In *Proc. of the 6th International Conference on Logic for Programming and Automated Reasoning, LPAR'99*, Tbilisi (Georgia).
Springer LNCS 1705, pp. 376–395, 1999

A Partial Evaluation Framework for Curry Programs*

Elvira Albert[1], María Alpuente[1], Michael Hanus[2], and Germán Vidal[1]

[1] DSIC, UPV, Camino de Vera s/n, E-46022 Valencia, Spain
{ealbert,alpuente,gvidal}@dsic.upv.es
[2] Informatik II, RWTH Aachen, D-52056, Germany
hanus@informatik.rwth-aachen.de

Abstract. In this work, we develop a partial evaluation technique for *residuating functional logic* programs, which generalize the concurrent computation models for logic programs with delays to functional logic programs. We show how to lift the nondeterministic choices from run time to specialization time. We ascertain the conditions under which the original and the transformed program have the same answer expressions for the considered class of queries as well as the same floundering behavior. All these results are relevant for program optimization in Curry, a functional logic language which is intended to become a standard in this area. Preliminary empirical evaluation of the specialized Curry programs demonstrates that our technique also works well in practice and leads to substantial performance improvements. To our knowledge, this work is the first attempt to formally define and prove correct a general scheme for the partial evaluation of functional logic programs with delays.

1 Introduction

The last few years have witnessed a maturity in the area of multiparadigm declarative languages in order to combine the most important features of functional programming (nested expressions, efficient demand-driven functional computations), logic programming (logical variables, partial data structures, constraints, built-in search), and concurrent programming (concurrent computations with synchronization on logical variables). The computation model of such integrated languages is based on a seamless combination of two different operational principles: narrowing and residuation.

The *residuation* principle is based on the idea of delaying function calls until they are ready for deterministic evaluation. Residuation preserves the deterministic nature of functions and naturally supports concurrent computations. Unfortunately, it is unable to compute solutions if arguments of functions are not sufficiently instantiated during the computation, though program analysis methods exist which provide sufficient criteria for the completeness of residuation [11,

* This work has been partially supported by CICYT TIC 98-0445-C03-01, by Acción Integrada hispano-alemana HA1997-0073, and by the German Research Council (DFG) under grant Ha 2457/1-1.

19]. Residuating functional logic languages employ dynamic scheduling similarly to modern (constraint) logic programming languages, where some calls are dynamically delayed until their arguments are sufficiently instantiated to allow the call to run efficiently. Residuation is the basis for implementing many concurrent (constraint) programming languages such as Oz [32] and is also used in other multiparadigm declarative languages such as Escher [25, 26], Le Fun [2], Life [1], and NUE-Prolog [31].

On the other hand, the *narrowing* mechanism allows the instantiation of variables in expressions and then applies reduction steps to the function calls of the instantiated expression. This instantiation is usually computed by unifying a subterm of the expression with the left-hand side of some program rule. Narrowing provides completeness in the sense of logic programming —computation of all solutions— as well as functional programming —computation of values— (see [18] for a survey). To avoid unnecessary computations and to deal with infinite data structures, demand-driven generation of the search space has recently been advocated by a flurry of outside-in, lazy narrowing strategies (see, e.g., [10, 16, 28, 29]). Due to its optimality properties w.r.t. the length of derivations and the number of computed solutions, *needed narrowing* [10] is currently the best lazy narrowing strategy for functional logic programs.

Curry is a modern multiparadigm declarative language which combines functional, logic and concurrent programming styles by unifying (needed) narrowing and residuation into a single model [20, 21]. To support coroutining, the model provides for *suspension* of function calls if a *demanded* argument is not sufficiently instantiated. Similarly to recent residuation-based languages like Escher [25] or Oz [32], Curry represents (don't know) non-deterministic choices by explicit disjunctions, in contrast to narrowing which is usually defined with implicit disjunctions as in classical logic programming. The precise mechanism (narrowing or residuation) for each function is specified by *evaluation annotations*, which are similar to coroutining declarations in Prolog [30], where the programmer specifies conditions under which a call is ready for a resolution step. Deterministic functions are declared *rigid* (which forces delayed evaluation by rewriting), while non-deterministic functions are declared *flex* (which enables narrowing steps). By default, only predicates (i.e., Boolean functions) are considered flexible, while all other functions are rigid, but the user can easily provide different evaluation annotations. The computation domain considers disjunctions of (*answer* $[\![$ *expression*) pairs in order to reflect not only the computed values but also the different variable bindings. The following example illustrates the integrated model (the computation steps are denoted by $\xrightarrow{\texttt{RN}}$ as in [20]).

Example 1. Consider the following rules defining the less-or-equal function "$\leqslant$" and the addition "+" on natural numbers (built from 0 and s):

$$
\begin{array}{rcllcrcl}
\texttt{0} \leqslant \texttt{N} & \to & \texttt{true} & \qquad & \texttt{0} + \texttt{X} & \to & \texttt{X} \\
\texttt{s(M)} \leqslant \texttt{0} & \to & \texttt{false} & & \texttt{s(X)} + \texttt{Y} & \to & \texttt{s(X + Y)} \\
\texttt{s(M)} \leqslant \texttt{s(N)} & \to & \texttt{M} \leqslant \texttt{N} & & & &
\end{array}
$$

where "$\leqslant$" is rigid and "$+$" is flexible. Then, the following goal is evaluated by freezing and awakening the function call to "$\leqslant$" (the subterm evaluated in the next step is underlined):[1]

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathtt{id} \,[\!]\, \mathtt{X} \leqslant \mathtt{Y} \;\&\; \underline{\mathtt{X}+\mathtt{0}} \doteq \mathtt{0}} \\
\stackrel{\text{RN}}{\longrightarrow} & \{\mathtt{X}=\mathtt{0}\} \,[\!]\, \underline{\mathtt{0} \leqslant \mathtt{Y}} \;\&\; \mathtt{0} \doteq \mathtt{0} \;\vee\; \{\mathtt{X}=\mathtt{s(Z)}\} \,[\!]\, \mathtt{s(Z)} \leqslant \mathtt{Y} \;\&\; \mathtt{s(Z+0)} \doteq \mathtt{0} \\
\stackrel{\text{RN}}{\longrightarrow} & \{\mathtt{X}=\mathtt{0}\} \,[\!]\, \mathtt{true} \;\&\; \mathtt{0} \doteq \mathtt{0} \;\vee\; \{\mathtt{X}=\mathtt{s(Z)}\} \,[\!]\, \mathtt{s(Z)} \leqslant \mathtt{Y} \;\&\; \underline{\mathtt{s(Z+0)} \doteq \mathtt{0}} \\
\stackrel{\text{RN}}{\longrightarrow} & \{\mathtt{X}=\mathtt{0}\} \,[\!]\, \mathtt{true} \;\&\; \underline{\mathtt{0} \doteq \mathtt{0}} \\
\stackrel{\text{RN}}{\longrightarrow} & \{\mathtt{X}=\mathtt{0}\} \,[\!]\, \underline{\mathtt{true} \;\&\; \mathtt{true}} \\
\stackrel{\text{RN}}{\longrightarrow} & \{\mathtt{X}=\mathtt{0}\} \,[\!]\, \mathtt{true}\,.
\end{array}
$$

Note that the second disjunction fails since $\mathtt{s(Z+0)} \doteq \mathtt{0}$ is unsolvable.

Partial evaluation (PE) has been established as an important research topic in both the functional [12, 22] and logic programming [15, 27] communities. Although the objectives are similar (typically, the specialization of a given program w.r.t. part of its input data), the general methods are often different due to the distinct underlying models and the different perspectives (see [6] for a detailed comparison). This separation has the negative consequence of duplicated work since developments are not shared and many similarities are overlooked.

Narrowing-driven PE [6] is the first generic algorithm for the specialization of functional logic programs. This framework provides the same potential for specialization as powerful (on-line) PE methods for logic programs (e.g., *conjunctive partial deduction* [24]) as well as functional programs (e.g., *positive supercompilation* [17]). The work in [7] formalizes an instance of the narrowing-driven PE method for inductively sequential programs based on needed narrowing. It lifts to the PE level the idea of only evaluating code when it is necessary. An attractive property of this instance is that it preserves the (inductively sequential) structure of the original program, and hence the same execution mechanism (namely, needed narrowing) can be safely used after the specialization. This property does not generally hold for other instances of the PE framework (see [7]).

The aim of this paper is to develop a partial evaluator for (kernel) Curry programs. Unfortunately, the approach of [7] is not powerful enough, since it follows the framework of [6] which does not consider the residuation principle. Hence, we generalize the original framework in order to deal with (inductively sequential) programs containing evaluation annotations for program functions. This task is difficult for several reasons. Firstly, a naïve adaptation of [7] in which floundering computations are simply stopped during PE is not adequate, since a poor specialization would be obtained in most cases and could even be unsafe in our setting (see Example 3). Thus, we introduce an extension of the standard computation model which allows us to ignore evaluation annotations during PE while still guaranteeing correctness. As a consequence, our method is less restrictive than many existing methods for (constraint) logic programs with

[1] Here & is the *concurrent conjunction operator*, i.e., the expression e_1 & e_2 is reduced by reducing either e_1 or e_2, and $\doteq$ is the *strict* equality predicate.

delays, in which suspended expressions cannot be unfolded (e.g., [14]). Secondly, the inference of safe evaluation annotations for the partially evaluated programs is far from trivial (see Example 4). In particular, we are forced to split resultants into several auxiliary (intermediate) functions in some cases to correctly preserve the answer expressions as well as the floundering behaviour.

The main contributions of this work can be summarized as follows. We provide (total) correctness results for the transformation, including the equivalence between the original and specialized programs w.r.t. floundering-freeness. This can be used for proving completeness of residuation for the considered class of goals in the original program by analyzing the floundering behavior of the resulting program. In particular, proving floundering-freeness for the specialized program is in many cases trivial (or easier than in the original program) because partial evaluation can transform a rigid function into a flexible one (whenever the specialized call is already sufficiently instantiated), but not vice versa. Moreover, we also prove that the transformation preserves the (inductively sequential) structure of programs.

The structure of the paper is as follows. After some basic definitions in Sect. 2, in Sect. 3 we recall the formal definition of needed narrowing and residuation. A PE scheme for residuating functional logic programs is formalized in Sect. 4, together with a method to properly synthesize evaluation annotations for specialized functions. We also provide results about the structure of specialized programs and the total correctness of the transformation. Section 5 shows the practical importance of our specialization techniques by means of some examples and Sect. 6 concludes. More details and proofs of technical results can be found in [4].

2 Preliminaries

We assume familiarity with basic notions of term rewriting [13] and functional logic programming [18]. 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 $\mathcal{X}$ are denoted by $\mathcal{T}(\mathcal{C} \cup \mathcal{F}, \mathcal{X})$ and $\mathcal{T}(\mathcal{C}, \mathcal{X})$, respectively. 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 term is *linear* if it does not contain multiple occurrences of one variable. We write $\overline{o_n}$ for the *list of objects* $o_1, \ldots, o_n$.

A *pattern* is a term of the form $f(\overline{d_n})$ where $f/n \in \mathcal{F}$ and $d_1, \ldots, d_n \in \mathcal{T}(\mathcal{C}, \mathcal{X})$. A term is *operation-rooted* if it has an operation symbol at the root. $root(t)$ denotes the symbol at the root of the term t. 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). Positions are ordered by the *prefix* ordering: $u \leq v$, if there exists w such that $u.w = v$. Given a term t, we let $\mathcal{P}os(t)$ and $\mathcal{FP}os(t)$ denote the set of positions and the set of nonvariable positions of t, respectively.

$t|_p$ 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 (see [13] for details).

We denote by $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ the *substitution* σ with $\sigma(x_i) = t_i$ for $i = 1, \ldots, n$ (with $x_i \neq x_j$ if $i \neq j$), and $\sigma(x) = x$ for all other variables x. The set $\mathcal{D}om(\sigma) = \{x \in \mathcal{X} \mid \sigma(x) \neq x\}$ is called the *domain* of σ. A substitution σ is *(ground) constructor*, if $\sigma(x)$ is (ground) constructor for all $x \in \mathcal{D}om(\sigma)$. The identity substitution is denoted by id. Given a substitution θ and a set of variables $V \subseteq \mathcal{X}$, we denote by $\theta_{\restriction V}$ the substitution obtained from θ by restricting its domain to V. We write $\theta = \sigma\ [V]$ if $\theta_{\restriction V} = \sigma_{\restriction V}$, and $\theta \leq \sigma\ [V]$ denotes the existence of a substitution γ such that $\gamma \circ \theta = \sigma\ [V]$. A term t' is an *instance* of t if there is a substitution σ with $t' = \sigma(t)$. This implies a *subsumption ordering* on terms which is defined by $t \leq t'$ iff t' is an instance of t.

A set of rewrite rules $l \to r$ such that $l \notin \mathcal{X}$, and $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$ is called a *term rewriting system* (TRS). The terms l and r are called the *left-hand side* (*lhs*) and the *right-hand side* (*rhs*) of the rule, respectively. A TRS $\mathcal{R}$ is left-linear if l is linear for all $l \to r \in \mathcal{R}$. A TRS is constructor–based (CB) if each lhs l is a pattern. In the remainder of this paper, a functional logic *program* is a left-linear CB-TRS. A *rewrite step* is an application of a rewrite rule to a term, i.e., $t \to_{p,R} s$ if there exists a position p in t, a rewrite rule $R = l \to r$ and a substitution σ with $t|_p = \sigma(l)$ and $s = t[\sigma(r)]_p$.

To evaluate terms containing variables, narrowing non-deterministically instantiates the variables such 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 denote by $t_0 \leadsto^*_\sigma t_n$ a sequence of narrowing steps $t_0 \leadsto_{\sigma_1} \ldots \leadsto_{\sigma_n} t_n$ with $\sigma = \sigma_n \circ \cdots \circ \sigma_1$. Due to the presence of free variables, an expression may be reduced to different values after instantiating free variables to different terms. In functional programming, one is interested in the computed *value* whereas logic programming emphasizes the different bindings (*answers*). Thus, for our integrated framework we define an *answer expression* as a pair $\sigma \,[\!]\, e$ consisting of a substitution σ (the answer computed so far) and an expression e. An answer expression $\sigma \,[\!]\, e$ is *solved* if e is a constructor term, otherwise it is *unsolved.* Since more than one answer may exist for expressions containing free variables, expressions are reduced to disjunctions of answer expressions. A *disjunctive expression* is a (multi-)set of answer expressions $\{\sigma_1 \,[\!]\, e_1, \ldots, \sigma_n \,[\!]\, e_n\}$, sometimes written as $(\sigma_1 \,[\!]\, e_1) \vee \ldots \vee (\sigma_n \,[\!]\, e_n)$. The set of all disjunctive expressions is denoted by $\mathcal{D}$.

The evaluation to ground constructor terms (and not to arbitrary expressions) is the intended semantics of functional languages and also of most functional logic languages. In particular, the equality predicate $\doteq$ used in some examples is defined (as in functional languages) as the *strict equality* on terms:

$$\begin{array}{rl} \mathtt{c} \doteq \mathtt{c} \to \mathtt{true} & \%\ \mathtt{c/0} \in \mathcal{C} \\ \mathtt{c(X_1,\ldots,X_n)} \doteq \mathtt{c(Y_1,\ldots,Y_n)} \to \mathtt{(X_1 \doteq Y_1)}\ \&\ \ldots\ \&\ \mathtt{(X_n \doteq Y_n)} & \%\ \mathtt{c/n} \in \mathcal{C} \end{array}$$

Thus we do not treat the strict equality in any special way, and it is sufficient to consider it as a Boolean function which must be reduced to the constant `true`.

3 A Unified Computation Model for FL Programs with Delays

The definition of needed narrowing [10] and its extension to concurrent programming [20] is based on definitional trees which have been introduced by Antoy [8] for the specification of efficient rewrite strategies. A definitional tree is a hierarchical structure containing all rules of a defined function. $\mathcal{T}$ is a *definitional tree with pattern* π iff the depth of $\mathcal{T}$ is finite and one of the following cases holds:

$\mathcal{T} = rule(\pi \to r)$, where $\pi \to r$ is a variant of a rule.
$\mathcal{T} = branch(\pi, o, r, \mathcal{T}_1, \ldots, \mathcal{T}_k)$, where o is an occurrence of a variable in π, $r \in \{rigid, flex\}$, $c_1, \ldots, c_k$ are different constructors of the sort of $\pi|_o$, for some $k > 0$, and, for all $i = 1, \ldots, k$, $\mathcal{T}_i$ is a definitional tree with pattern $\pi[c_i(x_1, \ldots, x_n)]_o$, where n is the arity of c_i and $x_1, \ldots, x_n$ are new variables.

A *definitional tree of an n-ary function* f is a definitional tree $\mathcal{T}$ with pattern $f(x_1, \ldots, x_n)$, where $x_1, \ldots, x_n$ are distinct variables, such that for each rule $l \to r$ with $l = f(t_1, \ldots, t_n)$ there is a node $rule(l' \to r')$ in $\mathcal{T}$ with l variant of l'. In the following, we write $pattern(\mathcal{T})$ for the pattern of a definitional tree $\mathcal{T}$. A defined function is called *inductively sequential* if it has a definitional tree. A rewrite system $\mathcal{R}$ is called *inductively sequential* if all its defined functions are inductively sequential.[2] We call a function *flexible* or *rigid* if all the branch nodes in its definitional tree are $flex$ or $rigid$, respectively.

Example 2. Consider the rules defining the function "$\leqslant$" in Example 1. Then

$$\begin{aligned}&branch(\mathtt{X} \leqslant \mathtt{Y}, 1, rigid, rule(\mathtt{0} \leqslant \mathtt{Y} \to \mathtt{true}),\\ &\qquad\qquad branch(\mathtt{s(M)} \leqslant \mathtt{Y}, 2, rigid, rule(\mathtt{s(M)} \leqslant \mathtt{0} \to \mathtt{false}),\\ &\qquad\qquad\qquad\qquad rule(\mathtt{s(M)} \leqslant \mathtt{s(N)} \to \mathtt{M} \leqslant \mathtt{N})))\end{aligned}$$

is a definitional tree of $\leqslant$. It is often convenient and simplifies understanding to provide a graphic representation of definitional trees. Each inner node is marked with a pattern and the $flex/rigid$ annotation, the inductive position in branches is surrounded by a box, and the leaves contain the corresponding rules. For instance, the definitional tree for the function "$\leqslant$" is illustrated in Fig. 1.

The definitional tree of a function determines the precise strategy in order to evaluate a call to this function. Informally, a rule node requires the application of this rule and a branch node requires the examination of the subterm of this function call which is specified by the position in the branch node. To provide concurrent computation threads, expressions can be combined by the *concurrent conjunction operator* &, i.e., the expression e_1 & e_2 can be reduced by reducing either e_1 or e_2. Note that we obtain the behavior of the needed narrowing strategy [10] if all functions are flexible. Moreover, functional logic languages

[2] Curry also supports rules with overlapping left-hand sides by providing *or* nodes in definitional trees, but we omit this feature here for simplicity.

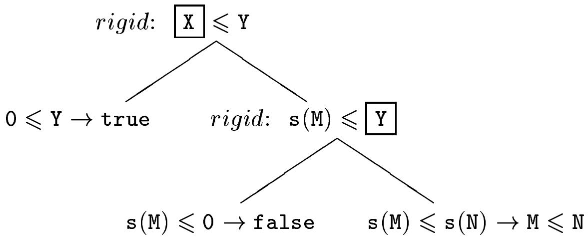

Fig. 1. Definitional tree for the function "$\leqslant$"

which are based on residuation, like Life or Escher, where functions are always deterministically evaluated or suspended and non–determinism is encoded by predicates, can be modeled with programs where all (non–Boolean) functions are rigid and all predicates (Boolean functions) are flexible.

For a precise definition of this operational semantics, it is convenient to distinguish between *complete* computation steps where one reduction has been performed and *incomplete* computation steps which are suspended due to some rigid branch.[3] Incomplete steps are called *degenerate* in [9] in the sense that some variables could have been instantiated but no subsequent reduction has been performed. We mark a substitution in an answer expression by the superscript s, i.e., $\sigma^s \circ \sigma' \,[\!]\, t$ to denote a *suspended answer expression* where the reduction part of the step has not been performed due to a suspension in a rigid branch. For convenience, we denote by σ^i a composed substitution with $\sigma = \sigma_n^s \circ \cdots \circ \sigma_1$, and by σ^c a composed substitution with $\sigma = \sigma_n \circ \cdots \circ \sigma_1$ where σ_1 does not have the form φ^s. Marks in substitutions are only a technical artifice to simplify our formulation and are simply ignored when composing and applying substitutions. $\mathcal{D}^s$ denotes the set of all disjunctive expressions where each disjunct could also be a suspended answer expression. Then the operational semantics of Curry is specified by the functions (see Fig. 2):

$$cs : \mathcal{T}(\mathcal{C} \cup \mathcal{F}, \mathcal{X}) \to \mathcal{D}^s \quad \text{and} \quad cst : \mathcal{T}(\mathcal{C} \cup \mathcal{F}, \mathcal{X}) \times DT \to \mathcal{D}^s$$

where DT stands for the set of all definitional trees. Moreover, the composition of substitutions and the replacement of subterms is extended to disjunctive expressions as follows:

$$\begin{array}{l} \{\sigma_1 \,[\!]\, t_1, \ldots, \sigma_n \,[\!]\, t_n\} \circ \sigma = \{\sigma_1 \circ \sigma \,[\!]\, t_1, \ldots, \sigma_n \circ \sigma \,[\!]\, t_n\} \\ t[\{\sigma_1 \,[\!]\, t_1, \ldots, \sigma_n \,[\!]\, t_n\}]_o = \{\sigma_1 \,[\!]\, \sigma_1(t)[t_1]_o, \ldots, \sigma_n \,[\!]\, \sigma_n(t)[t_n]_o\} \end{array}$$

As in proof procedures for logic programming, we assume that the definitional trees always contain new variables if they are used in a narrowing step. This implies that all computed substitutions are idempotent (we will implicitly assume this property in the following).

[3] In [20], this distinction is made by a special constant in the domain of disjunctive expressions while here we use a special mark at substitutions in answer expressions. We find this more convenient to formulate the PE method in residuating programs as will become apparent later.

Computation step for a single operation-rooted term t:

$$cs(f(t_1,\ldots,t_n)) = cst(f(t_1,\ldots,t_n),\mathcal{T}) \quad \text{if } \mathcal{T} \text{ is a definitional tree for } f$$

$$cs(t_1 \,\&\, t_2) = \begin{cases} \texttt{true} & \text{if } t_1 = t_2 = \texttt{true} \\ (t_1 \,\&\, t_2)[cs(t_1)]_1 & \text{if } t_1 \neq \texttt{true} \text{ and } cs(t_1) \text{ does not suspend} \\ (t_1 \,\&\, t_2)[cs(t_2)]_2 & \text{if } t_2 \neq \texttt{true},\ cs(t_2) \text{ does not suspend,} \\ & \quad \text{and } cs(t_1) \text{ suspends} \\ id^s \,\|\, t_1 \,\&\, t_2 & \text{otherwise} \end{cases}$$

$$cst(t, rule(l \to r)) = id \,\|\, \sigma(r) \quad \text{if } \sigma \text{ is a substitution with } \sigma(l) = t$$

$$cst(t, branch(\pi, o, r, \mathcal{T}_1, \ldots, \mathcal{T}_k))$$
$$= \begin{cases} cst(t,\mathcal{T}_i) \circ id & \text{if } t|_o = c(t_1,\ldots,t_n) \text{ and } pattern(\mathcal{T}_i)|_o = c(X_1,\ldots,X_n) \\ \emptyset & \text{if } t|_o = c(\ldots) \text{ and } pattern(\mathcal{T}_i)|_o \neq c(\ldots), i = 1,\ldots,k \\ id^s \,\|\, t & \text{if } t|_o = X \text{ and } r = rigid \\ \cup_{i=1}^{k} cst(\sigma_i(t),\mathcal{T}_i) \circ \sigma_i & \text{if } t|_o = X, r = flex, \text{ and } \sigma_i = \{X \mapsto pattern(\mathcal{T}_i)|_o\} \\ t[cs(t|_o)]_o \circ id & \text{if } t|_o = f(t_1,\ldots,t_n) \end{cases}$$

Derivation step for a disjunctive expression:

$$(\sigma^c \,\|\, t) \vee D \xrightarrow{\text{RN}} (\sigma_1 \circ \sigma^c \,\|\, t_1) \vee \ldots \vee (\sigma_n \circ \sigma^c \,\|\, t_n) \vee D$$
if t is operation-rooted and $cs(t) = \sigma_1 \,\|\, t_1 \vee \ldots \vee \sigma_n \,\|\, t_n$

Fig. 2. Operational semantics of concurrent functional logic programming

The overall computation strategy is a transformation $\xrightarrow{\text{RN}}$ on disjunctive expressions. It takes an operation-rooted term[4] t of a non-suspended disjunct. Then the computation step $cs(t)$ stemming from t is performed, and the selected disjunct is replaced by the computed disjunction composed with the answer computed to that point. A single computation step $cs(t)$ applies a rule, if possible (first case of *cst*), or checks the subterm corresponding to the inductive position of the branch (second case of *cst*): if it is a constructor, we proceed with the corresponding subtree (if possible); if it is a function, we evaluate it by recursively applying the strategy to this subterm; if it is a variable, we suspend (in the case of a rigid branch) or nondeterministically instantiate the variable to the constructors of all children and proceed. Hence, a concurrent conjunction of two expressions proceeds by evaluating the conjunct which does not suspend. We say that a computation $D \xrightarrow{\text{RN}}{}^{*} D'$ *flounders* if every answer expression $\sigma^i \,\|\, t \in D'$ is suspended. A goal e *flounders* iff the computation starting from e flounders.

This strategy was first introduced in [20] and differs from lazy functional languages only in the possible instantiation of free variables and from logic languages in the lazy evaluation of nested function calls. Moreover, logic programs with coroutining (i.e., delayed predicates waiting for the instantiation of some argument) can be modeled by the use of the concurrent conjunction operator &.

Note that, in each recursive step during the computation of *cst*, we compose the current substitution with the local substitution of this step (which can

[4] Here we consider only the evaluation of operation-rooted terms which is sufficient for functional logic programming where we are interested in reducing strict equalities to the constant `true`.

be the identity). Thus, each computation step can be represented as $cs(t) = \bigvee_{i=1}^{n} \sigma_{i,k_i} \circ \cdots \circ \sigma_{i,1} \,[\!]\, t_i$, where each $\sigma_{i,j}$ is either the identity or the replacement of a single variable computed in each recursive step. This is also called the *canonical representation* of a computation step. In contrast to the classical definition of narrowing (see Sect. 2), the definition of $\overset{\mathtt{RN}}{\longrightarrow}$ provides all (don't know nondeterministic) derivations at once by deriving an expression into a disjunctive expression. In order to relate $\overset{\mathtt{RN}}{\longrightarrow}$ to the classical nondeterministic narrowing relation, we also write $t \overset{\mathtt{RN}}{\leadsto}_{\sigma} t'$ if $\sigma \,[\!]\, t' \in cs(t)$.

The main difference with the needed narrowing strategy as introduced in [10] is the possibility that function calls may suspend and the special treatment of the concurrent conjunction & to deal with suspended evaluations. Therefore, we denote by $\overset{\mathtt{NN}}{\longrightarrow}$ and $\overset{\mathtt{NN}}{\leadsto}$ the relations defined similarly to $\overset{\mathtt{RN}}{\longrightarrow}$ and $\overset{\mathtt{RN}}{\leadsto}$ above but where the definition of $cs(t_1 \,\&\, t_2)$ is omitted and the case "$id^s \,[\!]\, t$ if $t|_o = X$ and $r = rigid$" is replaced by

$$cst(t, branch(\pi, o, r, \mathcal{T}_1, \ldots, \mathcal{T}_k)) \;=\; \cup_{i=1}^{k} cst(\sigma_i(t), \mathcal{T}_i) \circ \sigma_i^s$$
$$\text{if } t|_o = X, r = rigid, \text{ and } \sigma_i = \{X \mapsto pattern(\mathcal{T}_i)|_o\}.$$

The fact that $\overset{\mathtt{NN}}{\longrightarrow}$ also decorates suspended bindings with the superscript s instead of simply omitting the case "$id^s \,[\!]\, t$ if $t|_o = X$ and $r = rigid$" and the condition "$r = flex$" in the definition of cst (giving rise to the narrowing strategy of [10]) will become useful in the next section.

Note that the meaning of the concurrent conjunction & can be defined by the single rewrite rule `true & true → true` which we assume to be implicitly added to the rewrite system when we consider needed narrowing steps. This function is inductively sequential and has the two definitional trees

$$branch(\mathtt{X} \,\&\, \mathtt{Y}, 1, rigid, branch(\mathtt{true} \,\&\, \mathtt{Y}, 2, rigid, rule(\mathtt{true} \,\&\, \mathtt{true} \to \mathtt{true})))$$

and

$$branch(\mathtt{X} \,\&\, \mathtt{Y}, 2, rigid, branch(\mathtt{X} \,\&\, \mathtt{true}, 1, rigid, rule(\mathtt{true} \,\&\, \mathtt{true} \to \mathtt{true}))).$$

Now consider a term like $t_1 \,\&\, t_2$. It is obvious that a $\overset{\mathtt{RN}}{\longrightarrow}$ step where t_1 is evaluated corresponds to a needed narrowing step where the first definitional tree is taken for the root function &. Similarly, a $\overset{\mathtt{RN}}{\longrightarrow}$ step where t_2 is evaluated corresponds to a needed narrowing step with the second definitional tree for &. Thus, we obtain the following theorem which formalizes the relation between the two calculi.

Theorem 1. *Let $\mathcal{R}$ be an inductively sequential program and e a term.*

1. *If all steps in the derivation $e \overset{\mathtt{RN}^*}{\leadsto}_{\sigma} e'$ are complete, then there exists a needed narrowing derivation $e \overset{\mathtt{NN}^*}{\leadsto}_{\sigma} e'$ in $\mathcal{R}$.*
2. *If $e \overset{\mathtt{NN}^*}{\leadsto}_{\sigma} e'$ is a needed narrowing derivation for e in $\mathcal{R}$, then there exists a derivation $e \overset{\mathtt{RN}^*}{\leadsto}_{\theta} e''$ such that $\exists \varphi.\ \varphi(e'') \to^* e'$ and $\sigma = \varphi \circ \theta$.*

4 Partial Evaluation of Residuating Functional Logic Programs

In this section, we extend the framework of [6] (and, particularly, the instance introduced in [7]) in order to take into account delayed function calls during PE. Specialized definitions are basically produced by constructing a set of rules (called *resultants*) of the form

$$\begin{array}{l} \sigma_1(s) \to t_1 \\ \quad\ldots \\ \sigma_n(s) \to t_n \end{array}$$

associated to a given (partial) computation

$$id \,[\!]\, s \stackrel{\mathtt{RN}}{\longrightarrow}{}^{+} \{\sigma_1 \,[\!]\, t_1 \;\vee\; \ldots \;\vee\; \sigma_n \,[\!]\, t_n\} \;.$$

After that, a renaming transformation is performed in order to ensure that the specialized definition is inductively sequential and also to guarantee its *independence* (in the sense of [27]).

Informally, the renaming transformation proceeds as follows. First, an *independent renaming* ρ for a set of terms S is constructed, which consists of a mapping from terms to terms such that for all $s \in S$, we have $\rho(s) = f(\overline{x_n})$, where $\overline{x_n}$ are the distinct variables in s in the order of their first occurrence and f is a *fresh* function symbol. We also let $\rho(S)$ denote the set $S' = \{\rho(s) \mid s \in S\}$. While the independent renaming suffices to rename the left-hand sides of resultants (since they are constructor instances of the specialized calls), the right-hand sides are renamed by means of the auxiliary function ren_ρ, which *recursively* replaces each call in the given expression by a call to the corresponding renamed function (according to ρ).

Unfortunately, the framework of PE above cannot simply be transferred to residuating programs, since a naïve treatment of suspended calls can give rise to resultants which do not preserve the program's behavior, as illustrated in the following examples.

Example 3. Consider again the rules defining the functions "$\leqslant$" and "$+$" of Example 1, and assume now that "$\leqslant$" is flexible and "$+$" is rigid. Given the expression $\mathtt{X} \leqslant \mathtt{Y} + \mathtt{0}$, we have the partial computation

$$\mathtt{id} \,[\!]\, \mathtt{X} \leqslant \mathtt{Y} + \mathtt{0} \;\stackrel{\mathtt{RN}}{\longrightarrow}\; \{\mathtt{X} = \mathtt{0}\} \,[\!]\, \mathtt{true} \;\vee\; \{\mathtt{X} = \mathtt{s(M)}\}^{\mathtt{s}} \,[\!]\, \mathtt{s(M)} \leqslant \mathtt{Y} + \mathtt{0}$$

in which the second disjunct corresponds to an incomplete step. The associated resultants are the following:[5]

$$\begin{array}{r} \mathtt{0} \leqslant \mathtt{Y} + \mathtt{0} \to \mathtt{true} \\ \mathtt{s(M)} \leqslant \mathtt{Y} + \mathtt{0} \to \mathtt{s(M)} \leqslant \mathtt{Y} + \mathtt{0} \end{array}$$

Obviously, any specialization containing the second rule does not preserve the semantics of the original program (for the intended goals). Unfortunately, getting rid of this trivial resultant does not preserve the semantics either.

[5] We do not consider the renaming of resultants since it is not relevant here.

The above example reveals the need to relax the standard computation model during partial evaluation in order to "complete" the suspended steps in some suitable way. For instance, we could avoid suspensions by simply replacing $\stackrel{\mathtt{RN}}{\longrightarrow}$ with $\stackrel{\mathtt{NN}}{\longrightarrow}$ during PE. This raises the question of whether it is possible to infer safe evaluation annotations for the specialized definitions, i.e., annotations such that total correctness is entailed. The following example answers this question negatively.

Example 4. Reconsider the program and goal of Example 3, but use $\stackrel{\mathtt{NN}}{\longrightarrow}$ to construct the partial computation

$$\begin{aligned}\mathtt{id} \,\|\, \mathtt{X} \leqslant \mathtt{Y}+\mathtt{0} \stackrel{\mathtt{NN}}{\longrightarrow} \{\mathtt{X}=\mathtt{0}\} \,\|\, \mathtt{true} \vee \{\mathtt{X}=\mathtt{s(M)}, \mathtt{Y}=\mathtt{0}\} \,\|\, \mathtt{s(M)} \leqslant \mathtt{0} \\ \vee \{\mathtt{X}=\mathtt{s(M)}, \mathtt{Y}=\mathtt{s(Z)}\} \,\|\, \mathtt{s(M)} \leqslant \mathtt{s(Z+0)}\end{aligned}$$

whose associated resultants are

$$\begin{aligned}\mathtt{0} \leqslant \mathtt{Y}+\mathtt{0} &\rightarrow \mathtt{true} \\ \mathtt{s(M)} \leqslant \mathtt{0}+\mathtt{0} &\rightarrow \mathtt{s(M)} \leqslant \mathtt{0} \\ \mathtt{s(M)} \leqslant \mathtt{s(Z)}+\mathtt{0} &\rightarrow \mathtt{s(M)} \leqslant \mathtt{s(Z+0)}\end{aligned}$$

Then, neither *flex* nor *rigid* is a correct annotation for the specialized rules. If we assume that they are flexible, then a goal of the form $\mathtt{s(X)} \leqslant \mathtt{Y}+\mathtt{0}$ would succeed (with answer substitution $\{\mathtt{Y}=\mathtt{0}\}$) using the specialized rules whereas it suspends in the original program. On the other hand, declaring the new definition as *rigid* does not work either, since a goal $\mathtt{X} \leqslant \mathtt{Y}+\mathtt{0}$ succeeds in the original program (with answer $\{\mathtt{X}=\mathtt{0}\}$), whereas it suspends using the specialized rules.

Informally, the annotation *flex* for the specialized function is not safe since the bindings for the variable Y in the lhs of the second and third resultants have been brought by the evaluation of the rigid function "+". Similarly, the annotation *rigid* does not work since (at runtime) it prevents the considered call from matching the lhs of the first resultant because the variable X was instantiated by evaluating (at PE time) the flexible function "$\leqslant$".

Our proposed solution is essentially as follows. We distinguish between two kinds of computations: those in which the initial step for the considered expression is incomplete and those which involve no kind of suspension (because they are eventually stopped before). In the latter case, we simply use the $\stackrel{\mathtt{RN}}{\longrightarrow}$ computation model whereas, in the former case, we proceed to complete the degenerate step by using the relaxed relation $\stackrel{\mathtt{NN}}{\longrightarrow}$. This allows us to infer safe evaluation annotations for specialized definitions as follows:

- We annotate as *flex* the specialized definitions which result from computations with no suspension. This is justified by the fact that all variable bindings propagated to the left-hand sides of specialized rules come from the evaluation of flexible functions (since evaluation of rigid functions causes no binding for goal variables). Thus, the handling of these specialized functions as flexible (at runtime) cannot introduce undesired bindings.

$$
\begin{array}{ll}
slist(id) & = [\,] \\
slist(\varphi_k \circ \cdots \circ \varphi_1) & = [\theta^m | slist(\varphi_k \circ \cdots \circ \varphi_{j+1})] \\
& \quad \textbf{where } \theta = \varphi_j \circ \cdots \circ \varphi_1, m = eval(\varphi_1), \text{ and } j \text{ is the} \\
& \quad \text{maximum } i \in \{1, \ldots, k\} \text{ such that } \forall p \in \{1, \ldots, i\} \\
& \quad eval(\varphi_p) = eval(\varphi_1) \text{ and } eval(\varphi_{j+1}) \neq eval(\varphi_1)
\end{array}
$$

$$
eval(\varphi) = \begin{cases} rigid & \text{if } \varphi \text{ is marked with the superscript } s \\ flex & \text{otherwise} \end{cases}
$$

$$
\begin{array}{l}
split(l, r, [\varphi^a]) = \{\varphi^a(l) \to ren_\rho(r), \text{ with evaluation annotation } a\} \\
split(l, r, [\varphi^a, \theta^b | tail]) = \{\varphi^a(l) \to l', \text{ with eval. annotation } a\} \cup split(l', r, [\theta^b | tail]) \\
\qquad \textbf{where } l' = f(x_1, \ldots, x_n),\ f \in \Sigma_{\mathtt{inter}} \text{ is a fresh function} \\
\qquad \text{symbol, and } \mathcal{V}ar(\varphi^a(l)) = \{x_1, \ldots, x_n\}.
\end{array}
$$

Fig. 3. Auxiliary functions for partial evaluation

– In case of a suspension, we are constrained to split resultants by introducing several intermediate functions with befitting evaluation annotations. This is necessary because the $\xrightarrow{\mathtt{NN}}$ step can introduce bindings which come both from flexible and rigid functions (as shown in Example 4) and the splitting avoids the mixing of bindings of different nature (*flex* and *rigid*).

Formally, a partial evaluation based on the $\xrightarrow{\mathtt{RN}}$ calculus (RNPE for short) is constructed from a set of terms S together with a set of (partial) computations for the terms in S. In the following, we denote by $\Sigma_{\mathtt{inter}}$ a set of fresh function symbols. These are the symbols which are used to construct the intermediate functions associated to the partial evaluation of suspended expressions.

Definition 1 (partial evaluation). *Let $\mathcal{R}$ be a TRS, $S = \{s_1, \ldots, s_n\}$ a finite set of terms, and $\mathcal{A}_1, \ldots, \mathcal{A}_n$ finite (partial) $\xrightarrow{\mathtt{RN}}$ computations for $s_1, \ldots, s_n$ in $\mathcal{R}$ of the form:*

$$
\mathcal{A}_k = id \,[\![\, s_k \xrightarrow{\mathtt{RN}}{}^{+} D_k, \quad k = 1, \ldots, n
$$

where all steps are complete, except (possibly) for the initial one. Let ρ be an independent renaming of S. Then, the set of rewrite rules $\mathcal{R}' =$

$$
\begin{array}{ll}
\{\sigma^c(\rho(s_k)) \to ren_\rho(r) \mid \sigma^c \,[\![\, r \in D_k\}_{k=1}^n & \textit{(non-suspension)} \\
\bigcup & \\
\{split(\rho(s_k), r, slist(\theta \circ \sigma)) \mid \theta^i \,[\![\, \theta^i(s_k) \in D_k,\ \theta^i(s_k) \overset{\mathtt{NN}}{\rightsquigarrow}_\sigma r\}_{k=1}^n & \textit{(suspension)}
\end{array}
$$

is a partial evaluation of S in $\mathcal{R}$ (under ρ)*. The evaluation annotation for the derivations involving no suspension is flex, whereas the resultants (and their evaluation annotations) for the suspended derivations are computed by means of the auxiliary functions shown in Fig. 3.* [6]

[6] In the definition of $slist(\sigma)$ we consider that σ is expressed in its canonical representation.

Roughly speaking, the resultants associated to (one-step) $\xrightarrow{\mathtt{NN}}$ computations are split into a set of "intermediate" rules, one rule associated to each sequence of consecutive bindings with the same superscript mark (suspended or non-suspended). This way, the specialized rules mimic the behaviour of the original functions perfectly. Note that the intermediate rules play no particular role in the evaluation of expressions, but are only necessary to preserve the flex or rigid nature of the functions in the initial program. The following example shows the construction of a RNPE for a suspended expression.

Example 5. Let us consider the following rules:

```
f(a,b)  →  c   % flex
g(b,c)  →  b   % rigid
  h(c)  →  c   % flex
```

A PE for $\mathtt{f(X,g(Y,h(Z)))}$ constructed from the (suspended) derivation

$$\mathtt{f(X,g(Y,h(Z)))} \;\overset{\mathtt{RN}}{\rightsquigarrow}_{\{\mathtt{X}\mapsto\mathtt{a}\}^s}\; \mathtt{f(a,g(Y,h(Z)))}$$

proceeds as follows (here we assume that $\rho(\mathtt{f(X,g(Y,h(Z)))}) = \mathtt{f'(X,Y,Z)}$):

1. First, the $\overset{\mathtt{NN}}{\rightsquigarrow}$ step $\mathtt{f(a,g(Y,h(Z)))} \overset{\mathtt{NN}}{\rightsquigarrow}_{\{\mathtt{Y}\mapsto\mathtt{b},\mathtt{Z}\mapsto\mathtt{c}\}} \mathtt{f(a,g(b,c))}$ is computed.
2. Then, the call $\mathit{slist}(\{\mathtt{X}\mapsto\mathtt{a},\mathtt{Y}\mapsto\mathtt{b},\mathtt{Z}\mapsto\mathtt{c}\})$ is undertaken, which returns the set of substitutions $[\{\mathtt{X}\mapsto\mathtt{a}\}^{\mathtt{flex}}, \{\mathtt{Y}\mapsto\mathtt{b}\}^{\mathtt{rigid}}, \{\mathtt{Z}\mapsto\mathtt{c}\}^{\mathtt{flex}}]$.
3. Finally, the computation of *split* proceeds as follows:

$$\begin{array}{l}
\mathit{split}(\mathtt{f'(X,Y,Z)}, \mathtt{f(a,g(b,c))}, [\{\mathtt{X}\mapsto\mathtt{a}\}^{\mathtt{flex}}, \{\mathtt{Y}\mapsto\mathtt{b}\}^{\mathtt{rigid}}, \{\mathtt{Z}\mapsto\mathtt{c}\}^{\mathtt{flex}}]) \\
\quad = \{\mathtt{f'(a,Y,Z)} \;\to\; \mathtt{f'_1(Y,Z)}\} \\
\qquad \cup\, \mathit{split}(\mathtt{f'_1(Y,Z)}, \mathtt{f(a,g(b,c))}, [\{\mathtt{Y}\mapsto\mathtt{b}\}^{\mathtt{rigid}}, \{\mathtt{Z}\mapsto\mathtt{c}\}^{\mathtt{flex}}]) \\
\quad = \{\mathtt{f'(a,Y,Z)} \;\to\; \mathtt{f'_1(Y,Z)}, \\
\qquad\; \mathtt{f'_1(b,Z)} \;\to\; \mathtt{f'_2(Z)}\} \\
\qquad \cup\, \mathit{split}(\mathtt{f'_2(Z)}, \mathtt{f(a,g(b,c))}, [\{\mathtt{Z}\mapsto\mathtt{c}\}^{\mathtt{flex}}]) \\
\quad = \{\mathtt{f'(a,Y,Z)} \;\to\; \mathtt{f'_1(Y,Z)}, \\
\qquad\; \mathtt{f'_1(b,Z)} \;\to\; \mathtt{f'_2(Z)}, \\
\qquad\; \mathtt{f'_2(c)} \;\to\; \mathtt{ren}_\rho(\mathtt{f(a,g(b,c))})\}
\end{array}$$

   where "$\mathtt{f'}$" and "$\mathtt{f'_2}$" are flexible, and "$\mathtt{f'_1}$" is rigid.

A general requirement in the partial evaluation of lazy functional logic programs is that no constructor-rooted expression can be evaluated during PE [5, 7]. This is also true in our context, although we did not make this condition explicit in Def. 1 since the computation model is only defined for operation-rooted terms. If we consider the more general setting in which the operational semantics is also defined for constructor-rooted terms, then this condition must appear explicitly.

For the correctness of partial evaluation, a *closedness* condition is commonly required which ensures that all calls which might occur during the execution of the specialized program are covered by some program rule. The following is an easy extension of the closedness condition of [6] to the case of residuating programs. Informally, an operation-rooted term t is closed w.r.t. a set of calls S if it is an instance of a term in S and the terms in the matching substitution are recursively closed by S.

Definition 2 (closedness). *Let S be a finite set of terms. We say that a term t is S-closed if $closed(S,t)$ holds, where the predicate closed is defined inductively as follows:*

$$closed(S,t) \;\Leftrightarrow\; \begin{cases} true & \textit{if } t \in \mathcal{X} \\ \bigwedge_{i=1,\ldots,n} closed(S,t_i) & \textit{if } t = c(\overline{t_n}),\ c \in (\mathcal{C} \cup \{\doteq, \&\} \cup \Sigma_{\mathtt{inter}}) \\ \bigwedge_{x \mapsto t' \in \theta} closed(S,t') & \textit{if } \exists\theta, \exists s \in S \textit{ such that } \theta(s) = t \end{cases}$$

We say that a set of terms T is S-closed, written $closed(S,T)$, if $closed(S,t)$ holds for all $t \in T$, and we say that a TRS $\mathcal{R}$ is S-closed if $closed(S,\mathcal{R}_{calls})$ holds. Here we denote the set of the rhs's of the rules in $\mathcal{R}$ by $\mathcal{R}_{calls}$.

Note that expressions rooted by an "intermediate" function symbol in $\Sigma_{\mathtt{inter}}$ are S-closed by definition, independently of the considered set S. This is motivated by the fact that intermediate functions are not "visible" in the specialized program (i.e., they do not belong to the set of specialized calls), but are only intended as a mechanism to preserve the floundering behaviour.

The following theorem states an important property of RNPE: if the input program is inductively sequential, then the specialized program is also inductively sequential.

Theorem 2. *Let $\mathcal{R}$ be an inductively sequential program and S a finite set of operation-rooted terms. Then each RNPE of $\mathcal{R}$ w.r.t. S is inductively sequential.*

The following result establishes the precise relation between partial evaluations based on needed narrowing (without residuation, as defined in [7]), which we call NNPE for short, and partial evaluations as defined here. Intuitively, any RNPE $\mathcal{R}'$ can be transformed into an equivalent program $\mathcal{R}''$ (w.r.t. needed narrowing) by replacing each set of rules

$$\begin{aligned} \sigma(\rho(s)) &\to f_1(\overline{x_{m_1}}) \\ \varphi_1(f_1(\overline{x_{m_1}})) &\to f_2(\overline{x_{m_2}}) \\ &\cdots \\ \varphi_k(f_k(\overline{x_{m_k}})) &\to ren_\rho(r) \end{aligned}$$

associated to a suspended expression, by the new rule

$$\theta(\rho(s)) \to ren_\rho(r), \quad \text{with } \theta = \varphi_k \circ \cdots \circ \varphi_1 \circ \sigma$$

and ignoring all the evaluation annotations. The program constructed in this way is a correct NNPE of $\mathcal{R}$ w.r.t. S (under ρ), as formalized in the following.

Theorem 3. *Let $\mathcal{R}$ be an inductively sequential program. Let S be a finite set of operation-rooted terms and ρ an independent renaming of S. If $\mathcal{R}'$ is a RNPE of $\mathcal{R}$ w.r.t. S (under ρ), then there exists a NNPE $\mathcal{R}''$ of $\mathcal{R}$ w.r.t. S (under ρ) such that, for all goals e, we have $e \overset{\mathtt{NN}^*}{\leadsto}_\sigma true$ in $\mathcal{R}'$ iff $e \overset{\mathtt{NN}^*}{\leadsto}_\sigma true$ in $\mathcal{R}''$.*

Now, we state the partial correctness of RNPE, which amounts to the full computational equivalence between the original and specialized programs when the considered goal does not flounder.

Theorem 4 (partial correctness). *Let $\mathcal{R}$ be an inductively sequential program. Let e be an equation, $V \supseteq \mathcal{V}ar(e)$ a finite set of variables, S a finite set of operation-rooted terms, and ρ an independent renaming of S. Let $\mathcal{R}'$ be a RNPE of $\mathcal{R}$ w.r.t. S (under ρ) such that $\mathcal{R}' \cup \{e'\}$ is S'-closed, where $e' = ren_\rho(e)$ and $S' = \rho(S)$.*

1. *If $e \overset{\mathtt{RN}^*}{\leadsto}_{\sigma'}$ true in $\mathcal{R}'$, then $e \overset{\mathtt{RN}^*}{\leadsto}_{\sigma}$ t and $\varphi(t) \to^*$ true in $\mathcal{R}$ with $\sigma' = \varphi \circ \sigma$ $[V]$.*
2. *If $e \overset{\mathtt{RN}^*}{\leadsto}_{\sigma}$ true in $\mathcal{R}$, then $e \overset{\mathtt{RN}^*}{\leadsto}_{\sigma'}$ t and $\varphi(t') \to^*$ true in $\mathcal{R}'$ with $\sigma = \varphi \circ \sigma'$ $[V]$.*

Loosely speaking, the previous result establishes that, if evaluation annotations are not considered (that is, no function calls are delayed), then the specialized program $\mathcal{R}'$ *is able* to produce the same answers (computed by needed narrowing) as the original one $\mathcal{R}$ (and vice versa). The preservation of floundering-freeness (i.e., absence of floundering) for the intended goals is needed to establish the total correctness of the transformation. On the other hand, it ensures that the transformation does not introduce additional floundering points, which is of crucial importance when we are using the transformation for optimizing a program. Moreover, this feature may allow us to use the transformation as a tool for proving floundering-freeness of the original program (see Example 7). In fact, if after the transformation we can state that $\mathcal{R}' \cup \{e'\}$ does not flounder, then we are also sure that $\mathcal{R} \cup \{e\}$ does not flounder either, where $e' = ren_\rho(e)$.

Unfortunately, the recursive notion of closedness introduced in Def. 2 is too weak (generous) to preserve the floundering behaviour, as illustrated by the following example.

Example 6. Let us consider the following set of rules:

```
f(X,a) → g(X)   % flex        h(a) → b   % rigid
  g(b) → c      % flex
```

A RNPE of $\{\mathtt{f(X,Y)}, \mathtt{h(X)}\}$ under $\rho = \{\mathtt{f(X,Y)} \mapsto \mathtt{f'(X,Y)},\ \mathtt{h(X)} \mapsto \mathtt{h'(X)}\}$ is

```
f'(b,a) → c   % flex
  h'(a) → b   % rigid
```

Now, the S-closed expression $\mathtt{f(h(X),X)}$ has the following successful computation in the original program

$$\mathtt{id} \parallel \mathtt{f(h(X),X)} \xrightarrow{\mathtt{RN}} \{\mathtt{X} \mapsto \mathtt{a}\} \parallel \mathtt{g(h(a))} \xrightarrow{\mathtt{RN}} \{\mathtt{X} \mapsto \mathtt{a}\} \parallel \mathtt{g(b)} \xrightarrow{\mathtt{RN}} \{\mathtt{X} \mapsto \mathtt{a}\} \parallel \mathtt{c}$$

whereas $\rho(\mathtt{f(h(X),X)}) = \mathtt{f'(h'(X),X)}$ may suspend in the specialized program, e.g., by considering the following definitional tree

$$\begin{array}{l} branch(\mathtt{f'(X,Y)}, 1, flex, \\ \qquad\qquad branch(\mathtt{f'(b,Y)}, 2, flex, \\ \qquad\qquad\qquad\qquad rule(\mathtt{f'(b,a)} \to \mathtt{c}))) \end{array}$$

for the specialized function $\mathtt{f'}$.

Informally, the problem is that the recursive notion of closedness only works when the considered operational model is *compositional*, as it essentially exploits the fact that the meaning of a complex expression $\mathtt{f(h(X),X)}$ can be retrieved from the semantics of its "unnested" constituents $\mathtt{f(Y,X)}$ and $\mathtt{h(X)}$ [6]. However, the $\overset{\mathtt{RN}}{\longrightarrow}$ calculus is not compositional due to the presence of delayed function calls, and hence the meaning of the call $\mathtt{f(h(X),X)}$ (which does not flounder) cannot be obtained from the meaning of the calls $\mathtt{f(Y,X)}$ and $\mathtt{h(X)}$ since the second one flounders. Thus, we consider in the following a restricted notion of closedness (called *basic* closedness in [6], in symbols $closed^-$) which is defined as the recursive closedness of Def. 2 except for the case

$$closed(S,t) = \bigwedge_{x \mapsto t' \in \theta} closed(S,t') \quad \text{if } \exists\theta, \exists s \in S \text{ such that } \theta(s) = t$$

which is replaced by the more simple condition

$$closed^-(S,t) = true \quad \text{if } \exists\theta, \exists s \in S \text{ such that } \theta(s) = t \text{ and } \theta \text{ is constructor.}$$

The following result states the equivalence between the original and specialized programs w.r.t. floundering-freeness.

Theorem 5 (floundering-freeness). *Let $\mathcal{R}$ be an inductively sequential program, e an equation, S a finite set of operation-rooted terms, and ρ an independent renaming of S. Let $\mathcal{R}'$ be a RNPE of $\mathcal{R}$ w.r.t. S (under ρ) such that $\mathcal{R}' \cup \{e'\}$ is S'-closed$^-$, where $e' = ren_\rho(e)$ and $S' = \rho(S)$. Then, e flounders in $\mathcal{R}$ iff e' flounders in $\mathcal{R}'$.*

As a corollary of Theorems 4 and 5, we can establish the total correctness of the transformation.

Theorem 6 (total correctness). *Let $\mathcal{R}$ be an inductively sequential program. Let e be an equation, $V \supseteq \mathcal{V}ar(e)$ a finite set of variables, S a finite set of operation-rooted terms, and ρ an independent renaming of S. Let $\mathcal{R}'$ be a RNPE of $\mathcal{R}$ w.r.t. S (under ρ) such that $\mathcal{R}' \cup \{e'\}$ is S'-closed$^-$, where $e' = ren_\rho(e)$ and $S' = \rho(S)$.*

1. *If $e' \overset{\mathtt{RN}^*}{\rightsquigarrow}_{\sigma'} true$ in $\mathcal{R}'$, then $e \overset{\mathtt{RN}^*}{\rightsquigarrow}_{\sigma} true$ in $\mathcal{R}$ where $\sigma' = \sigma\ [V]$ (soundness)*
2. *If $e \overset{\mathtt{RN}^*}{\rightsquigarrow}_{\sigma} true$ in $\mathcal{R}$, then $e' \overset{\mathtt{RN}^*}{\rightsquigarrow}_{\sigma'} true$ in $\mathcal{R}'$ where $\sigma' = \sigma\ [V]$ (completeness)*

5 Some Experiments

The INDY system v1.8 is a rather concise implementation of a partial evaluator for functional logic programs (a detailed description of the system can be found in [3]). The partial evaluator described in Sect. 4 has been implemented in the INDY system and used to conduct some experiments (extracted from the Curry library[7]) which illustrate the advantages of the RNPE method in the context of residuating functional logic programs as well as the practicality of our approach.

[7] Available from URL: `http://www-i2.informatik.rwth-aachen.de/~hanus/curry`.

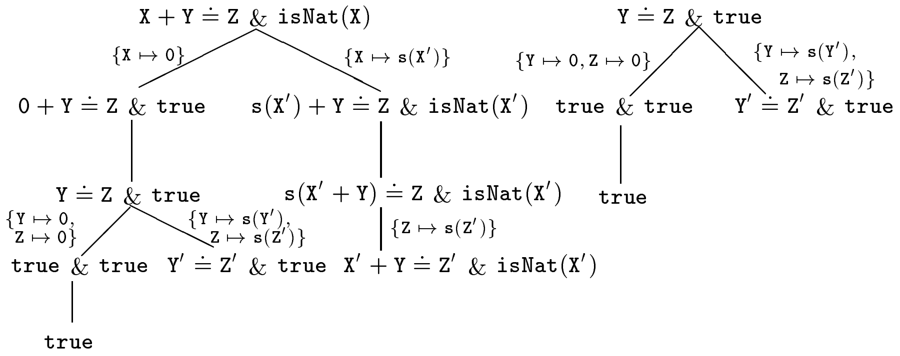

Fig. 4. Partial computations for $\mathtt{X} + \mathtt{Y} \doteq \mathtt{Z}\ \&\ \mathtt{isNat(X)}$ and $\mathtt{Y} \doteq \mathtt{Z}\ \&\ \mathtt{true}$

Let us introduce an example which shows that RNPE can be used for proving floundering-freeness of a class of goals in a given program.

Example 7. Consider the following program which defines the arithmetic addition and the predicate `isNat`, which returns `true` when the argument is a natural number:

$$\begin{array}{rclcrcl} \mathtt{0} + \mathtt{Y} & \to & \mathtt{Y} & \qquad & \mathtt{isNat(0)} & \to & \mathtt{true} \\ \mathtt{s(X)} + \mathtt{Y} & \to & \mathtt{s(X+Y)} & & \mathtt{isNat(s(X))} & \to & \mathtt{isNat(X)} \end{array}$$

where "+" is rigid and "`isNat`" is flexible. Let $S = \{\mathtt{X} + \mathtt{Y} \doteq \mathtt{Z} \,\&\, \mathtt{isNat(X)},\ \mathtt{Y} \doteq \mathtt{Z} \,\&\, \mathtt{true}\}$ and consider the independent renaming $\rho = \{\mathtt{X} + \mathtt{Y} \doteq \mathtt{Z} \,\&\, \mathtt{isNat(X)} \mapsto \mathtt{and3(X,Y,Z)},\ \mathtt{Y} \doteq \mathtt{Z} \,\&\, \mathtt{true} \mapsto \mathtt{and2(Y,Z)}\}$. Now, by considering the partial computations depicted in Fig. 4,[8] the following RNPE of the program w.r.t. S (under ρ) is constructed:

$$\begin{array}{rclcrcl} \mathtt{and3(0,0,0)} & \to & \mathtt{true} & \qquad & \mathtt{and2(0,0)} & \to & \mathtt{true} \\ \mathtt{and3(0,s(Y),s(Z))} & \to & \mathtt{and2(Y,Z)} & & \mathtt{and2(s(X),s(Y))} & \to & \mathtt{and2(X,Y)} \\ \mathtt{and3(s(X),Y,s(Z))} & \to & \mathtt{and3(X,Y,Z)} & & & & \end{array}$$

where both `and3` and `and2` are flexible functions. Then, for proving floundering-freeness it is sufficient to check that no operation symbol of the resulting partially evaluated program has a *rigid* annotation. For instance, one can easily see that the goal $\mathtt{X} + \mathtt{Y} = \mathtt{Z} \,\&\, \mathtt{isNat(X)}$ is floundering-free in the residual program (hence in the original), since the program has no rigid functions, while in the original program this is not immediate.

In the next example, we intend to show that RNPE can be also used to simplify the dynamic behavior of a program, thus allowing us to achieve a significant optimization.

[8] Here we assume that the strict equality $\doteq$ is flexible.

Example 8. Consider the classical map coloring program which assigns a color to each of four countries such that countries with a common border have different colors:

$$
\begin{array}{ll}
\texttt{isColor(red)} & \rightarrow \texttt{true} \\
\texttt{isColor(yellow)} & \rightarrow \texttt{true} \\
\texttt{isColor(green)} & \rightarrow \texttt{true} \\
\texttt{coloring(l1,l2,l3,l4)} & \rightarrow \texttt{isColor(l1)} \,\&\, \texttt{isColor(l2)} \\
 & \qquad \,\&\, \texttt{isColor(l3)} \,\&\, \texttt{isColor(l4)} \\
\texttt{correct(l1,l2,l3,l4)} & \rightarrow \texttt{diff(l1,l2)} \,\&\, \texttt{diff(l1,l3)} \\
 & \qquad \,\&\, \texttt{diff(l2,l4)} \,\&\, \texttt{diff(l3,l4)}
\end{array}
$$

where the predefined function `diff` is the only rigid function (it makes use of the strict equality predicate in order to check whether its arguments are different). Now, we consider the specialization of the expression $\texttt{correct(l1,l2,l3,l4)}$ & $\texttt{coloring(l1,l2,l3,l4)}$, which gives the following specialized program:

$$
\begin{array}{ll}
\texttt{and4(red,yellow,green,red)} & \rightarrow \texttt{true} \\
\texttt{and4(red,green,yellow,red)} & \rightarrow \texttt{true} \\
\texttt{and4(yellow,red,green,yellow)} & \rightarrow \texttt{true} \\
\texttt{and4(yellow,green,red,yellow)} & \rightarrow \texttt{true} \\
\texttt{and4(green,red,yellow,green)} & \rightarrow \texttt{true} \\
\texttt{and4(green,yellow,red,green)} & \rightarrow \texttt{true}
\end{array}
$$

where some potential colorings have been discarded, thus simplifying the dynamic behavior of the program and achieving a significant speedup (actually it runs 23 times faster).

Our preliminary experiments show that RNPE is able to produce significant speed-up's on several typical concurrent Curry programs. Moreover, it is a conservative extension of the previous INDY system based on needed narrowing, since RNPE boils down to NNPE when all program functions are flexible.

6 Conclusions

We have presented a general partial evaluation framework for Curry, a truly lazy functional logic language whose development is an international initiative intended to provide a standard for the area. The framework derives from that of [7] and extends it to the combination of needed narrowing and residuation. The extended framework allows us to safely deal with the evaluation annotations, which is crucial for controlling unfolding during PE as well as for correctly synthesizing evaluation annotations for the specialized functions.

Despite the practical importance of logic programs with dynamic scheduling, there has been surprisingly little work devoted to their specialization. The only transformation framework that we are aware of for logic languages with delays is that of Etalle and Gabbrielli [14], which is based on the fold/unfold approach to program transformation. It differs from our methodology, since our framework is based on the (automatic) PE approach and applies to logic languages with lazy functions. Moreover, we allow unfolding of suspended expressions at PE time, which is not the case of [14].

An interesting prospect for future work is to extend the framework to encompass the PE of non-deterministic (i.e., non-confluent) functions, which is ahead of the state of the art as we know it even for pure functional programming languages [23]. We are also considering how to discover slices of code in the residual program which are "semantically dead", according to the considered operational principle of functional logic programs with delays, since they can be safely removed without influencing the intended result.

References

1. H. Aït-Kaci. An Overview of LIFE. In J.W. Schmidt and A.A. Stogny, editors, *Proc. Workshop on Next Generation Information Systems Technology*, pages 42–58. Springer LNCS 504, 1990.
2. H. Aït-Kaci, P. Lincoln, and R. Nasr. Le Fun: Logic, equations, and Functions. In *Proc. of Fourth IEEE Int'l Symp. on Logic Programming*, pages 17–23. IEEE, New York, 1987.
3. E. Albert, M. Alpuente, M. Falaschi, and G. Vidal. INDY User's Manual. Technical Report DSIC-II/12/98, UPV, 1998. Available from URL: `http://www.dsic.upv.es/users/elp/papers.html`.
4. E. Albert, M. Alpuente, M. Hanus, and G. Vidal. Partial Evaluation of Residuating Functional Logic Programs. Technical report, DSIC, UPV, 1999. Available from URL: `http://www.dsic.upv.es/users/elp/papers.html`.
5. M. Alpuente, M. Falaschi, P. Julián, and G. Vidal. Specialization of Lazy Functional Logic Programs. In *Proc. of the ACM SIGPLAN Conf. on Partial Evaluation and Semantics-Based Program Manipulation, PEPM'97*, volume 32, 12 of *Sigplan Notices*, pages 151–162, New York, 1997. ACM Press.
6. M. Alpuente, M. Falaschi, and G. Vidal. Partial Evaluation of Functional Logic Programs. *ACM Transactions on Programming Languages and Systems*, 20(4):768–844, 1998.
7. M. Alpuente, M. Hanus, S. Lucas, and G. Vidal. Specialization of Functional Logic Programs Based on Needed Narrowing. In P. Lee, editor, *Proc. of the Int'l Conference on Functional Programming, ICFP'99*, Paris (France). ACM, New York, 1999.
8. S. Antoy. Definitional trees. In *Proc. of the 3rd Int'l Conference on Algebraic and Logic Programming, ALP'92*, pages 143–157. Springer LNCS 632, 1992.
9. S. Antoy. Optimal non-deterministic functional logic computations. In *Proc. of the Int'l Conference on Algebraic and Logic Programming, ALP'97*, pages 16–30. Springer LNCS 1298, 1997.
10. S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. In *Proc. 21st ACM Symp. on Principles of Programming Languages, Portland*, pages 268–279, 1994.
11. J. Boye. Avoiding Dynamic Delays in Functional Logic Languages. In J. Penjam and M. Bruynooghe, editors, *Proc. of PLILP'93*, pages 12–27. Springer LNCS 714, 1993.
12. C. Consel and O. Danvy. Tutorial notes on Partial Evaluation. In *Proc. of 20th Annual ACM Symp. on Principles of Programming Languages*, pages 493–501. ACM, New York, 1993.

13. N. Dershowitz and J.-P. Jouannaud. Rewrite Systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, pages 243–320. Elsevier, Amsterdam, 1990.
14. S. Etalle, M. Gabbrielli, and E. Marchiori. A Transformation System for CLP with Dynamic Scheduling and CCP. In *Proc. of the ACM Sigplan PEPM'97*, pages 137–150. ACM Press, New York, 1997.
15. J. Gallagher. Tutorial on Specialisation of Logic Programs. In *Proc. of Partial Evaluation and Semantics-Based Program Manipulation, Copenhagen, Denmark, June 1993*, pages 88–98. ACM, New York, 1993.
16. E. Giovannetti, G. Levi, C. Moiso, and C. Palamidessi. Kernel Leaf: A Logic plus Functional Language. *Journal of Computer and System Sciences*, 42:363–377, 1991.
17. R. Glück and M.H. Sørensen. Partial Deduction and Driving are Equivalent. In *Proc. of PLILP'94*, pages 165–181. Springer LNCS 844, 1994.
18. M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, 19&20:583–628, 1994.
19. M. Hanus. Analysis of Residuating Logic Programs. *Journal of Logic Programming*, 24(3):161–199, 1995.
20. M. Hanus. A unified computation model for functional and logic programming. In *Proc. of the 24th ACM Symposium on Principles of Programming Languages (Paris)*, pages 80–93. ACM, New York, 1997.
21. M. Hanus (ed.). Curry: An Integrated Functional Logic Language. Available at `http://www-i2.informatik.rwth-aachen.de/~hanus/curry`, 1999.
22. N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Englewood Cliffs, NJ, 1993.
23. Neil D. Jones. An Introduction to Partial Evaluation. *ACM Computing Surveys*, 28(3):480–503, Sept. 1996.
24. M. Leuschel, D. De Schreye, and A. de Waal. A Conceptual Embedding of Folding into Partial Deduction: Towards a Maximal Integration. In M. Maher, editor, *Proc. of JICSLP'96*, pages 319–332. The MIT Press, Cambridge, MA, 1996.
25. J.W. Lloyd. Combining Functional and Logic Programming Languages. In *Proc. of the International Logic Programming Symposium*, pages 43–57, 1994.
26. J.W. Lloyd. Declarative Programming in Escher. Technical Report CSTR-95-013, Computer Science Department, University of Bristol, 1995.
27. J.W. Lloyd and J.C. Shepherdson. Partial Evaluation in Logic Programming. *Journal of Logic Programming*, 11:217–242, 1991.
28. R. Loogen, F. López-Fraguas, and M. Rodríguez-Artalejo. A Demand Driven Computation Strategy for Lazy Narrowing. In J. Penjam and M. Bruynooghe, editors, *Proc. of PLILP'93*, pages 184–200. Springer LNCS 714, 1993.
29. J.J. Moreno-Navarro and M. Rodríguez-Artalejo. Logic Programming with Functions and Predicates: The language Babel. *Journal of Logic Programming*, 12(3):191–224, 1992.
30. L. Naish. *Negation and Control in Prolog*. Springer LNCS 238, 1987.
31. L. Naish. Adding equations to NU-Prolog. In J. Maluszyński and M. Wirsing, editors, *Proc. of the 3rd Int'l Symp. on Programming Languages Implementation and Logic Programming*, pages 15–26. Springer LNCS 528, 1991.
32. G. Smolka. The Oz Programming Model. In Jan van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, pages 324–343. Springer LNCS 1000, 1995.