based on a combination of needed narrowing and residuation.

- We define a formal specification of the cost-augmented semantics (and its residualized version) and prove their cost equivalence. Albert et al. (2001) show how the cost of a narrowing derivation can be computed, but there is no formalization of a "cost-augmented narrowing" relation.

- We prove that some costs (U, C, HO) are closed under instantiation, while other costs (A, N) may decrease when a call is further instantiated. This is an important property to relate the computed cost improvements with actual improvements at execution time.

Furthermore, we consider a partial evaluation scheme for flat programs based on a residualizing semantics—the RLNT calculus—to perform computations at partial evaluation time. These features are essential to define a partial evaluator which is applicable to realistic multi-paradigm languages.

8. Conclusions and Future Work

This work described the integration of quantitative aspects into the narrowing-driven partial evaluation framework. We introduced cost-augmented versions of both the standard and residualizing semantics

and proved their cost equivalence. We also presented the scheme of a cost-augmented partial evaluator and proved some useful properties. Experimental evaluation with an implementation of the enhanced partial evaluator for Curry programs shows the practicality of the approach.

A promising application of our developments is the generation of *cost-guided* partial evaluators. By experimenting with the developed partial evaluation tool, we discovered several common patterns associated with "successful" specializations. For instance, deforestation problems often reduce both the number of function unfoldings and the number of case evaluations 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. On the other hand, graphs of functional dependencies do not take into account the laziness of the underlying semantics. Therefore, another interesting line of research involves the definition of more refined speedup analyses which take into account the lazy nature of our computational model.

Acknowledgements

The author wishes to thank Elvira Albert, Sergio Antoy, Michael Hanus and Frank Huch for many helpful discussions on the topics of this work. I gratefully acknowledge the anonymous referees, also of PEPM 2002, for their detailed comments and suggestions.

References

Ager, M. S., O. Danvy, and H. Rohde: 2002, 'On Obtaining Knuth, Morris and Pratt's String Matcher by Partial Evaluation'. In: *Proc. of the ASIAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (ASIA-PEPM'02)*. pp. 32–46, ACM Press.

Albert, E., S. Antoy, and G. Vidal: 2001, '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 2000)*. pp. 103–124, Springer LNCS 2042.

Albert, E., M. Hanus, F. Huch, J. Olvier, and G. Vidal: 2002a, 'Operational Semantics for Functional Logic Languages'. In: *Proc. of the Int'l Workshop on Functional and (Constraint) Logic Programming (WFLP 2002)*, Vol. 76 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers.

Albert, E., M. Hanus, and G. Vidal: 2000, '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)*. pp. 381–398, Springer LNAI 1955.

Albert, E., M. Hanus, and G. Vidal: 2002b, 'A Practical Partial Evaluation Scheme for Multi-Paradigm Declarative Languages'. *Journal of Functional and Logic Programming* **2002**(1), 1–34.

Albert, E., M. Hanus, and G. Vidal: 2003, 'A Residualizing Semantics for the Partial Evaluation of Functional Logic Programs'. *Information Processing Letters* **85**(1), 19–25.

Albert, E. and G. Vidal: 2002a, 'The Narrowing-Driven Approach to Functional Logic Program Specialization'. *New Generation Computing* **20**(1), 3–26.

Albert, E. and G. Vidal: 2002b, 'Symbolic Profiling for Multi-paradigm Declarative Languages'. In: *Proc. of the 11th Int'l Workshop on Logic-Based Program Synthesis and Transformation (LOPSTR'01)*. pp. 148–167, Springer LNCS 2372.

Alpuente, M., M. Falaschi, P. Julián, and G. Vidal: 1997, 'Specialization of Lazy Functional Logic Programs'. In: *Proc. of the ACM SIGPLAN Conf. on Partial Evaluation and Semantics-Based Program Manipulation, PEPM'97*, Vol. 32, 12 of *Sigplan Notices*. New York, pp. 151–162, ACM Press.

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

Alpuente, M., M. Hanus, S. Lucas, and G. Vidal: 1999, 'Specialization of Inductively Sequential Functional Logic Programs'. In: *Proc. of the Fourth ACM SIGPLAN Int'l Conf. on Functional Programming (ICFP'99)*, Vol. 34.9 of *ACM Sigplan Notices*. pp. 273–283, ACM Press.

Amtoft, T.: 1991, 'Properties of Unfolding-based Meta-level Systems'. In: *Proc. of the ACM Symp. on Partial Evaluation and Semantics-based Program Transformation (PEPM'91)*. Sigplan Notices **26**(9), 243–254.

Andersen, L. and C. Gomard: 1992, 'Speedup Analysis in Partial Evaluation: Preliminary Results'. In: *Proc. of the ACM Workshop on Partial Evaluation and Semantics-based Program Transformation (PEPM'92)*. pp. 1–7, Yale University.

Antoy, S.: 1992, 'Definitional Trees'. In: *Proc. of the 3rd Int'l Conference on Algebraic and Logic Programming (ALP'92)*. pp. 143–157, Springer LNCS 632.

Antoy, S., R. Echahed, and M. Hanus: 2000, 'A Needed Narrowing Strategy'. *Journal of the ACM* **47**(4), 776–822.

Baader, F. and T. Nipkow: 1998, *Term Rewriting and All That*. Cambridge University Press.

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

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

Consel, C. and O. Danvy: 1989, 'Partial Evaluation of Pattern Matching in Strings'. *Information Processing Letters* **30**, 79–86.

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

Giovannetti, E., G. Levi, C. Moiso, and C. Palamidessi: 1991, 'Kernel Leaf: A Logic plus Functional Language'. *Journal of Computer and System Sciences* **42**, 363–377.

Hanus, M.: 1994, 'The Integration of Functions into Logic Programming: From Theory to Practice'. *Journal of Logic Programming* **19&20**, 583–628.

Hanus, M.: 1997, 'A Unified Computation Model for Functional and Logic Programming'. In: *Proc. of ACM Symp. on Principles of Programming Languages*. pp. 80–93, ACM, New York.

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

Hanus (ed.), M.: 2003, 'Curry: An Integrated Functional Logic Language'. Available at `http://www.informatik.uni-kiel.de/~mh/curry/`.

Hanus (ed.), M., S. Antoy, M. Engelke, K. Höppner, J. Koj, P. Niederau, R. Sadre, and F. Steiner: 2003, 'PAKCS 1.5.0: The Portland Aachen Kiel Curry System User Manual'. Technical report, University of Kiel, Germany.

Hortalá-González, T. and E. Ullán: 2001, '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)*. pp. 216–232, Springer LNCS 2024.

Huet, G. and J. Lévy: 1992, 'Computations in Orthogonal Rewriting Systems, Part I + II'. In: J. Lassez and G. Plotkin (eds.): *Computational Logic – Essays in Honor of Alan Robinson*. pp. 395–443.

Jones, N., C. Gomard, and P. Sestoft: 1993, *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Englewood Cliffs, NJ.

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

Lloyd, J.: 1994, 'Combining Functional and Logic Programming Languages'. In: *Proc. of the Int'l Logic Programming Symposium (ILPS'94)*. pp. 43–57.

Lloyd, J. and J. Shepherdson: 1991, 'Partial Evaluation in Logic Programming'. *Journal of Logic Programming* **11**, 217–242.

Loogen, R., F. López-Fraguas, and M. Rodríguez-Artalejo: 1993, 'A Demand Driven Computation Strategy for Lazy Narrowing'. In: *Proc. of PLILP'93*. pp. 184–200, Springer LNCS 714.

Lux, W.: 2003, 'Münster Curry v0.9.1: User's Guide'. Technical report, University of Münster, Germany.

Moreno-Navarro, J. and M. Rodríguez-Artalejo: 1992, 'Logic Programming with Functions and Predicates: The language Babel'. *Journal of Logic Programming* **12**(3), 191–224.

Peyton Jones, S. and A. Santos: 1998, 'A Transformation-Based Optimiser for Haskell'. *Science of Computer Programming* **32**(1-3), 3–47.

Reynolds, J.: 1998, 'Definitional interpreters for higher-order programming languages'. *Higher-Order and Symbolic Computation* **11**(4), 363–297. Reprinted from the proceedings of the 25th ACM National Conference (1972).

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

Sørensen, M. and R. Glück: 1995, 'An Algorithm of Generalization in Positive Supercompilation'. In: *Proc. of the Int'l Logic Programming Symposium (ILPS'95)*. pp. 465–479, The MIT Press, Cambridge, MA.

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

Vidal, G.: 2002, 'Cost-Augmented Narrowing-Driven Specialization'. In: *Proc. of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'02)*. pp. 52–62, ACM Press.

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

Warren, D. H. D.: 1982, 'Higher-Order Extensions to Prolog – Are they needed?'. In: M. Hayes-Roth and Pao (eds.): *Machine Intelligence*, Vol. 10. Ellis Horwood.

Appendix

A. Proofs

Theorem 1. Let e be an expression, e' a head normal form, and $\mathcal{R}$ a flat program. For each cost-augmented LNT derivation $\langle K_0, e\rangle \Rightarrow_\sigma^* \langle k', e'\rangle$ in $\mathcal{R}$ where rule hnf is not applied, there exists a cost-augmented RLNT derivation $\langle K_0, e\rangle \Rightarrow^* \langle k'', e''\rangle$ in $\mathcal{R}$ such that $\langle k'', e''\rangle \hookrightarrow_\sigma^* \langle k', e'\rangle \not\hookrightarrow$, and vice versa.

Proof. Our proof proceeds by relating the application of a rule in one calculus to the application of one or more rules in the other calculus. We prove a slightly more general claim: for each cost-augmented LNT derivation $\langle k, e\rangle \Rightarrow_\sigma^* \langle k', e'\rangle$ in $\mathcal{R}$ where rule hnf rule is not applied, there exists a cost-augmented RLNT derivation $\langle k, e\rangle \Rightarrow^* \langle k'', e''\rangle$ in $\mathcal{R}$ such that $\langle k'', e''\rangle \hookrightarrow_\sigma^* \langle k', e'\rangle \not\hookrightarrow$, and vice versa (i.e., the initial cost is not necessarily the empty cost K_0).

($\Rightarrow$) Consider a cost-augmented LNT derivation of the form $\langle k, e\rangle \Rightarrow_\sigma^* \langle k', e'\rangle$. We proceed by induction on the length l of this derivation.

Base case ($l = 0$). Trivial.

Inductive case ($l > 0$). Assume that the LNT derivation has the following form:

$$\langle k, e\rangle \Rightarrow_\theta \langle k^a, e^a\rangle \Rightarrow_\gamma^* \langle k', e'\rangle$$

where $\sigma = \gamma \circ \theta$. Now, we distinguish several cases depending on the rule applied in the first step:

(case_select) Since its definition is the same in both calculi, the claim follows trivially by induction.

(case_guess_det) Then, expression e has the form $fcase\ x\ of\ \{p_1 \to e_1\}$ and, thus, $e^a = \sigma_1(e_1)$ and $k^a = k[C \leftarrow C+1, A \leftarrow A+|p_1|+alloc(e_1)]$, where $\sigma_1 = \theta = \{x \mapsto p_1\}$. In the cost-augmented RLNT calculus, we can also apply rule case_guess_det to perform the step

$$\langle k, e\rangle \Rightarrow \langle alt(k^a), fcase\ x\ of\ \{p_1 \to e^a\}\rangle$$

Since $\langle k^a, e^a\rangle \Rightarrow_\gamma^* \langle k', e'\rangle$, by the inductive hypothesis, there exists a cost-augmented RLNT derivation $\langle k^a, e^a\rangle \Rightarrow^* \langle k^b, e^b\rangle$ with $\langle k^b, e^b\rangle \hookrightarrow_\gamma^* \langle k', e'\rangle \not\hookrightarrow$. Since $\langle k^a, e^a\rangle \Rightarrow^* \langle k^b, e^b\rangle$, by applying rule case_guess_eval repeatedly, we have the following cost-augmented RLNT derivation: $\langle alt(k^a), fcase\ x\ of\ \{p_1 \to e^a\}\rangle \Rightarrow^* \langle alt(k^b), fcase\ x\ of\ \{p_1 \to e^b\}\rangle = \langle k'', e''\rangle$. Thus, $\langle k, e\rangle \Rightarrow^* \langle k'', e''\rangle$. Finally, since $\langle k'', e''\rangle \hookrightarrow_\theta \langle k^b, e^b\rangle$

and $\langle k^b, e^b\rangle \hookrightarrow^*_\gamma \langle k', e'\rangle \not\hookrightarrow$ (by the inductive hypothesis), we have $\langle k'', e''\rangle \hookrightarrow^*_\sigma \langle k', e'\rangle \not\hookrightarrow$ and the claim follows.

(case_guess_nondet) In this case, e has the form $fcase\ x\ of\ \{\overline{p_m \to e_m}\}$ and, thus, $e^a = \sigma_i(e_i)$ and $k^a = k[C \leftarrow C+1, A \leftarrow A + |p_i| + alloc(e_i), N \leftarrow N+1]$, where $\sigma_i = \theta = \{x \mapsto p_i\}$, for some $i \in \{1, \ldots, m\}$. In the cost-augmented RLNT calculus, we can also apply rule case_guess_nondet and perform the step

$$\langle k, e\rangle \Rightarrow \langle alt(k^{a_1}, \ldots, k^{a_m}), fcase\ x\ of\ \{\overline{p_m \to \sigma_m(e_m)}\}\rangle$$

where $\sigma_j = \{x \mapsto p_j\}$ and $k^{a_j} = k[C \leftarrow C+1, A \leftarrow A + |p_j| + alloc(e_j), N \leftarrow N+1]$ for all $j = 1, \ldots, m$ (in particular, $k^{a_i} = k^a$). Since $\langle k^a, e^a\rangle \Rightarrow^*_\gamma \langle k', e'\rangle$, by the inductive hypothesis, there exists a cost-augmented RLNT derivation $\langle k^a, e^a\rangle \Rightarrow^* \langle k^b, e^b\rangle$ with $\langle k^b, e^b\rangle \hookrightarrow^*_\gamma \langle k', e'\rangle \not\hookrightarrow$. Since $\langle k^a, e^a\rangle \Rightarrow^* \langle k^b, e^b\rangle$, by applying rule case_guess_eval repeatedly, we have the following derivation: $\langle alt(k^{a_1}, \ldots, k^{a_i}, \ldots, k^{a_m}), fcase\ x\ of\ \{\overline{p_m \to \sigma_m(e_m)}\}\rangle \Rightarrow^* \langle alt(k^{a_1}, \ldots, k^b, \ldots, k^{a_m}), fcase\ x\ of\ \{\overline{p_m \to e'_m}\}\rangle = \langle k'', e''\rangle$ where $e'_j = \sigma_j(e_j)$ if $j \neq i$ and $e'_i = e^b$. Thus, $\langle k, e\rangle \Rightarrow^* \langle k'', e''\rangle$. Finally, since $\langle k'', e''\rangle \hookrightarrow_\theta \langle k^b, e^b\rangle$ and $\langle k^b, e^b\rangle \hookrightarrow^*_\gamma \langle k', e'\rangle \not\hookrightarrow$ (by the inductive hypothesis), we have $\langle k'', e''\rangle \hookrightarrow^*_\sigma \langle k', e'\rangle \not\hookrightarrow$ and the claim follows.

(case_eval) This case is immediate except when e has the form

$$(f)case\ (fcase\ x\ of\ \{\overline{p'_n \to e'_n}\})\ of\ \{\overline{p_m \to e_m}\}$$

(since, in the remaining cases, rule case_eval behaves identically in both calculi). Then, we consider two possibilities: $n = 1$ and $n > 1$. If $n = 1$, then we get

$$e^a = \sigma_1((f)case\ \sigma_1(e'_1)\ of\ \{\overline{p_m \to e_m}\}) = \sigma_1((f)case\ e'_1\ of\ \{\overline{p_m \to e_m}\})$$

with $\theta = \sigma_1 = \{x \mapsto p'_1\}$ and $k^a = k[C \leftarrow C+1, A \leftarrow A + |p'_1| + alloc(e'_1)]$, by one application of rule case_eval which, recursively, demands the application of rule case_guess_det. In the RLNT calculus, we can first apply rule case_of_case and perform the following step:

$$\langle k, e\rangle \Rightarrow \langle k^*, e^*\rangle = \langle k, fcase\ x\ of\ \{p'_1 \to (f)case\ e'_1\ of\ \{\overline{p_m \to e_m}\}\}\rangle$$

By applying rule case_guess_det, we can perform the RLNT step

$$\langle k^*, e^*\rangle \Rightarrow \langle alt(k^a), fcase\ x\ of\ \{p'_1 \to \sigma_1((f)case\ e'_1\ of\ \{\overline{p_m \to e_m}\})\}\rangle$$

Since $\langle k^a, e^a\rangle \Rightarrow^*_\gamma \langle k', e'\rangle$, by the inductive hypothesis, we have that $\langle k^a, e^a\rangle \Rightarrow^* \langle k^b, e^b\rangle$ where $\langle k^b, e^b\rangle \hookrightarrow^*_\gamma \langle k', e'\rangle \not\hookrightarrow$. Since $\langle k^a, e^a\rangle \Rightarrow^* \langle k^b, e^b\rangle$, by applying rule case_guess_eval repeatedly, we have the following RLNT derivation: $\langle alt(k^a), fcase\ x\ of\ \{p'_1 \to e^a\}\rangle \Rightarrow^* \langle alt(k^b), fcase$

$x\ of\ \{p'_1 \to e^b\}\rangle = \langle k'', e''\rangle$. Finally, since $\langle k'', e''\rangle \hookrightarrow_\theta \langle k^b, e^b\rangle$ and $\langle k^b, e^b\rangle \hookrightarrow^*_\gamma \langle k', e'\rangle \not\hookrightarrow$ (by the inductive hypothesis), we have $\langle k'', e''\rangle \hookrightarrow_\theta \langle k^b, e^b\rangle \hookrightarrow^*_\gamma \langle k', e'\rangle \not\hookrightarrow$ and the claim follows.

If $n > 1$, then we get

$e^a = \sigma_i((f)case\ \sigma_i(e'_i)\ of\ \{\overline{p_m \to e_m}\}) = \sigma_i((f)case\ e'_i\ of\ \{\overline{p_m \to e_m}\})$

with $\theta = \sigma_i = \{x \mapsto p'_i\}$ and $k^a = k[C \leftarrow C + 1, A \leftarrow A + |p'_i| + alloc(e'_i), N \leftarrow N + 1]$, for some $i \in \{1, \ldots, n\}$, by one application of rule case_eval which, recursively, demands the application of rule case_guess_nondet. Similarly to the previous case, we first apply rule case_of_case of the RLNT calculus to perform the following step:

$$\langle k, e\rangle \Rightarrow \langle k^*, e^*\rangle = \langle k, fcase\ x\ of\ \{\overline{p'_n \to (f)case\ e'_n\ of\ \{\overline{p_m \to e_m}\}}\}\rangle$$

By applying rule case_guess_nondet, we can perform the RLNT step

$$\langle k^*, e^*\rangle \Rightarrow \langle alt(\overline{k^{a_n}}), fcase\ x\ of\ \{\overline{p'_n \to \sigma_n((f)case\ e'_n\ of\ \{\overline{p_m \to e_m}\})}\}\rangle$$

with $\sigma_j = \{x \mapsto p'_j\}$ and $k^{a_j} = k[C \leftarrow C + 1, A \leftarrow A + |p'_j| + alloc(e'_j), N \leftarrow N + 1]$ for all $j = 1, \ldots, n$. In particular, $k^{a_i} = k^a$. Since $\langle k^a, e^a\rangle \Rightarrow^*_\gamma \langle k', e'\rangle$, by the inductive hypothesis, we have that $\langle k^a, e^a\rangle \Rightarrow^* \langle k^b, e^b\rangle$ where $\langle k^b, e^b\rangle \hookrightarrow^*_\gamma \langle k', e'\rangle \not\hookrightarrow$. Since $\langle k^a, e^a\rangle \Rightarrow^* \langle k^b, e^b\rangle$, by applying rule case_guess_eval repeatedly, we have the following RLNT derivation:

$$\begin{aligned}&\langle alt(\ldots, k^{a_i}, \ldots), fcase\ x\ of\ \{\overline{p'_n \to \sigma_n((f)case\ e'_n\ of\ \{\overline{p_m \to e_m}\})}\}\rangle\\ &\quad \Rightarrow^* \langle alt(\ldots, k^b, \ldots), fcase\ x\ of\ \{\overline{p'_n \to e''_n}\}\rangle = \langle k'', e''\rangle\end{aligned}$$

where $e''_j = \sigma_j((f)case\ e'_j\ of\ \{\overline{p_m \to e_m}\})$ if $j \neq i$ and $e''_i = e^b$. Thus, $\langle k, e\rangle \Rightarrow^* \langle k'', e''\rangle$. Finally, since $\langle k'', e''\rangle \hookrightarrow_\theta \langle k^b, e^b\rangle$ and $\langle k^b, e^b\rangle \hookrightarrow^*_\gamma \langle k', e'\rangle \not\hookrightarrow$ (by the inductive hypothesis), we have $\langle k'', e''\rangle \hookrightarrow_\theta \langle k^b, e^b\rangle \hookrightarrow^*_\gamma \langle k', e'\rangle \not\hookrightarrow$ and the claim follows.

(fun_eval, apply_total, apply_partial, apply_eval) The claim follows trivially by induction since their definitions are the same in both calculi.

($\Leftarrow$) Consider a cost-augmented RLNT derivation $\langle e, k\rangle \Rightarrow^* \langle k'', e''\rangle$ with $\langle k'', e''\rangle \hookrightarrow^*_\sigma \langle k', e'\rangle \not\hookrightarrow$. We proceed by induction on the sum l of the length of the RLNT derivation plus the length of the $\hookrightarrow$ derivation.

Base case ($l = 0$). Trivial.

Inductive case ($l > 0$). Assume that the cost-augmented RLNT derivation has the form

$$\langle k, e\rangle \Rightarrow \langle k^a, e^a\rangle \Rightarrow^* \langle k'', e''\rangle \text{ and } \langle k'', e''\rangle \hookrightarrow^*_\sigma \langle k', e'\rangle \not\hookrightarrow$$

Now, we distinguish several the following cases depending on the rule applied in the first step:

(case_select) Since its definition is the same in both calculi, the claim follows trivially by induction.

(case_guess_det) Then, e has the form $(f)case\ x\ of\ \{p_1 \to e_1\}$. Note that, if the case expression was rigid, e' could not be a head normal form since $\hookrightarrow$ could not remove the (residualized) outer case expression. Thus, e should definitively be rooted by a flexible case. Then, by applying rule case_guess_det, we get

$$\langle alt(k^a), fcase\ x\ of\ \{p_1 \to \sigma_1(e_1)\}\rangle = \langle alt(k^a), fcase\ x\ of\ \{p_1 \to e^a\}\rangle$$

where $\sigma_1 = \{x \mapsto p_1\}$ and $k^a = k[C \leftarrow C+1, A \leftarrow A+|p_1|+alloc(e_1)]$. Then, in the LNT calculus, we can also perform the following step with rule case_guess_det: $\langle k, e\rangle \Rightarrow_{\sigma_1} \langle k^a, e^a\rangle$.

In the RLNT derivation, we have $\langle alt(k^a), fcase\ x\ of\ \{p_1 \to e^a\}\rangle \Rightarrow^* \langle k'', e''\rangle$ where $k'' = alt(k^b)$ and $e'' = fcase\ x\ of\ \{p_1 \to e^b\}$ for some k^b and e^b; this is a consequence of the fact that the RLNT calculus cannot modify the (residualized) outermost case structure. Moreover, $\langle k'', e''\rangle \hookrightarrow^*_\sigma \langle k', e'\rangle$. By definition of $\hookrightarrow$, this derivation should have the form $\langle k'', e''\rangle \hookrightarrow_\theta \langle k^b, e^b\rangle \hookrightarrow^*_\gamma \langle k', e'\rangle$ with $\theta = \sigma_1$. Trivially, we have the RLNT derivation $\langle k^a, e^a\rangle \Rightarrow^* \langle k^b, e^b\rangle$ with $\langle k^b, e^b\rangle \hookrightarrow^*_\gamma \langle k', e'\rangle$. By the inductive hypothesis, we have $\langle k^a, e^a\rangle \Rightarrow^*_\gamma \langle k', e'\rangle$ and the claim follows.

(case_guess_nondet) By a similar argument as before, e should be rooted by a flexible case, i.e., it has the form $fcase\ x\ of\ \{\overline{p_m \to e_m}\}$. Then, by applying rule case_guess_nondet, we get the RLNT step

$$\langle k, e\rangle \Rightarrow \langle alt(k^{a_1}, \ldots, k^{a_m}), fcase\ x\ of\ \{\overline{p_m \to \sigma_m(e_m)}\}\rangle$$

where $\sigma_j = \{x \mapsto p_j\}$ and $k^{a_j} = k[C \leftarrow C+1, A \leftarrow A+|p_j| + alloc(e_j), N \leftarrow N+1]$ for all $j = 1, \ldots, m$. Now, we look at the $\hookrightarrow$ derivation, since it will determine the corresponding LNT step. Let us assume that $\langle k'', e''\rangle \hookrightarrow_\theta \langle k^b, e^b\rangle \hookrightarrow^*_\gamma \langle k', e'\rangle$ holds, with $\sigma = \gamma \circ \theta$. Since the outermost case has been residualized in the former step, then

$$\langle k'', e''\rangle = \langle alt(k^{b_1}, \ldots, k^{b_m}), fcase\ x\ of\ \{\overline{p_1 \to e^{b_m}}\}\rangle$$

where $k^{b_i} = k^b$, $e^{b_i} = e^b$ and $\theta = \sigma_i$ for some $i \in \{1, \ldots, m\}$. Similarly, in the LNT calculus, we can perform the following step:

$$\langle k, e\rangle \Rightarrow_{\sigma_i} \langle k^{a_i}, \sigma_i(e_i)\rangle$$

by selecting the same branch of the outermost case expression, where $k^{a_i} = k[C \leftarrow C+1, A \leftarrow A+|p_i|+alloc(e_i), N \leftarrow N+1]$. Since

$$\langle alt(k^{a_1},\ldots,k^{a_m}), fcase\ x\ of\ \{\overline{p_m \to \sigma_m(e_m)}\}\rangle \Rightarrow^* \langle alt(k^{b_1},\ldots,k^{b_m}), fcase\ x\ of\ \{\overline{p_1 \to e^{b_m}}\}\rangle$$

then $\langle k^{a_i}, \sigma_i(e_i)\rangle \Rightarrow^* \langle k^{b_i}, e^{b_i}\rangle = \langle k^b, e^b\rangle$ with $\langle k^b, e^b\rangle \hookrightarrow^*_\gamma \langle k', e'\rangle$. Therefore, by the inductive hypothesis, we have the following LNT derivation: $\langle k^{a_i}, \sigma_i(e_i)\rangle \Rightarrow^*_\gamma \langle k', e'\rangle$, and the claim follows.

(case_guess_eval) This case follows trivially by the inductive hypothesis since rule case_guess_eval is only used to (recursively) evaluate the different branches of a residualized case expression.

(case_of_case) In this case, expression e must have the form

$$(f)case\ ((f)case\ x\ of\ \{\overline{p'_n \to e'_n}\})\ of\ \{\overline{p_m \to e_m}\}$$

By applying rule case_of_case, we get:

$$e^a = (f)case\ x\ of\ \{\overline{p'_n \to (f)case\ e'_n\ of\ \{\overline{p_m \to e_m}\}}\}$$

with the same associated cost k. Now, the same considerations of the case of rule case_guess_det apply; thus the outer case expression must be flexible. Now, we distinguish two possibilities: $n = 1$ and $n > 1$. If $n = 1$, then we apply the RLNT rule case_guess_det:

$$\langle k, e^a\rangle \Rightarrow \langle alt(k^a), fcase\ x\ of\ \{p'_1 \to (f)case\ \sigma_1(e'_1)\ of\ \{\overline{p_m \to e_m}\}\}\rangle$$

with $\sigma_1 = \{x \mapsto p'_1\}$ and $k^a = k[C \leftarrow C + 1, A \leftarrow A + |p'_1| + alloc(e'_1)]$. Then, the proof proceeds analogously to the case of rule case_guess_det.

If $n > 1$, we perform the following RLNT step with case_guess_nondet:

$$\langle k, e^a\rangle \Rightarrow \langle alt(k^{a_1},\ldots,k^{a_n}), fcase\ x\ of\ \{\overline{p'_n \to (f)case\ \sigma_j(e'_n)\ of\ \{\overline{p_m \to e_m}\}}\}\rangle$$

with $\sigma_j = \{x \mapsto p'_j\}$ and $k^{a_j} = k[C \leftarrow C + 1, A \leftarrow A + |p'_j| + alloc(e'_j), N \leftarrow N + 1]$ for all $j = 1,\ldots,n$. The the proof proceeds analogously to the case of rule case_guess_nondet.

(case_eval, fun_eval, apply_total, apply_partial, apply_eval) In these cases, the claim follows trivially by induction since their definitions are equivalent in both calculi. □

Theorem 2. Let e be an expression with $\langle K_0, e\rangle \Rightarrow^*_\sigma \langle k_1, e'\rangle$. Let $\theta(e)$ be a constructor instance of e with $\langle K_0, \theta(e)\rangle \Rightarrow^*_{\sigma'} \langle k_2, e''\rangle$, where e'' is a constructor instance of e' and $\sigma' \leq \sigma$. Then $U(k_1) = U(k_2)$, $C(k_1) = C(k_2)$, $A(k_1) \geq A(k_2)$, $HO(k_1) = HO(k_2)$, and $N(k_1) \geq N(k_2)$.

Proof. As in the proof of Theorem 1, we prove a slightly more general claim: given two costs k_1 and k_2 with $U(k_1) = U(k_2)$, $C(k_1) = C(k_2)$, $A(k_1) \geq A(k_2)$, $HO(k_1) = HO(k_2)$, and $N(k_1) \geq N(k_2)$ and the cost-augmented LNT derivations $\langle k_1, e\rangle \Rightarrow^*_\sigma \langle k'_1, e'\rangle$ and $\langle k_2, \theta(e)\rangle \Rightarrow^*_{\sigma'} \langle k'_2, e''\rangle$, where e'' is a constructor instance of e' and $\sigma' \leq \sigma$, we have that $U(k'_1) = U(k'_2)$, $C(k'_1) = C(k'_2)$, $A(k'_1) \geq A(k'_2)$, $HO(k'_1) = HO(k'_2)$, and $N(k'_1) \geq N(k'_2)$ (i.e., the initial costs need not be the same but should be in the desired relation). We prove the claim by induction on the number s of steps in the former derivation.

Base case ($s = 0$). Trivial.

Inductive case ($s > 0$). Here, we assume that the LNT derivation has the form: $\langle k_1, e\rangle \Rightarrow_\delta \langle k''_1, e''\rangle \Rightarrow^*_\gamma \langle k'_1, e'\rangle$, where $\sigma = \gamma \circ \delta$. Now, we make a case distinction depending on the rule applied in the first step:

(hnf) This case is trivial since constructor symbols in the expression to be evaluated do not have an associated cost. Rule hnf is only used to recursively apply one of the remaining rules.

(case_select) Here, the expression e should have the form

$$e = (f)case\ c(\overline{e_n})\ of\ \{\overline{p_m \to e'_m}\}$$

where $p_i = c(\overline{x_n})$ for some $i \in \{1, \ldots, m\}$. Hence, the first LNT step is: $\langle k_1, e\rangle \Rightarrow_{id} \langle k''_1, e''\rangle$, where $k''_1 = k_1[C \leftarrow C + 1, A \leftarrow A + alloc(e'_i)]$, $e'' = \delta(e'_i)$, and $\delta = \{\overline{x_n \mapsto e_n}\}$. Then, $\theta(e)$ should have the form:

$$\theta(e) = (f)case\ \theta(c(\overline{e_n}))\ of\ \{\overline{p_m \to \theta(e'_m)}\}$$

Therefore, rule case_select can also be applied to the state $\langle k_2, \theta(e)\rangle$ so that the following LNT step can be done: $\langle k_2, \theta(e)\rangle \Rightarrow_{id} \langle k''_2, e^*\rangle$, where $k''_2 = k_2[C \leftarrow C + 1, A \leftarrow A + alloc(\theta(e'_i))]$, $e^* = \delta'(\theta(e'_i))$, and $\delta' = \{\overline{x_n \mapsto \theta(e_n)}\} = \delta \circ \theta$. Trivially, $U(k''_1) = U(k''_2)$, $C(k''_1) = C(k''_2)$, $A(k''_1) \geq A(k''_2)$, $HO(k''_1) = HO(k''_2)$, and $N(k''_1) \geq N(k''_2)$, since cost variable C is incremented in both cases, $alloc(e'_i) = alloc(\theta(e'_i))$ by definition, and the remaining cost variables are not modified. Moreover, $\delta'(e'_i) = \theta(\delta(e'_i))$. Therefore, the claim follows by induction.

(case_guess_det) In this case, expression e should have the form

$$e = fcase\ x\ of\ \{p_1 \to e'_1\}$$

Hence, the first LNT step is as follows: $\langle k_1, e\rangle \Rightarrow_\delta \langle k_1'', e''\rangle$, where $k_1'' = k_1[C \leftarrow C+1, A \leftarrow A+|p_1|+alloc(e_1')]$, $e'' = \delta(e_1')$, and $\delta = \{x \mapsto p_1\}$. On the other hand, $\theta(e)$ should have the form

$$\theta(e) = (f)case\ \theta(x)\ of\ \{p_1 \rightarrow \theta(e_1')\}$$

Now, we distinguish two cases depending on whether $\theta(x)$ is a variable or not. If $\theta(x)$ is a variable, then the claim follows trivially by induction. If $\theta(x) = c(\overline{d_n})$ (with $\overline{d_n}$ constructor terms), then $p_1 = c(\overline{x_n})$ (otherwise, the corresponding LNT derivation could not be done). Now, the following LNT step can be performed by applying rule case_select: $\langle k_2, \theta(e)\rangle \Rightarrow_{id} \langle k_2'', e^*\rangle$, where $k_2'' = k_2[C \leftarrow C+1, A \leftarrow A+alloc(\theta(e_1'))]$, $e^* = \delta'(\theta(e_1'))$, and $\delta' = \{\overline{x_n \mapsto d_n}\}$. Trivially, $U(k_1'') = U(k_2'')$, $C(k_1'') = C(k_2'')$, $A(k_1'') \geq A(k_2'')$, $HO(k_1'') = HO(k_2'')$, and $N(k_1'') \geq N(k_2'')$, since cost variable C is incremented in both cases, $alloc(e_1') = alloc(\theta(e_1'))$ by definition, $|p_1|$ is only added to cost k_1, and the remaining cost variables are not modified. Moreover, since $\theta(x) = c(\overline{d_n})$ and $\delta = \{x \mapsto c(\overline{x_n})\}$, we have that $\delta'(\theta(e_1'))$ is a constructor instance of $\delta(e_1')$. Therefore, the claim follows by induction.

(case_guess_nondet) This case can be proved analogously to the previous case. The only difference is that cost variable N is always incremented in the first LNT derivation while it depends on the value of substitution θ in the second derivation. If $\theta(x)$ is a variable (where $e = fcase\ x\ of\ \{\overline{p_m \rightarrow e_m}\}$), then cost variable N is incremented in both derivations. If $\theta(x)$ is a constructor term, then N is only incremented in the first derivation. Thus, the claim follows easily.

(case_eval) Similarly to the case of rule hnf, this case follows trivially since rule case_eval is only used to recursively apply one of the remaining rules.

(apply_total,apply_eval) These cases follow by induction since the instantiation of some variables to constructor terms does not affect to the application of these rules (only symbols *apply* and *partcall* matter).

(apply_eval) The same considerations of rule case_eval apply. □