Preliminary version. Final version in New Generation Computing (Ohmsha Ltd.), Vol. 20, No. 1, pp. 3-26, 2002

The Narrowing-Driven Approach to Functional Logic Program Specialization

Elvira ALBERT and Germán VIDAL

DSIC, Universidad Politécnica de Valencia
Camino de Vera s/n, E-46022 Valencia, Spain

{ealbert,gvidal}@dsic.upv.es

Abstract Partial evaluation is a semantics-based program optimization technique which has been investigated within different programming paradigms and applied to a wide variety of languages. Recently, a partial evaluation framework for functional logic programs has been proposed. In this framework, narrowing—the standard operational semantics of integrated languages—is used to drive the partial evaluation process. This paper surveys the essentials of narrowing-driven partial evaluation.

Keywords Program Optimization, Partial Evaluation, Narrowing.

§1 Introduction

Traditionally, the main purpose of partial evaluation is to specialize a given program w.r.t. part of its input data—hence it is also known as *program specialization*. The partially evaluated program will be (hopefully) executed more efficiently since those computations that depend only on the known data are performed—at partial evaluation time—once and for all. From a broader perspective, some partial evaluation techniques are able to optimize programs further by, e.g., shortening computations, removing unnecessary data structures and composing several procedures or functions into a comprehensive definition. Within this broader approach, given a program and a partial call, the essential components of partial evaluation are: the construction of a *finite* representation—generally a graph—of the possible executions of the program call, followed by the systematic extraction of a *residual* program (i.e., the

partially evaluated program) from this graph. Intuitively, optimization can be achieved by compressing paths in the graph and by renaming expressions while removing unnecessary function symbols.

Interest in the development of partial evaluation techniques within the more popular declarative programming paradigms (namely, functional and logic programming) has increased during the last two decades. Recently, a partial evaluation framework for functional logic programs has been proposed.[9, 61] Within this framework, narrowing—the standard operational semantics of functional logic languages—is used to drive the partial evaluation process. In this paper, we survey recent developments in narrowing-driven partial evaluation.

The paper is organized as follows. In Section 2 we recall the basic principles of functional logic languages. Section 3 gives a gentle introduction to narrowing-driven partial evaluation and Section 4 explains how the use of different narrowing strategies interferes with the partial evaluation process. Section 5 introduces a recent extension of the framework which makes it viable for the partial evaluation of *realistic* languages. In Section 6, we present the basic control issues as well as several refinements which provide for more accurate specialization. Section 7 compares narrowing-driven partial evaluation with related works on partial evaluation and, finally, Section 8 concludes.

§2 Functional Logic Programming

In this section we recall some basic notions related to the syntax and semantics of functional logic programs[29] and fix the notation used in the rest of the paper. Term rewriting systems[14, 39, 54] provide an adequate computational model for functional logic languages with functions defined by means of patterns. Thus, in the sequel, we follow mainly the standard framework of term rewriting.

The syntax of functional logic programs is presented in Figure 1. A functional logic *program* $\mathcal{R}$ is a sequence of rewrite rules whose left-hand sides are *functional patterns* and whose right-hand sides are *terms*. Terms are constructed from variables (e.g., x, y, z,...), constructors (e.g., a, b, c,...) and defined functions or operations (e.g., f, g, h,...). Functional patterns are rooted by an operation symbol and all the arguments are constructor terms. A *constructor term* can only contain variables and constructors. To keep things simple, we will only consider *constructor-based* term rewriting systems, i.e., left-hand sides of rules must be functional patterns. In general, this requirement is not necessary under certain evaluation strategies, but we assume it for uniformity.

$$
\begin{array}{llll}
\mathcal{R} & ::= & R_1 \ldots R_m & \text{(program)} \\
R & ::= & l \ \rightarrow\ t & \text{(rule)} \\
l & ::= & f(p_1, \ldots, p_n) & \text{(functional pattern)} \\
t & ::= & x & \text{(variable)} \\
 & | & c(t_1, \ldots, t_n) & \text{(constructor-rooted term)} \\
 & | & f(t_1, \ldots, t_n) & \text{(operation-rooted term)} \\
p & ::= & x & \text{(constructor term or pattern)} \\
 & | & c(p_1, \ldots, p_n) &
\end{array}
$$

Fig. 1 Syntax of functional logic programs

Subterm occurrences are referred to by using the notion of *position*: a sequence of natural numbers where Λ denotes the empty sequence (i.e., the root position). For instance, given a term $t = f(c(x), b)$, the subterm at position 1.1, in symbols $t|_{1.1}$, is x, while the subterm at position Λ, in symbols $t|_\Lambda$, is $f(c(x), b)$. We denote by $t[t']_p$ the result of replacing the subterm of t at position p by the term t'. We denote by $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ the *substitution* σ with $\sigma(x_i) = t_i$ for all $i = 1, \ldots, n$ (with $x_i \neq x_j$ if $i \neq j$) and $\sigma(x) = x$ for any other variable 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 $\gamma \circ \theta = \sigma$ (here "$\circ$" denotes composition). Substitutions are extended to morphisms on terms by $\sigma(f(t_1, \ldots, t_n)) = f(\sigma(t_1), \ldots, \sigma(t_n))$. A substitution σ is *constructor* if $\sigma(x)$ is a constructor term for all x. A term t' is a (constructor) *instance* of t if there is a (constructor) substitution σ with $t' = \sigma(t)$. We say that two terms t_1 and t_2 *unify* if there exists a substitution σ—the *unifier* of t_1 and t_2—such that $\sigma(t_1) = \sigma(t_2)$. A substitution σ is the *most general unifier* of two terms t and t' iff for every other unifier θ of t and t' we have $\sigma \leq \theta$.[42]

A *rewrite step* is an application of a rewrite rule to a term, i.e., $t \rightarrow_{p,R} t'$ is a rewrite step if there is a position p in t, a rewrite rule $R = l \rightarrow t''$ and a substitution σ such that $t|_p = \sigma(l)$ and $t' = t[\sigma(t'')]_p$. The reduced subterm $t|_p$ is called a *redex*. A term t is in *normal form* (w.r.t. a program $\mathcal{R}$) if there is no term t' with $t \rightarrow_{p,R} t'$ for any position p and rule R. We say that a term is a *head normal form* if it is a variable or it is rooted by a constructor symbol. A program is *confluent* if, whenever a term can be rewritten to two terms t_1 and t_2, both t_1 and t_2 can be rewritten to the same term. A program is *noetherian* (or terminating) if there are no infinite sequences of the form $t_1 \rightarrow t_2 \rightarrow t_3 \rightarrow \ldots$

Functional logic languages are extensions of functional languages with principles borrowed from logic programming.[55] Basically, one of the main differences between term rewriting and functional logic programming relies on the fact

that the integrated paradigm allows the use of terms containing *free* variables. In order to evaluate these terms, functional logic languages are usually based on *narrowing*, an extension of rewriting where unification replaces matching: both the rewrite rule and the term can be instantiated. Intuitively, a narrowing step can be seen as a two-phase process: first, some variables of the term are instantiated and, then, a rewrite step is applied to the instantiated term.[29)]

Formally, $t \leadsto_{p,R,\sigma} t'$ is a *narrowing step* if p is a non-variable position of t and $\sigma(t) \to_{p,R} t'$ is a rewrite step.[*1] We will often write $t \leadsto_{\sigma} t'$ when the position and the rule are clear from the context. Trivially, if a term t can be rewritten to t', then t can be also narrowed to t' computing the identity substitution; hence narrowing is a conservative extension of rewriting. Narrowing *derivations* are denoted by $t_0 \leadsto^*_\sigma t_n$, which is a shorthand for the sequence of 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 $\sigma = id$). As in logic programming, narrowing derivations can be represented by a (possibly infinite) finitely branching *tree*. Since we are interested in computing *values* (constructor terms) as well as *answers* (substitutions) in functional logic programming, we only consider narrowing derivations ending in a constructor term, e.g., $t \leadsto^*_\sigma p$, and we say that t *computes the result* p *with answer* σ.

Example 2.1
Consider the well-known operation "app" to concatenate two lists:[*2]

$$\begin{array}{lcl} \texttt{app}\ [\,]\ \texttt{y} & = & \texttt{y} \\ \texttt{app}\ (\texttt{x} : \texttt{x}_\texttt{s})\ \texttt{y} & = & \texttt{x} : \texttt{app}\ \texttt{x}_\texttt{s}\ \texttt{y} \end{array}$$

where we use ":" and "[]" as constructors of lists. Given the term $\texttt{app}\ (\texttt{A} : \texttt{x}_\texttt{s})\ \texttt{y}$, we have the following narrowing derivation (among others):

$$\texttt{app}\ (\texttt{A} : \texttt{x}_\texttt{s})\ \texttt{y} \leadsto_{id} \texttt{A} : \texttt{app}\ \texttt{x}_\texttt{s}\ \texttt{y} \leadsto_{\{\texttt{x}_\texttt{s} \mapsto [\,]\}} \texttt{A} : \texttt{y}$$

Here, we say that the term $\texttt{app}\ (\texttt{A} : \texttt{x}_\texttt{s})\ \texttt{y}$ computes the value $\texttt{A} : \texttt{y}$ with answer substitution $\{\texttt{x}_\texttt{s} \mapsto [\,]\}$.

Narrowing was originally introduced by Slagle[56)] as a theorem proving mechanism. It is a sound and complete method to solve equations w.r.t. a *confluent*

[*1] Usually, σ is obtained by computing the most general unifier between the redex $t|_p$ and the left-hand side of the rule R, restricted to the variables in t.

[*2] Although we consider a first-order representation for programs, we use a curried notation in the examples; additionally, we denote program rules by $\texttt{l} = \texttt{t}$ instead of $l \to t$ and write constructor symbols starting with upper case, as it is usual in functional languages.

and *terminating* set of rules.[36] This justifies the use of narrowing as a basis to execute functional logic programs.[29] Since it was adopted as the standard operational semantics, great effort has been devoted to define sophisticated narrowing *strategies*[13, 23, 26, 47, 51] which are able to avoid useless derivations without losing completeness.

§3 Narrowing-Driven Partial Evaluation

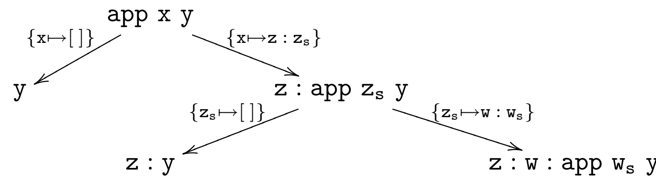
There exist two trends in the field of partial evaluation: *off-line*, staged partial evaluators, and *on-line*, monolithic partial evaluators.[19] In the off-line style, partial evaluation is regarded as a two-phase process: a pre-processing phase, usually based on a *binding-time analysis*,[37] and the proper specialization phase which uses the information gathered by the analysis. In turn, on-line partial evaluators are essentially non-standard interpreters.[19] They evaluate expressions while enough information is available and produce residual code otherwise. The treatment for each expression is determined on the fly, which is in sharp contrast with off-line partial evaluation. Following the on-line approach, we shall present the basic scheme of narrowing-driven partial evaluation.

As in the partial evaluation of logic programs—usually known as *partial deduction*[46]—, given a program and a call, we start by constructing a *finite* (possibly partial) narrowing tree for this call. Then, two situations can arise:

- the narrowing tree represents *all* possible executions of the initial call or
- some leaves of the tree are not properly *covered*.

The first case occurs when all the leaves are either constructor terms—hence not evaluable—or terms whose function calls are *constructor instances* of the root node—hence their executions are particular instances of the constructed tree.

Example 3.1
Consider the rules defining operation `app`. In order to partially evaluate this program w.r.t. the call `app x y`, we start by constructing a narrowing tree, e.g.:



The leftmost leaves, `y` and `z : y`, are constructor terms while the rightmost leaf, `z : w : app` $\mathtt{w_s}$ `y`, is properly covered by the root node since its only operation-

rooted subterm, `app` $\mathtt{w_s}$ `y`, is a constructor instance of the root node `app x y` (indeed, they are equal up to variable renaming).

This example was aimed at showing that, even if the constructed narrowing tree is *finite* and incomplete, it can still represent all possible—*infinitely* many—narrowing derivations for (all constructor instances of) the root node.

Sometimes, though, this ideal situation does not happen and a more complex process is required, as the following example illustrates.

Example 3.2
Consider again program `app` augmented with the following rule defining function `dapp` to concatenate three lists:

```
dapp x y z   =   app (app x y) z
```

Given the call `dapp (1 : x) y z`, we construct the following narrowing tree:

```
  dapp (1 : x) y z
        id |
           v
app (app (1 : x) y) z
        id |
           v
 app (1 : app x y) z
        id |
           v
 1 : app (app x y) z
```

This tree does not represent all possible executions for the initial call, since the leaf node contains an operation-rooted subterm, `app (app x y) z`, which is not a constructor instance of the root node.

When a situation of this kind occurs, we are constrained to continue iteratively with the (partial) evaluation of those operation-rooted subterms which are not covered by the root yet. Our concrete solution allows some degree of freeness when choosing such subterms. In particular, we allow a nondeterministic decomposition of terms before proceeding with their evaluations. For instance, in the above example, we have two alternatives: either to construct a narrowing tree for the whole term, `app (app x y) z`, or construct a narrowing tree for `app x y` (e.g., the one constructed in Example 3.1), since it will safely cover all the computations for (the constructor instances of) `app (app x y) z`. This notion of "coveredness" is formalized by the *closedness test*.[9)]

Definition 3.1 (closedness)
Let S be a finite set of terms. We say that a term t is S-closed iff one of the following conditions hold:

- t is a variable;
- $t = c(t_1, \ldots, t_n)$ is a constructor-rooted term and $t_1, \ldots, t_n$ are S-closed;
- t is an operation-rooted term and there is a term t' in S and a matching substitution σ such that $t = \sigma(t')$ and for all $x \mapsto t''$ in σ, t'' is S-closed.

This definition is nondeterministic when t is an instance of more than one term in S. This is the case, e.g., of the term `app (app x y) z` w.r.t. the set $S = \{$`app` $\mathtt{w_1}$ $\mathtt{w_2}$, `app (app` $\mathtt{w_1}$ $\mathtt{w_2}$`)` $\mathtt{w_3}\}$. As we mentioned, it can be proved S-closed either by checking that it is an instance of `app (app` $\mathtt{w_1}$ $\mathtt{w_2}$`)` $\mathtt{w_3}$ with matching substitution $\{\mathtt{w_1} \mapsto \mathtt{x},\ \mathtt{w_2} \mapsto \mathtt{y},\ \mathtt{w_3} \mapsto \mathtt{z}\}$, where `x`, `y` and `z` are variables, or by checking that it is an instance of `app` $\mathtt{w_1}$ $\mathtt{w_2}$ with matching substitution $\{\mathtt{w_1} \mapsto \mathtt{app\ x\ y},\ \mathtt{w_2} \mapsto \mathtt{z}\}$, where `z` is a variable and `app x y` is recursively S-closed.

A more restrictive notion of closedness is considered in partial deduction[46] and positive supercompilation.[59] According to their conditions, operation-rooted terms are S-closed as long as they are *constructor* instances of some term in S. This implies that a term like $f(g(p))$, where f, g are operation symbols and p is a constructor term, is not closed w.r.t. the set $\{f(x),\ g(y)\}$. In order to avoid this restriction, both frameworks have been refined: *conjunctive* partial deduction[21, 44] allows the *partitioning* of conjunctions of atoms when testing for closedness; analogously, the positive supercompiler of Sørensen and Glück[58] allows the *splitting* of function calls when constructing the partial process trees. The use of a *recursive* notion of closedness in the style of Definition 3.1 makes narrowing-driven partial evaluation as powerful as these refinements.

So far we have presented partial evaluation as the iterative process of constructing (partial) narrowing trees. This first stage stops when all the leaves of the constructed trees are closed w.r.t. the root nodes of these trees.[*3] The closedness test is somehow a means to ensure that a *fold* step—in the sense of Burstall and Darlington[18]—is actually possible. Hence, the whole collection of trees can be organized as a single graph, where some leaves are connected to different root nodes by implicit "fold" arrows. Coming back to the call `dapp (1 :` $\mathtt{x_s}$`) y z`, we continue with the partial evaluation of `app (app x y) z` which, at the same time, brings up the (partial) evaluation of `app x y`. The graph connecting

[*3] For some examples, this simple strategy can lead to an infinite process. Section 6 presents a solution to this problem based on some form of generalization.

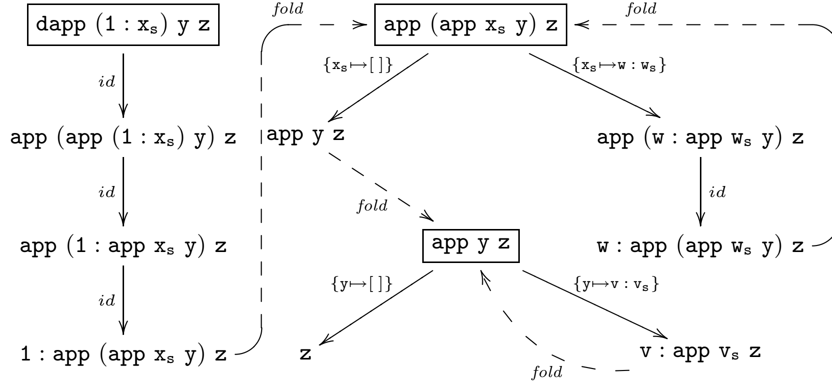


Fig. 2 Narrowing trees for the partial evaluation of dapp (1 : x_s) y z

the three partial narrowing trees for each call is depicted in Figure 2. Here, we have explicitly written the fold arrows showing the terms which have been used to prove the closedness of the different leaves (according to Definition 3.1).

Pettorossi and Proietti[53] introduced the idea that many program transformation techniques can be recast in terms of a three-phase process: symbolic computation, search for regularities, and program extraction. Our symbolic computation mechanism is simply based on the standard semantics of the language, narrowing, while regularities are searched during the iterative construction of partial narrowing trees using the closedness condition. Regarding program extraction, we follow mainly the framework of Lloyd and Shepherdson[46] for partial deduction. In particular, we use the notion of *resultant*[9] to generate a program rule associated to each root-to-leaf derivation of the constructed trees.

Definition 3.2 (resultant)
Let t be a term and $t \leadsto^*_\sigma t'$ a (possibly incomplete) narrowing derivation for t. Then, the associated resultant is the rule: $\sigma(t) \;\to\; t'$.

A *partial evaluation* is then defined as the sequence of resultants associated to the derivations of the constructed partial narrowing trees. For instance, the partial evaluation associated to the narrowing trees of Figure 2 is as follows:

$$\begin{array}{lcl@{\qquad}lcl}
\texttt{dapp (1 : x}_\texttt{s}\texttt{) y z} & = & \texttt{1 : app (app x}_\texttt{s}\texttt{ y) z} & & & \\
\texttt{app (app [ ] y) z} & = & \texttt{app y z} & \texttt{app [ ] z} & = & \texttt{z} \\
\texttt{app (app (w : w}_\texttt{s}\texttt{) y) z} & = & \texttt{w : app (app w}_\texttt{s}\texttt{ y) z} & \texttt{app (v : v}_\texttt{s}\texttt{) z} & = & \texttt{v : app v}_\texttt{s}\texttt{ z}
\end{array}$$

An important observation is that the above sequence of resultants does not fulfill the syntax established in Figure 1. The problem lies in the fact that some

left-hand sides are not functional patterns. In order to recover a legal program, a *post-processing of renaming* is performed. Additionally, this renaming may remove unnecessary function symbols. The post-processing begins by computing an independent renaming for the root nodes of the constructed partial narrowing trees. More formally, an *independent renaming* ρ for a set of terms S is a mapping from terms to terms defined as follows: for each term t in S, $\rho(t) = f_t(x_1, \ldots, x_n)$, where $x_1, \ldots, x_n$ are the distinct variables of t and f_t is a *fresh* function symbol.[8, 11] Arbitrary terms are renamed by means of function ren_ρ, which *recursively* replaces each call in the given term by a call to the corresponding renamed function (according to ρ) as follows:

Definition 3.3 (renaming)
Let S be a finite set of terms and ρ an independent renaming for S. The renaming $ren_\rho(t)$ of the term t under ρ is

- t, if t is a variable;
- $c(ren_\rho(t_1), \ldots, ren_\rho(t_n))$, if $t = c(t_1, \ldots, t_n)$ is constructor-rooted;
- $\sigma'(t')$, if t is operation-rooted and there is a term t_1 in S and a substitution σ with $t = \sigma(t_1)$, $t' = \rho(t_1)$, and $\sigma' = \{x \mapsto ren_\rho(t_2) \mid x \mapsto t_2 \in \sigma\}$.

Something worth noticing is the parallelism between this recursive renaming and the recursive closedness of Definition 3.1. Indeed, if a term t is not S-closed, then it cannot be renamed using an independent renaming for S. Intuitively, the renaming function performs the same steps as the closedness test, but additionally replaces closed subterms by new function calls. Continuing with the previous example, consider the following independent renaming:[*4]

$$\rho = \{ \begin{array}{lcl} \texttt{dapp (1 : x}_\texttt{s}\texttt{) y z} & \mapsto & \texttt{dapp_pe3 x}_\texttt{s}\texttt{ y z}, \\ \texttt{app (app x}_\texttt{s}\texttt{ y) z} & \mapsto & \texttt{app_pe3 x}_\texttt{s}\texttt{ y z}, \\ \texttt{app y z} & \mapsto & \texttt{app_pe2 y z} \end{array} \}$$

for the root nodes of Figure 2. Then, the partial evaluation of `dapp (1 : xs) y z` yields the residual program:

```
dapp_pe3 xs y z      = 1 : app_pe3 xs y z
app_pe3 [] y z       = app_pe2 y z          app_pe2 [] z       = z
app_pe3 (w : ws) y z = w : app_pe3 ws y z   app_pe2 (v : vs) z = v : app_pe2 vs z
```

The partially evaluated program is an *improvement* of the original one for several reasons: first, computations that depend only on the known data—the construc-

[*4] New function names are obtained from the outermost function symbol by concatenating the suffix "`_peN`", where `N` is the arity of the new function.

tor "1" in the first argument of dapp—have been performed at partial evaluation time; second, some computations have been shortened, e.g., for each element of list x_s, the original program performs twice the number of steps than the residual one; also, some functions have been *composed* into a new comprehensive definition, e.g., two nested calls to app are mapped to a single call to the residual function app_pe3; finally, some unnecessary symbols have been removed by renaming, e.g., the constructor "1" in the original call.

By construction, the partial evaluation method sketched throughout this section guarantees that the right-hand sides of the residual rules are closed w.r.t. (the renaming of) the root nodes of the constructed trees. This condition is necessary (but not always sufficient) to demonstrate the correctness of the partial evaluation process. In fact, the conditions for ensuring the correctness are, by the nature of the transformation, dependent on the particular strategy used to construct the narrowing trees. This is meaningful since different narrowing strategies have quite different semantic properties.[29)]

§4 Symbolic Computation

In this section, we briefly review the main features of the different instances of the narrowing-driven partial evaluation framework. They have been developed by considering distinct variants of the narrowing relation. In principle, all the instances consider the same operational mechanism both for standard execution and for partial evaluation. This is an intrinsic feature of the original definition of narrowing-driven partial evaluation.

4.1 Unrestricted Narrowing

The first instance of the framework considered *unrestricted narrowing* to perform partial computations.[9, 10)] The original formulation of narrowing[36, 56)] considers no strategy to select redexes in a term: all of them are candidates to perform an evaluation step (hence it is called "unrestricted"). The generality of unrestricted narrowing introduces some difficulties, e.g., the method is forced to generate a huge number of resultants to ensure correctness results. For example, given the call app (1 : 2 : [ ]) y = z,[*5] unrestricted narrowing yields the following set of answers: {{z ↦ app (1 : 2 : [ ]) y}, {z ↦ 1 : app (2 : [ ]) y}, {z ↦ 1 : 2 : app [ ] y}, {z ↦ 1 : 2 : y}.} Logically, partial evaluation must produce

[*5] Equality is treated by implicitly adding to each program a rule of the form: $x = x \rightarrow true$, which encodes unification.

resultants which ensure the generation of such computed answers. To this end, given a narrowing derivation, it is necessary to produce a resultant not only for the entire derivation but also for any of its "prefixes." This was originally formulated by using equations: to partially evaluate a term t, we construct a narrowing tree for the *equation* $t = x$, where x is a fresh variable not occurring in t. This implies that, at each step, it is always possible to apply the equality rule $x = x \ \rightarrow \ true$ and terminate the derivation, thus producing an associated resultant. For instance, in order to partially evaluate the call app (1 : 2 : [ ]) y, we produce the resultants:

```
app (1 : 2 : [ ]) y  =  1 : app (2 : [ ]) y
app (1 : 2 : [ ]) y  =  1 : 2 : app [ ] y
app (1 : 2 : [ ]) y  =  1 : 2 : y
```

where all the *intermediate* evaluations of the initial call are explicitly encoded by residual rules. This situation can be improved by using *normalization*,[36] where terms are reduced to their normal form—by rewriting—before a narrowing step is actually attempted. This enforces the application of deterministic steps and, consequently, avoids some useless resultants.[10]

4.2 Eager Narrowing

The second instance of the framework employed eager (call-by-value) narrowing[23] to perform computations during partial evaluation.[10] This strategy only applies narrowing steps to innermost positions—e.g., the leftmost ones—of the considered terms. We say that a position is *innermost* if it addresses a functional pattern. Eager narrowing corresponds to the leftmost computation rule of Prolog and to eager evaluation in functional languages.

It is accepted that the use of call-by-value evaluation during partial evaluation is not convenient, e.g., to eliminate intermediate data structures.[57] This is mainly due to the handling of nested calls since constructors cannot be propagated from inner calls—the so-called *producers*—to outer calls—the *consumers*. For instance, reconsider the nested call app (app (1 : x_s) y) z. By only evaluating the inner call to app, the constructor "1" will never be consumed by the outer call to app. Unfortunately, one cannot avoid this undesirable behavior at partial evaluation time—e.g., by considering a call-by-name strategy—without destroying the correctness of the transformation. As in the previous case, the use of *normalization* may alleviate the problem,[10] since rewrite steps are not tied to a call-by-value order.

4.3 Lazy Narrowing

In order to provide for computations with infinite data structures as well as a demand-driven generation of the search space, recent research has advocated lazy (call-by-name) narrowing strategies.[26, 47, 51] This motivated the investigation on partial evaluation based on lazy narrowing.[8, 38] This narrowing strategy first tries to apply a narrowing step to the outermost position of the term. When the attempted unification does not succeed due to a clash between a constructor of the rule and an operation symbol of the term, further evaluation of the operation-rooted subterm is *demanded*. Only in this situation, narrowing steps at inner positions will be allowed. Lazy strategies do not require terminating programs and, thus, the equality predicate is defined, like in functional languages, as the *strict equality* on terms.[13, 26, 51]

The use of lazy narrowing during partial evaluation gives a better overall behavior w.r.t. previous instances of the framework regarding both the elimination of intermediate data structures and the propagation of information.[8, 38] Unfortunately, this approach also introduces new drawbacks into the partial evaluation process. Firstly, the class of programs is not preserved by the transformation. Lazy narrowing requires *orthogonal* programs—informally, a program is orthogonal if there are no rules whose left-hand sides unify[51]—to ensure the completeness of the strategy. However, partial evaluation based on lazy narrowing may destroy the orthogonality of the residual program.[11] This prevents us from using lazy narrowing to evaluate terms in the partially evaluated program. Secondly, terms in *head normal form* cannot be evaluated at partial evaluation time. This restriction is imposed because the *backpropagation* of bindings can incorrectly restrict the domain of residual functions within a lazy context.[8]

Example 4.1
Consider a program containing the following rules:

$$\begin{array}{lcl} \texttt{foo x} & = & (\texttt{isZero x}) : [\,] \\ \texttt{isZero Z} & = & \texttt{True} \end{array}$$

where we use "Z" and "Succ" to construct natural numbers. Given the (unique) computation for foo y:

$$\texttt{foo y} \leadsto_{id} (\texttt{isZero y}) : [\,] \leadsto_{\{\texttt{y} \mapsto \texttt{Z}\}} \texttt{True} : [\,]$$

we produce the resultant: $\texttt{foo Z} = \texttt{True} : [\,]$. Here, the backpropagation of binding $\{\texttt{y} \mapsto \texttt{Z}\}$ to the left-hand side of the residual rule has incorrectly restricted the domain of function foo (regarding the computation of head normal

forms). For example, the term `foo (Succ Z)` can be reduced to a head normal form, (`isZero (Succ Z)`) : [ ], using the original functions, whereas it is not possible using the residual rule.

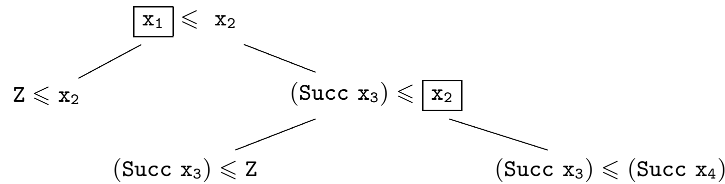
Finally, lazy narrowing may compute "redundant" derivations since some steps in a derivation may not contribute to the computation of the final value; indeed, this property is undecidable for the class of orthogonal programs. This could degrade the quality of partially evaluated programs by introducing, e.g., infinite derivations which could not be performed in the original program.[11]

4.4 Needed Narrowing

The next instance of the framework used needed narrowing to perform partial evaluations.[11] Needed narrowing is considered an optimal evaluation strategy w.r.t. both the length of derivations and the independence of computed solutions.[13] It extends the Huet and Lévy's notion of a needed rewrite step[35] by allowing the instantiation of variables in input expressions. An efficient implementation of needed narrowing exists for the class of *inductively sequential* programs.[12] Intuitively, a program is *inductively sequential* when all its operations can be defined by rewrite rules that, recursively, make a case distinction on their arguments, analogously to a data type induction. Consider, for instance, the following rules defining the Boolean function "$\leqslant$":

$$
\begin{array}{rcl}
\mathtt{Z} \leqslant \mathtt{x} & = & \mathtt{True} \\
(\mathtt{Succ\ x}) \leqslant \mathtt{Z} & = & \mathtt{False} \\
(\mathtt{Succ\ x}) \leqslant (\mathtt{Succ\ y}) & = & \mathtt{x} \leqslant \mathtt{y}
\end{array}
$$

Their rules can be organized in a hierarchical structure called a *definitional tree*.[12] A definitional tree for operation "$\leqslant$" above is the following:



The leaves of this tree represent (modulo a renaming of variables) the left-hand sides of the corresponding rewrite rules. Operation "$\leqslant$" makes an initial case distinction on its first argument. Then, it scrutinizes the second one, leading to the remaining rewrite rules. The argument which is the subject of a case distinc-

tion is rounded by a box and is usually called the *inductive position*. Definitional trees are used to guide the needed narrowing strategy. For instance, to evaluate an expression like $t_1 \leqslant t_2$, we first evaluate t_1 to a head normal form. If the result is Z, we will apply the first rule. If the result is of the form (Succ t), we will evaluate t_2 to a head normal form in hopes of determining the rule to fire.

The use of needed narrowing to perform partial evaluation presents several advantages w.r.t. previous approaches.[11] Firstly, the inductively sequential structure of programs is preserved by the partial evaluation process. Furthermore, terms which are *deterministically* evaluable—i.e., only one redex is exploited at each step of their evaluation—in the original program are also deterministically evaluable in the residual program, which is important from an implementation point of view. This means that no redundant derivations will be encoded in the residual program, as it could happen in the lazy instance.[11]

4.5 Needed Narrowing + Residuation

The computational model of modern multi-paradigm declarative languages, which integrate the most important features of functional, logic and concurrent programming, is based on a combination of two different operational principles: narrowing and residuation.[30] The *residuation* principle is based on the idea of delaying function calls until they are sufficiently instantiated for a deterministic evaluation by rewriting. The particular mechanism (narrowing or residuation) is specified by *evaluation annotations*: deterministic functions are annotated as *rigid* (which forces a delayed evaluation by rewriting), while non-deterministic functions are annotated as *flexible* (which enables narrowing steps). Concurrency is expressed by a built-in operator "&" which evaluates its two arguments concurrently. In this setting, the evaluation of a term may *suspend* when it contains rigid function calls which are not sufficiently instantiated.

Narrowing-driven partial evaluation was originally formulated for functional logic languages based uniquely on narrowing. Nevertheless, it was still possible to accommodate the use of distinct operational mechanisms. Along this line, the framework has been adjusted to perform partial evaluations using the combined operational semantics described above.[2, 4] More precisely, the actual proposal allows the use of needed narrowing even to evaluate *rigid* function calls during partial evaluation. This helps to propagate more information, but the resulting scheme does not use the standard semantics during partial evaluation anymore, which contrasts with previous approaches. Now, the difficulty lies in

preserving the *floundering* behavior of the original program, i.e., ensuring that computations suspend in the original program iff they suspend in the residual one. Roughly speaking, the proposed solution consists in splitting resultants—by introducing *intermediate* functions with appropriate evaluation annotations—so that bindings coming from the evaluation of both flexible and rigid function calls are distinguished.[4] Etalle, Gabbrielli and Marchiori[22] present a transformation framework for the partial evaluation of concurrent logic programs. In contrast to the above proposal, they simply stop suspended computations during partial evaluation. By allowing the evaluation of suspended computations, we get a more accurate propagation of information at the price of a more complex generation of resultants.

§5 An Extension of the Framework

The narrowing-driven methodology has been extended[6, 7] in order to make it viable for defining partial evaluators for *realistic* multi-paradigm functional logic languages like Curry[33] or TOY.[48] When one considers a realistic language, several high-level constructs have to be considered, e.g., higher-order functions, concurrent constraints, calls to external functions, etc. In order to deal with these additional features, the underlying operational calculus becomes usually more complex. As we mentioned before, an on-line partial evaluator normally includes an interpreter of the language.[19] Then, as the operational semantics becomes more elaborated, the associated partial evaluation techniques become (more powerful but) also increasingly more complex. To avoid this problem, an approach successfully tested in other contexts[16, 27, 52] is to consider the partial evaluation of programs written in a maximally simplified programming language. It is expected that programs written in a higher-level language can be automatically translated to the simpler language.

Hanus and Prehofer[32] have introduced a *flat* representation for functional logic programs in which definitional trees—used to guide the needed narrowing strategy—are embedded in the rewrite rules by means of case expressions. In particular, functions are defined by a single rule whose left-hand side contains only variables as parameters and the right-hand side is a term composed by variables, constructors, function calls, and case expressions for pattern-matching.

Example 5.1
Function "$\leqslant$" of Section 4.4. can be written in the flat representation as follows:

$$\begin{array}{llllll} \texttt{x} \leqslant \texttt{y} = & \texttt{case x of} & \{0 & \to & \texttt{True}; & \\ & & (\texttt{Succ}\ \texttt{x}_1) & \to & \texttt{case y of} & \{0 \to \texttt{False}; \\ & & & & & (\texttt{Succ}\ \texttt{y}_1) \to \ \texttt{x}_1 \leqslant \texttt{y}_1\}\ \} \end{array}$$

Two nice properties of the flat representation are that it provides more explicit control—hence the associated calculus is simpler than needed narrowing—and source programs can be automatically translated to the new representation. Moreover, it constitutes the basis of a recent proposal for an intermediate language, FlatCurry, used during the compilation of Curry programs.[31] A new partial evaluation scheme[2, 6, 7] has been designed by considering such a *flat* representation for functional logic programs.

However, the use of the standard semantics for flat programs—the LNT calculus[32]—during partial evaluation does not avoid the backpropagation of bindings when evaluating terms in head normal form, which can be problematic within a lazy context (see Example 4.1). To overcome this problem, a *residualizing* version of the standard semantics is introduced: the RLNT calculus.[2, 6, 7] If one does not consider the distinction between *flexible* and *rigid* functions, the resulting RLNT calculus happens to be essentially equivalent to the *driving* mechanism of Sørensen, Glück and Jones[59] (although we obtained it independently by refining the original LNT calculus to avoid the backpropagation of bindings). This does not mean that positive supercompilation and the new partial evaluation scheme based on the RLNT calculus behave identically. There is still a significant difference regarding control issues (see Section 7).

Finally, since modern lazy functional logic languages can be automatically translated to this flat representation—which still contains all the necessary information about programs—, the resulting technique is widely applicable. Let us remark that, within this scheme, residual programs will be also written in the flat syntax. Nonetheless, this is not a restriction because existing compilers rely on a similar representation for the intermediate code.[31, 34, 49] Rather, the partial evaluation process can be seen as a source-to-source optimization phase (transparent to the user) performed during the compilation of the program.[7]

§6 Control Issues

At the beginning of Section 3, we outlined a partial evaluation algorithm. Basically, it consists of iteratively constructing partial narrowing trees until all their leaves are covered by the root nodes. This section presents such an algorithm in more detail and discusses several improvements.

Input: a program $\mathcal{R}$ and a set of terms T
Output: a set of terms S
Initialization: $i := 0;\ T_0 := T$
Repeat
  $\mathcal{R}' := \mathit{unfold}(T_i, \mathcal{R});$
  $T_{i+1} := \mathit{abstract}(T_i, \mathcal{R}'_{calls});$
  $i := i + 1;$
Until $T_i = T_{i-1}$ (modulo renaming)
Return: $S := T_i$

Fig. 3 Basic algorithm for narrowing-driven partial evaluation

6.1 The Basic Algorithm

The original formulation[9, 10] is parametric w.r.t. the *narrowing relation* which is used to construct the partial trees, the *unfolding rule* which determines when and how to terminate the construction of these trees, and the *abstraction operator* which is used to guarantee that the number of trees is kept finite. The procedure is formalized in Figure 3.[*6] Informally, the algorithm proceeds as follows. Given an input program $\mathcal{R}$ and a set of terms T, the first step consists on applying an unfolding rule to compute finite (possibly incomplete) narrowing trees for these terms; it returns the set of resultants—i.e., a program—associated to the root-to-leaf derivations of these trees. Then, an abstraction operator is applied to properly add the terms in the right-hand sides of resultants to the set of terms already partially evaluated; the abstraction phase yields a new set of terms which may need further evaluation and, thus, the process is iteratively repeated while new terms are introduced. Let us note that the algorithm does not return a partially evaluated program but a set of terms which unambiguously determines the associated partial evaluation. In particular, by applying once the same unfolding rule, we generate the corresponding resultants which form the residual program. Renaming is then applied over this residual program.

The procedure follows the style of Gallagher's partial deduction method[25] and two control levels are clearly distinguished: the *local level*—which is managed by an unfolding rule—and the *global level*—which is controlled by an abstraction operator. Trivially, in order to ensure the termination of the algorithm, we must ensure both *local* and *global* termination, i.e., partial narrowing trees must be finite and the iterative construction of partial trees must eventually terminate. The remaining of this section provides some insights on both control levels as well as on their termination.

[*6] By $\mathcal{R}_{calls}$ we denote the set of terms in the right-hand sides of the rules of $\mathcal{R}$.

In order to ensure the local termination of the algorithm, the unfolding rule must incorporate some mechanism to stop the construction of narrowing trees. For this purpose, there exist several well-known techniques in the literature, e.g., depth-bounds, loop-checks,[15] well-founded orderings,[17] well-quasi orderings,[58] etc. Within the narrowing-driven approach, unfolding rules have been defined by using a particular type of well-quasi ordering: *homeomorphic embedding*.[40] Nowadays, it is broadly used in the context of on-line program manipulation techniques.[21, 41, 45, 58] The interested reader is referred to Leuschel's work,[43] where a detailed description of homeomorphic embedding can be found. Informally, term t_1 *embeds* term t_2 if t_2 can be obtained from t_1 by deleting some operators, e.g., $\mathtt{Succ}\ (\underline{\mathtt{Succ}}\ ((\underline{\mathtt{u}} + \mathtt{w})\underline{\times}(\underline{\mathtt{u}+}(\mathtt{Succ}\ \underline{\mathtt{v}}))))$ embeds $\mathtt{Succ}\ (\mathtt{u} \times (\mathtt{u} + \mathtt{v}))$.

Unfolding rules based on the embedding ordering allow the expansion of narrowing derivations until reaching a term which embeds a previous term of the same derivation.[3, 9, 10] An important remark is that the embedding test can be applied either on *terms* or on *redexes*; namely, given a narrowing derivation:

$$t_1 \ \leadsto_{p_1,R_1,\sigma_1} \ t_2 \ \leadsto_{p_2,R_2,\sigma_2} \ \cdots \ \leadsto_{p_n,R_n,\sigma_n} t_{n+1}$$

in order to decide whether to evaluate t_{n+1} or not, we can check either that i) t_{n+1} does not embed any term in the sequence $t_1, \ldots, t_n$, or that ii) no redex in t_{n+1} embeds any previous redex $t_1|_{p_1}, \ldots, t_n|_{p_n}$. In both cases, the test is only applied on *comparable* terms, i.e., terms with the same outermost function symbol. In narrowing-driven partial evaluation, the best results were achieved by using an embedding ordering on comparable redexes.[10]

Global control cannot be managed with the same flexibility as the local control if we want to preserve the correctness of the method. At the local level, this flexibility allows us to safely stop the construction of a tree at any point. In contrast, we cannot stop the iterative construction of partial trees until all their leaves are closed w.r.t. the corresponding set of root nodes. This condition is necessary to ensure that the resulting partial evaluation is closed and, hence, correctness is guaranteed. On the other hand, it may also happen that this condition is never reached and, in this case, the iterative process runs forever. Therefore, global control usually includes some kind of generalization to enforce the termination of the process. The most popular generalization operator is the *msg* (*most specific generalization*) between terms. We say that a term t is a *generalization* of terms t_1 and t_2 if both t_1 and t_2 are instances of t. Furthermore, term t is the *msg* of t_1 and t_2 if t is a generalization of t_1 and t_2 and, for any other generalization t' of t_1 and t_2, t is an instance of t'.

Restricting attention to the narrowing-driven approach, abstraction operators also use the embedding ordering to decide when to generalize and when to continue with the iterative construction of partial narrowing trees.[*7] The abstraction operator *abstract* takes two sets of terms (the terms already partially evaluated T_i and the terms to be added to this set, $\mathcal{R}'_{calls}$, as shown in Figure 3) and returns a *safe* approximation of $T_i \cup \mathcal{R}'_{calls}$. By safe we mean that each term in $T_i \cup \mathcal{R}'_{calls}$ is closed w.r.t. the set of terms resulting from $abstract(T_i, \mathcal{R}'_{calls})$. In order to add a new term, t, to the current set of partially evaluated terms, S, existing abstraction operators essentially proceed as follows:[9, 10]

- if t does not embed any term in S, then t is just added to S;
- if t embeds some term in S, say t', we distinguish two cases:
  - if t is already S-closed, then it is simply discarded;
  - otherwise, both terms are generalized by computing the *msg* of t and t', say t'', and the abstraction operator is recursively applied to add t'' as well as the terms in the matching substitutions (i.e., terms in σ and in σ', with $\sigma(t'') = t$ and $\sigma'(t'') = t'$).

An important remark is that the application of the *msg* should be delayed until it is really unavoidable, since some information might be lost due to the generalization of the involved terms.

6.2 Refinements on the Basic Algorithm

The basic partial evaluation algorithm does not make any distinction between *primitive* symbols—i.e., symbols predefined by the language environment—and user-defined function symbols. In principle, primitive functions can be treated either as user-defined functions—by evaluating them at partial evaluation time—or as constructors—by leaving them as residual functions—since their definitions will be available in the residual program anyway. Unfortunately, the efficiency of partial evaluation might be degraded by always treating them as user-defined functions.

Example 6.1

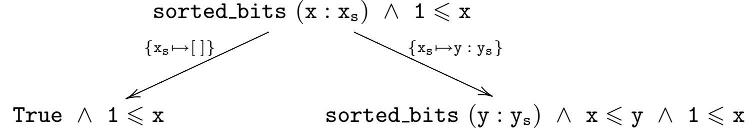
Let us consider the following program excerpt:[3)]

$$\begin{array}{lcl} \texttt{sorted_bits}\ (\texttt{x} : [\,]) & = & \texttt{True} \\ \texttt{sorted_bits}\ (\texttt{x}_1 : \texttt{x}_2 : \texttt{x}_s) & = & \texttt{sorted_bits}\ (\texttt{x}_2 : \texttt{x}_s)\ \wedge\ \texttt{x}_1 \leqslant \texttt{x}_2 \end{array}$$

[*7] For simplicity, in Figure 3 we considered that the current collection of partially evaluated terms is represented by means of a set. A more precise treatment can be easily given by using sequences of terms[9, 10] or *global trees*[50] instead of sets.

$$0 \leqslant 0 \;=\; \texttt{True} \qquad 0 \leqslant 1 \;=\; \texttt{True} \qquad 1 \leqslant 1 \;=\; \texttt{True}$$

where conjunction, "∧", is a primitive symbol of the language. Given the following partial narrowing tree for the call $\texttt{sorted_bits}\ (\texttt{x} : \texttt{x}_\texttt{s})\ \wedge\ \texttt{1} \leq \texttt{x}$:[*8]



it brings to light two weak points of the basic algorithm. Firstly, the rightmost branch has been stopped because the leftmost redex $\texttt{sorted_bits}\ (\texttt{y} : \texttt{y}_\texttt{s})$ of the leaf *embeds* the redex $\texttt{sorted_bits}\ (\texttt{x} : \texttt{x}_\texttt{s})$ selected in the previous step. However, the remaining conjuncts have not been evaluated at all, which does not seem very fair. On the other hand, when the control is passed to the global level, the abstraction operator will attempt to add the term $\texttt{sorted_bits}\ (\texttt{y} : \texttt{y}_\texttt{s})\ \wedge\ \texttt{x} \leq \texttt{y}\ \wedge\ \texttt{1} \leq \texttt{x}$ to the current set $\{\texttt{sorted_bits}\ (\texttt{x} : \texttt{x}_\texttt{s})\ \wedge\ \texttt{1} \leq \texttt{x}\}$. The new call embeds the previous one but it is not closed. Therefore, the *msg* $\texttt{sorted_bits}\ (\texttt{x} : \texttt{x}_\texttt{s})\ \wedge\ \texttt{z}$ will be computed, which gives up the intended specialization (e.g., the communication between variables will be lost).

The first drawback motivates the definition of an unfolding rule able to achieve a *balanced* evaluation. To this end, some form of *dynamic scheduling* was introduced in the unfolding rule.[3)] More precisely, when several redexes can be narrowed in a don't-care nondeterministic way—which is the case, e.g., of the elements in a conjunction—, it selects one among those not embedding any previous redex. For instance, in the previous example, this dynamic rule allows us to continue further the evaluation of the rightmost branch by selecting the second or third conjuncts, since they do not embed any previous selected redex.

The second drawback suggests the definition of an abstraction operator able to *split* complex terms before attempting their generalization. Inspired by the partitioning techniques of conjunctive partial deduction,[21)] a more concerned abstraction operator was introduced.[3)] The main difference with the previous one lies in the treatment of terms rooted by a primitive function symbol. For these terms, it now considers two possible actions: ignore the primitive symbol and try to add the arguments of the term—sort of *splitting*—or consider it as an arbitrary operation-rooted term. The concrete action will be the one involving the smaller loss of information.[3)]

[*8] A fixed left-to-right selection of components within conjunctions is assumed.

§7 Related Work

The work by Darlington and Pull[20] is the most clear antecedent of narrowing-driven partial evaluation. They proposed the use of narrowing as an alternative to the combination of *instantiation* and *unfolding*—in the sense of Burstall and Darlington[18]—to perform partial evaluation. Their approach yielded a partial evaluator for the functional language HOPE extended with unification. From now on, we center the discussion on two partial evaluation methods which are, in our opinion, the closest to narrowing-driven partial evaluation: positive supercompilation[59] and conjunctive partial deduction.[21] A more exhaustive comparison was presented by Alpuente, Falaschi and Vidal.[10]

Positive supercompilation[59] is a program transformation technique for functional programs which is based on *driving* (originally introduced in Turchin's supercompilation[60]). Driving is a unification-based function evaluation mechanism closely related to narrowing. In contrast to narrowing-driven partial evaluation, local and global control are not (explicitly) distinguished in positive supercompilation; in other words, only one-step unfolding is performed at the local level and, thus, only the global level matters. Their method builds a large evaluation structure which somehow subsumes both our local narrowing trees and the global trees of Martens and Gallagher.[50] In order to ensure the finiteness of this structure, they use an embedding ordering to stop driving while Martens and Gallagher advocated the use of well-founded orders.

Narrowing and driving are similar mechanisms; indeed, the *perfect* driving used to perform inverse computations[1] basically coincides with needed narrowing. Therefore, it is reasonable that the resulting partial evaluation methods share many similarities too. Nevertheless, our "root-to-leaf" construction of resultants—similar to partial deduction—still represents a significant difference with positive supercompilation. In particular, it gives rise to fewer rules than in the case of positive supercompilation, where rules are extracted from each single computation step. Hence, in their case, a postunfolding phase of simplification is usually required to remove redundant resultants. As a counterpart, we have less opportunities to find "regularities" (in the sense of Pettorossi and Proietti[53]), just because fewer residual functions are produced. Another difference lies in our recursive notion of closedness (cf. Definition 3.1), which subsumes both the *perfect* ("α-identical") closedness test of positive supercompilation[27] and the standard notion of closedness in partial deduction.[46] This enhances the optimization power of narrowing-driven partial evaluation since it supports a sort of

implicit partitioning for terms. Sørensen and Glück[58] achieve the same potential in positive supercompilation by introducing an extended *folding* operator.

A precise correspondence between partial deduction[46] and driving was stated by Glück and Sørensen.[28] They focused on the similarities between the driving of a functional program and the construction of an SLD-tree for a similar Prolog program. This correspondence can be straightforwardly extended to the evaluation of functional logic programs by narrowing. Furthermore, by virtue of our recursive notion of closedness, the basic narrowing-driven partial evaluation method is essentially as powerful as conjunctive partial deduction.[21, 44] Indeed, one can establish a clear correspondence between the partitioning techniques used in conjunctive partial deduction and the use of a recursive notion of closedness. In essence, both techniques encode a form of compositionality, i.e., an expression—a conjunction in partial deduction or a term in the narrowing-driven approach—is considered closed whenever its components—the conjuncts or subterms—are also closed.

§8 Conclusions

This paper surveys recent developments in the field of partial evaluation for functional logic languages with an operational semantics based on narrowing. The practicality of these ideas is witnessed by the implementation of a partial evaluator for the multi-paradigm language Curry,[33] written in Curry itself.[7] This system has been fully integrated into the last distribution of the PAKCS[31] programming environment for the language and it is publicly available at `http://www.dsic.upv.es/users/elp/peval/peval.html`.

There are several challenging topics related to narrowing-driven partial evaluation which are interesting for further research:

- *Assessment of the efficiency achieved by partial evaluation.* The estimation of the speedup achieved by partial evaluators is of utmost importance since it constitutes the main motivation for this transformation.
- *Mixing on-line/off-line.* Binding-time analyses used in off-line specialization can be used to improve the efficiency of on-line partial evaluators.[19]
- *Self-application.*[16, 24, 52] This is an interesting application of partial evaluation which allows, e.g., the generation of compilers from interpreters.

A first step towards the estimation of the improvement achieved by narrowing-driven partial evaluation has been taken by Albert, Antoy and Vidal.[5]

Acknowledgment

This work greatly benefits from joint work with María Alpuente, Sergio Antoy, Moreno Falaschi, Michael Hanus, Pascual Julián, Salvador Lucas and Ginés Moreno. Our sincere gratitude to all of them. Thanks also to Robert Glück and Yoshihiko Futamura for stimulating discussions during a short visit of the authors at Waseda University supported by the Japan Society for the Promotion of Science. Finally, we would like to thank the participants of the *Partial Evaluation and Program Transformation Day* (Waseda University, Tokyo, 1999) for useful comments and suggestions on an earlier version of this paper. This work has been partially supported by Spanish CICYT TIC 98-0445-C03-01.

References

1) Abramov S. and Glück R., "The Universal Resolving Algorithm: Inverse Computation in a Functional Language," in *Proc. of the 5th Int'l Conf. on Mathematics of Program Construction*, Springer LNCS 1837, pp. 187–212, 2000.
2) Albert E., *Partial Evaluation of Multi-Paradigm Declarative Languages: Foundations, Control, Algorithms and Efficiency*, Ph.D. thesis, DSIC, Universidad Politécnica de Valencia, 2001. Available from URL: `http://www.dsic.upv.es/users/elp/papers.html`.
3) Albert E., Alpuente M., Falaschi M., Julián P., and Vidal G., "Improving Control in Functional Logic Program Specialization," in *Proc. of the 5th Int'l Static Analysis Symposium (SAS'98)*, Springer LNCS 1503, pp. 262–277, 1998.
4) Albert E., Alpuente M., Hanus M., and Vidal G., "A Partial Evaluation Framework for Curry Programs," in *Proc. of the 6th Int'l Conf. on Logic Programming and Automated Reasoning (LPAR'99)*, Springer LNAI 1705, pp. 376–395, 1999.
5) Albert E., Antoy S., and Vidal G., "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)*, Springer LNCS 2042, pp. 103–124, 2001.
6) Albert E., Hanus M., and Vidal G., "Using an Abstract Representation to Specialize Functional Logic Programs," in *Proc. of the 7th Int'l Conf. on Logic for Programming and Automated Reasoning (LPAR'2000)*, Springer LNAI 1955, pp. 381–398, 2000.
7) Albert E., Hanus M., and Vidal G., "A Practical Partial Evaluator for a Multi-Paradigm Declarative Language," in *Proc. of the 4th Fuji Int'l Symp. on Functional and Logic Programming (FLOPS'2001)*, Springer LNCS 2024, pp. 326–342, 2001.
8) Alpuente M., Falaschi M., Julián P., and Vidal G., "Specialization of Lazy Functional Logic Programs," in *Proc. of the ACM SIGPLAN Symp. on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'97)*, volume 32, 12 of *Sigplan Notices*, ACM Press, pp. 151–162, 1997.
9) Alpuente M., Falaschi M., and Vidal G., "Narrowing-driven Partial Evaluation of Functional Logic Programs," in *Proc. of the 6th European Symposium on Programming (ESOP'96)*, Springer LNCS 1058, pp. 45–61, 1996.

10) Alpuente M., Falaschi M., and Vidal G., "Partial Evaluation of Functional Logic Programs," *ACM Transactions on Programming Languages and Systems*, *20*, *4*, pp. 768–844, 1998.
11) Alpuente M., Hanus M., Lucas S., and Vidal G., "Specialization of Inductively Sequential Functional Logic Programs," in *Proc. of the 4th ACM SIGPLAN Int'l Conf. on Functional Programming (ICFP'99)*, volume 34, 9 of *Sigplan Notices*, ACM Press, pp. 273–283, 1999.
12) Antoy S., "Definitional Trees," in *Proc. of the 3rd Int'l Conf. on Algebraic and Logic Programming (ALP'92)*, Springer LNCS 632, pp. 143–157, 1992.
13) Antoy S., Echahed R., and Hanus M., "A Needed Narrowing Strategy," *Journal of the ACM*, *47*, *4*, pp. 776–822, 2000.
14) Baader F. and Nipkow T., *Term Rewriting and All That*, Cambridge University Press, 1998.
15) Bol R., "Loop Checking in Partial Deduction," *Journal of Logic Programming*, *16*, *1&2*, pp. 25–46, 1993.
16) Bondorf A., "A Self-Applicable Partial Evaluator for Term Rewriting Systems," in *Proc. of the Int'l Joint Conf. on Theory and Practice of Software Development (TAPSOFT'89)*, Springer LNCS 352, pp. 81–95, 1989.
17) Bruynooghe M., De Schreye D., and Martens B., "A General Criterion for Avoiding Infinite Unfolding," *New Generation Computing*, *11*, *1*, pp. 47–79, 1992.
18) Burstall R. and Darlington J., "A Transformation System for Developing Recursive Programs," *Journal of the ACM*, *24*, *1*, pp. 44–67, 1977.
19) Consel C. and Danvy O., "Tutorial notes on Partial Evaluation," in *Proc. of the 20th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL'93)*, ACM Press, pp. 493–501, 1993.
20) Darlington J. and Pull H., "A Program Development Methodology Based on a Unified Approach to Execution and Transformation," in *Proc. of the IFIP TC2 Workshop on Partial Evaluation and Mixed Computation*, North-Holland, pp. 117–131, 1988.
21) De Schreye D., Glück R., Jørgensen J., Leuschel M., Martens B., and Sørensen M., "Conjunctive Partial Deduction: Foundations, Control, Algorihtms, and Experiments," *Journal of Logic Programming*, *41*, *2&3*, pp. 231–277, 1999.
22) Etalle S., Gabbrielli M., and Marchiori E., "A Transformation System for CLP with Dynamic Scheduling and CCP," in *Proc. of the ACM SIGPLAN Symp. on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'97)*, volume 32, 12 of *Sigplan Notices*, ACM Press, pp. 137–150, 1997.
23) Fribourg L., "SLOG: A Logic Programming Language Interpreter Based on Clausal Superposition and Rewriting," in *Proc. of the 2nd IEEE Int'l Symp. on Logic Programming*, IEEE Press, pp. 172–185, 1985.
24) Futamura Y., "Partial Evaluation of Computation Process—An Approach to a Compiler-Compiler," *Higher-Order and Symbolic Computation*, *12*, *4*, pp. 381–391, 1999. Reprint of article in *Systems, Computers, Controls* 1971.
25) Gallagher J., "Tutorial on Specialisation of Logic Programs," in *Proc. of the ACM SIGPLAN Symp. on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'93)*, pp. 88–98, ACM Press, 1993.
26) Giovannetti E., Levi G., Moiso C., and Palamidessi C., "Kernel Leaf: A Logic plus Functional Language," *Journal of Computer and System Sciences*, *42*, pp. 363–377, 1991.

27) Glück R. and Klimov A., "Occam's Razor in Metacomputation: the Notion of a Perfect Process Tree," in *Proc. of the 3rd Int'l Workshop on Static Analysis*, Springer LNCS 724, pp. 112–123, 1993.

28) Glück R. and Sørensen M., "Partial Deduction and Driving are Equivalent," in *Proc. of the 6th Int'l Symp. on Programming Language Implementation and Logic Programming (PLILP'94)*, Springer LNCS 844, pp. 165–181, 1994.

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

30) Hanus M., "A Unified Computation Model for Functional and Logic Programming," in *Proc. of 24th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL'97)*, ACM Press, pp. 80–93, 1997.

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

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

33) Hanus (ed.) M., "Curry: An Integrated Functional Logic Language," Available at `http://www.informatik.uni-kiel.de/~curry/report.html`, 2000.

34) Hortalá-González T. and Ullán E., "An Abstract Machine Based System for a Lazy Narrowing Calculus," in *Proc. of the 4th Fuji Int'l Symp. on Functional and Logic Programming (FLOPS'2001)*, pp. 216–232, Springer LNCS 2024, 2001.

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

36) Hullot J., "Canonical Forms and Unification," in *Proc. of the 5th Int'l Conf. on Automated Deduction*, Springer LNCS 87, pp. 318–334, 1980.

37) Jones N., Sestoft P., and Søndergaard H., "Mix: A Self-Applicable Partial Evaluator for Experiments in Compiler Generation," *Lisp and Symbolic Computation*, *2*, *1*, pp. 9–50, 1989.

38) Julián-Iranzo P., *Specialization of Lazy Functional Logic Programs*, Ph.D. thesis, DSIC, Universidad Politécnica de Valencia, May 2000. In Spanish.

39) Klop J., "Term Rewriting Systems," in *Handbook of Logic in Computer Science* (S. Abramsky, D. Gabbay, and T. Maibaum, eds.), volume I, Oxford University Press, pp. 1–112, 1992.

40) Kruskal J., "Well-quasi-ordering, the tree theorem, and Vazsonyi's conjecture," *Transactions of the American Mathematical Society*, *95*, pp. 210–225, 1960.

41) Lafave L. and Gallagher J., "Partial Evaluation of Functional Logic Programs in Rewriting-based Languages," Technical Report CSTR-97-001, Department of Computer Science, University of Bristol, England, March 1997.

42) Lassez J.L., Maher M.J., and Marriott K., "Unification Revisited," in *Foundations of Deductive Databases and Logic Programming* (J. Minker, ed.), Morgan Kaufmann, Los Altos, Ca., pp. 587–625, 1988.

43) Leuschel M., "On the Power of Homeomorphic Embedding for Online Termination," in *Proc. of the 5th Int'l Static Analysis Symposium (SAS'98)*, Springer LNCS 1503, pp. 230–245, 1998.

44) Leuschel M., De Schreye D., and de Waal A., "A Conceptual Embedding of Folding into Partial Deduction: Towards a Maximal Integration," in *Proc. of*

the 1996 Joint Int'l Conf. and Symp. on Logic Programming, The MIT Press, Cambridge, MA, pp. 319–332, 1996.

45) Leuschel M., Martens B., and De Schreye D., "Controlling Generalization and Polyvariance in Partial Deduction of Normal Logic Programs," *ACM Transactions on Programming Languages and Systems*, *20*, *1*, pp. 208–258, 1998.

46) Lloyd J. and Shepherdson J., "Partial Evaluation in Logic Programming," *Journal of Logic Programming*, *11*, pp. 217–242, 1991.

47) Loogen R., López-Fraguas F., and Rodríguez-Artalejo M., "A Demand Driven Computation Strategy for Lazy Narrowing," in *Proc. of the 5th Int'l Symp. on Programming Language Implementation and Logic Programming (PLILP'93)*, Springer LNCS 714, pp. 184–200, 1993.

48) López-Fraguas F. and Sánchez-Hernández J., "TOY: A Multiparadigm Declarative System," in *Proc. of the 10th Int'l Conf. on Rewriting Techniques and Applications (RTA'99)*, Springer LNCS 1631, pp. 244–247, 1999.

49) Lux W. and Kuchen H., "An Efficient Abstract Machine for Curry," in *Proc. of the 8th Int'l Workshop on Functional and Logic Programming (WFLP'99)*, pp. 171–181, 1999.

50) Martens B. and Gallagher J., "Ensuring Global Termination of Partial Deduction while Allowing Flexible Polyvariance," in *Proc. of the 12th Int'l Conf. on Logic Programming (ICLP'95)*, The MIT Press, pp. 597–611, 1995.

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

52) Nemytykh A., Pinchuk V., and Turchin V., "A Self-Applicable Supercompiler," in *Partial Evaluation, Proceedings*, Springer LNCS 1110, pp. 322–337, 1996.

53) Pettorossi A. and Proietti M., "A Comparative Revisitation of Some Program Transformation Techniques," in *Partial Evaluation, Proceedings*, Springer LNCS 1110, pp. 355–385, 1996.

54) Plasmeijer R. and Eekelen M., *Functional Programming and Parallel Graph Rewriting*, Addison Wesley, 1993.

55) Reddy U.S., "Narrowing as the Operational Semantics of Functional Languages," in *Proc. of the 2nd IEEE Int'l Symp. on Logic Programming*, IEEE, New York, pp. 138–151, 1985.

56) Slagle J., "Automated Theorem-Proving for Theories with Simplifiers, Commutativity and Associativity," *Journal of the ACM*, *21*, *4*, pp. 622–642, 1974.

57) Sørensen M., "Turchin's Supercompiler Revisited: An Operational Theory of Positive Information Propagation," Technical Report 94/7, Master's Thesis, DIKU, University of Copenhagen, Denmark, 1994.

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

59) Sørensen M., Glück R., and Jones N., "A Positive Supercompiler," *Journal of Functional Programming*, *6*, *6*, pp. 811–838, 1996.

60) Turchin V., "The Concept of a Supercompiler," *ACM Transactions on Programming Languages and Systems*, *8*, *3*, pp. 292–325, July 1986.

61) Vidal G., *Semantics-Based Analysis and Transformation of Functional Logic Programs*, Ph.D. thesis, DSIC, Universidad Politécnica de Valencia, Sept. 1996. In Spanish.