Improving Control in Functional Logic Program Specialization*

E. Albert[1], M. Alpuente[1], M. Falaschi[2], P. Julián[3], and G. Vidal[1]

[1] DSIC, U. Politécnica de Valencia, Camino de Vera s/n, 46022 Valencia, Spain.
{ealbert,alpuente,gvidal}@dsic.upv.es
[2] Dip. di Mat. e Informatica, U. Udine, Via delle Scienze 206, 33100 Udine, Italy.
falaschi@dimi.uniud.it
[3] Dep. de Informática, Ronda de Calatrava s/n, 13071 Ciudad Real, Spain.
pjulian@inf-cr.uclm.es

Abstract. We have recently defined a framework for Narrowing-driven Partial Evaluation (NPE) of functional logic programs. This method is as powerful as *partial deduction* of logic programs and *positive supercompilation* of functional programs. Although it is possible to treat complex terms containing primitive functions (e.g. conjunctions or equations) in the NPE framework, its basic control mechanisms do not allow for effective polygenetic specialization of these complex expressions. We introduce a sophisticated unfolding rule endowed with a dynamic narrowing strategy which permits flexible scheduling of the elements (in conjunctions) which are reduced during specialization. We also present a novel abstraction operator which extends some partitioning techniques defined in the framework of *conjunctive partial deduction*. We provide experimental results obtained from an implementation using the INDY system which demonstrate that the control refinements produce better specializations.

1 Introduction

Functional logic programming languages allow us to integrate some of the best features of the classical declarative paradigms, namely functional and logic programming. Lazy, efficient, functional computations are combined with the expressivity of logic variables, which allows for function inversion as well as logical search. The operational semantics of functional logic languages is usually based on (some form of) narrowing, which is a unification-based, parameter-passing mechanism which extends functional evaluation through goal solving capabilities as in logic programming (see [14] for a survey). In order to avoid unnecessary computations and to compute with infinite data structures, most recent work has concentrated on *lazy narrowing strategies* [11, 15, 23, 25].

The aim of *partial evaluation* (PE) is to specialize a given program w.r.t. part of its input data (hence, also called *program specialization*). PE techniques

* This work has been partially supported by CICYT TIC 95-0433-C03-03, by HCM project CONSOLE, and by Acción Integrada HA1997-0073.

have been widely applied to the optimization of functional (see [16] and references therein) and logic programs [10, 18, 22, 27]. Unfortunately, these techniques generally cannot be easily transferred to functional logic languages, since logical variables in function calls place specific demands that have to be tackled in order to achieve effective specialization.

Narrowing-driven PE [4] (NPE) provides a general scheme for the specialization of functional logic languages. The method is formalized within the theoretical framework established in [22, 24] for the PE of logic programs (also known as *partial deduction*, PD). However, a number of concepts have been generalized for dealing with features such as nested function calls, eager and lazy evaluation strategies and the standard optimization based on deterministically reducing functions. Control issues are managed by using standard techniques as in [24, 28]. The NPE method of [4] distinguishes a *local* and a *global* levels of control. At the local level, (finite) narrowing trees for (nested) function calls are constructed. At the global level, the calls extracted from the leaves of the local trees are considered for the next iteration of the algorithm, after a proper abstraction (generalization) that guarantees that only a finite number of calls is specialized. A close, automatic approach is that of positive supercompilation (PS) [29], whose basic transformation operation is *driving*, a unification-based transformation mechanism which is similar to (lazy) narrowing. A different PE method for a rewriting-based, functional logic language is considered in [19].

Classical PD computes partial evaluations for separate atoms independently. Recently, [12, 21] have introduced a technique for the partial deduction of conjunctions of atoms. This technique achieves a number of program optimizations such as (some form of) tupling and deforestation which are usually obtained through more expensive fold/unfold transformations, which are difficult to automate and which cannot be obtained through classical PD.

The NPE method of [4] is able to produce *polygenetic* specializations [13], i.e. it is able to extract specialized definitions which combine several function definitions of the original program. That means that NPE has the same potential for specialization as conjunctive PD or PS within the considered paradigm (a detailed comparison can be found in [5, 6]). This is because the generic method of [4] may allow one to deal with equations and conjunctions during specialization by simply considering the equality and conjunction operators as *primitive* function symbols of the language. Unfortunately, the use of primitive functions may encumber the nature of the specialization problems and it often turns out that some form of *tupling* (as defined in [27] for logic programs) is required for specializing expressions which contain conjunctive calls. The NPE algorithm of [4] does not incorporate a specific treatment for such primitive symbols, which depletes many opportunities for reaching the *closedness* condition and forces the method to dramatically generalize calls, thus giving up the intended specialization (see Example 1). Inspired by the challenging results of conjunctive PD in [12], this paper extends [4, 3] by formulating and experimentally testing concrete NPE control options that effectively handle primitive function symbols in lazy functional logic languages.

Some of the original contributions of our paper are as follows: i) we introduce a well-balanced dynamic unfolding rule and a novel abstraction operator that do not depend on the narrowing strategy and which highly improve the specialization of the NPE method; ii) these options allow us to tune the specialization algorithm to handle conjunctions (and other expressions containing primitive functions, such as conditionals and strict equalities) in a natural way, which provides for polygenetic specialization without any ad-hoc artifice; and iii) our method is applicable to modern functional logic languages with a lazy narrowing semantics such as Babel [25], Curry [15] and Toy [8], thus giving a specialization method which subsumes both lazy functional and conventional logic program specialization. We demonstrate the quality of these improvements by specializing some examples which were not handled well by classical NPE. The control strategies have been tested in the NPE system INDY [2].

The structure of the paper is as follows. Section 2 contains basic definitions. Section 3 extends the NPE algorithm of [3] to care for the appropriate handling of primitive function symbols. The algorithm is still generic in that no concrete control options are settled (i.e., there is no commitment to any concrete unfolding rule or abstraction operator). In Sect. 4, the concrete control options are described by formalizing some appropriate unfolding and abstraction operators. We illustrate the usefulness of our approach through some simple examples. Preliminary performance results, given in Sect. 5, show the practical importance of the proposed strategies. Finally, Sect. 6 concludes the paper. An extended version of this paper containing more details and proofs can be found in [1].

2 Preliminaries

We briefly summarize some well-known results about rewrite systems and functional logic programs [9, 14]. The definitions below are given in the homogeneous case. The extension to many-sorted signatures is straightforward [26].

Throughout this paper, $\mathcal{X}$ denotes a countably infinite set of *variables* and $\mathcal{F}$ denotes a set of *function symbols* (also called the *signature*), each of which has a fixed associated arity. We assume that the signature $\mathcal{F}$ is partitioned into two sets $\mathcal{F} = \mathcal{C} \cup \mathcal{D}$ with $\mathcal{C} \cap \mathcal{D} = \emptyset$. Symbols in $\mathcal{C}$ are called *constructors* and symbols in $\mathcal{D}$ are called *defined functions*. $\mathcal{T}(\mathcal{F}, \mathcal{X})$ denotes the set of *terms* or *expressions* built from $\mathcal{F}$ and $\mathcal{X}$. $\mathcal{T}(\mathcal{F})$ denotes the set of *ground terms*, while $\mathcal{T}(\mathcal{C}, \mathcal{X})$ denotes the set of *constructor* terms. If $t \notin \mathcal{X}$, then $\mathcal{H}ead(t)$ is the function symbol heading term t, also called the *root symbol* of t. A *pattern* is a term of the form $f(d_1, \ldots, d_n)$ where $f/n \in \mathcal{D}$ and $d_1, \ldots, d_n$ are constructor terms. $\mathcal{V}ar(s)$ is the set of variables occurring in the syntactic object s.

A *substitution* is a mapping from $\mathcal{X}$ to $\mathcal{T}(\mathcal{F}, \mathcal{X})$ s.t. its *domain* $\mathcal{D}om(\sigma) = \{x \in \mathcal{X} \mid x\sigma \not\equiv x\}$ is finite. We frequently identify a substitution σ with the set $\{x \mapsto x\sigma \mid x \in \mathcal{D}om(\sigma)\}$. We denote the identity substitution by *id*. We consider the usual preorder on substitutions $\leq$: θ is *more general* than σ (in symbols $\theta \leq \sigma$) iff $\exists\gamma.\ \sigma \equiv \theta\gamma$. The *restriction* $\sigma_{|V}$ of a substitution σ to a set

V of variables is defined by $\sigma_{|V} = x\sigma$ if $x \in V$ and $\sigma_{|V} = x$ if $x \notin V$. We write $\sigma =_V \theta$ iff $\sigma_{|V} = \theta_{|V}$, and $\sigma \leq_V \theta$ iff $\exists\gamma.\ \sigma\gamma =_V \theta$.

A term t is *more general* than s (or s is an *instance* of t), in symbols $t \leq s$, if $\exists\sigma.\ t\sigma \equiv s$. A *unifier* of a pair of terms $\{t_1, t_2\}$ is a substitution σ such that $t_1\sigma \equiv t_2\sigma$. A unifier σ is called *most general unifier* (*mgu*) if $\sigma \leq \sigma'$ for every other unifier σ'. A *generalization* of a set of terms $\{t_1, \ldots, t_n\}$ is a pair $\langle t, \{\theta_1, \ldots, \theta_n\}\rangle$ such that $t\theta_i = t_i$, $i = 1, \ldots, n$. A generalization $\langle t, \Theta\rangle$ is the *most specific generalization* (*msg*) if $t' \leq t$ for every other generalization $\langle t', \Theta'\rangle$.

Positions of a term t are represented by sequences of natural numbers used to address subterms of t. They are ordered by the prefix ordering: $p \leq q$ if there is w such that $p.w = q$, where $p.w$ denotes the concatenation of sequences p and w. We let Λ denote the empty sequence. $\mathcal{P}os(t)$ and $\mathcal{FP}os(t)$ denote, respectively, the set of positions and the set of nonvariable positions of the term t. $t|_p$ is the subterm of t at position p. $t[s]_p$ is the term t with the subterm at position p replaced with s.

We find it useful to simplify our description by limiting the discussion to unconditional term rewriting systems. A *rewrite rule* is pair $l \to r$ with $l, r \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, $l \notin \mathcal{X}$, and $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$. l and r are called the *left-hand side* (lhs) and *right-hand side* (rhs) of the rewrite rule, respectively. A *term rewriting system* (TRS) $\mathcal{R}$ is a finite set of rewrite rules. A *rewrite step* is an application of a rewrite rule to a term, i.e. $t \to_{p,l\to r} s$ if there exists a position $p \in \mathcal{P}os(t)$, a rewrite rule $l \to r$, and a substitution σ with $t|_p = l\sigma$ and $s = t[r\sigma]_p$. We say that $t|_p$ is a *redex* (*red*ucible *ex*pression) of t. A term t is *reducible* to term s if $t \to^* s$. A term t is *irreducible* or in *normal form* if there is no term s with $t \to s$. A TRS is *terminating* if there are no infinite sequences of the form $t_1 \to t_2 \to \ldots$ A TRS is called *confluent* if, whenever a term s reduces to two terms t_1 and t_2, both t_1 and t_2 reduce to the same term. Since we do not require terminating rules, normal forms may not exist.

Functional Logic Programming

The operational semantics of functional logic programs is usually based on (some variant) of *narrowing*. Essentially, narrowing consists of computing an appropriate substitution so that when applied to the current term, it becomes reducible, and then reducing it [14]. In this section, we briefly introduce a functional logic language whose syntax and demand-driven reduction mechanism is essentially equivalent to that of (a subset of) Babel [23, 25], Toy [8], and Curry [15], which has been recently proposed to become a standard in the area.

A TRS $\mathcal{R}$ is *constructor-based* (CB) if for each rule $l \to r \in \mathcal{R}$ the lhs l is a pattern. A CB TRS $\mathcal{R}$ is *weakly-orthogonal* if $\mathcal{R}$ is left-linear (i.e., for each rule $l \to r \in \mathcal{R}$, the lhs l does not contain multiple occurrences of the same variable) and $\mathcal{R}$ contains only trivial overlaps (i.e., if $l \to r$ and $l' \to r'$ are variants of distinct rules in $\mathcal{R}$ and σ is a unifier for l and l', then $r\sigma \equiv r'\sigma$). It is well-known that weakly-orthogonal TRS's are confluent. We henceforth consider CB weakly-orthogonal TRS's as programs. For this class of programs, a term t is a *head normal form* if t is a variable or $\mathcal{H}ead(t) \in \mathcal{C}$.

The signature $\mathcal{F}$ is augmented with a set of primitive function symbols $\mathcal{P} = \{\approx, \wedge, \Rightarrow\}$ in order to handle complex expressions containing equations $s \approx t$, conjunctions $b_1 \wedge b_2$, and conditional (guarded) terms $b \Rightarrow t$, i.e. $\mathcal{F} = \mathcal{C} \cup \mathcal{D} \cup \mathcal{P}$. We usually treat the symbols in $\mathcal{P}$ as infix operators. We assume that the following *predefined rules* belong to any given program:

$$c \approx c \;\to\; true \qquad\qquad \%\; c/0 \in \mathcal{C}$$
$$c(x_1, \ldots, x_n) \approx c(y_1, \ldots, y_n) \;\to\; (x_1 \approx y_1) \wedge \ldots \wedge (x_n \approx y_n) \qquad \%\; c/n \in \mathcal{C}$$
$$true \wedge x \;\to\; x \qquad x \wedge true \;\to\; x \qquad (true \Rightarrow x) \;\to\; x$$

These rules are weakly-orthogonal and define the validity of an equation as a *strict equality* between terms, which is common in functional languages when computations may not terminate [11, 25]. Note that, although the basic computation model only supports unconditional rules, it is still adequate to support logic programs since conditional rewrite rules $l \to r \Leftarrow C$ can be encompassed by guarded unconditional rules $l \to (C \Rightarrow r)$ by using the conditional primitive '$\Rightarrow$' as in Babel [25]. For reasons of simplicity, we assume the associativity of '$\wedge$' and assume that '$\approx$' binds more than '$\wedge$' and '$\wedge$' binds more than '$\Rightarrow$'.

We consider that programs are executed by *lazy narrowing*, which allows us to deal with nonterminating functions [23, 25]. Roughly speaking, laziness means that a given expression is only narrowed at inner positions if they are *demanded* (by the pattern in the lhs of some rule) and this contributes to a later narrowing step at an outer position. Formally, given a program $\mathcal{R}$, we define the *one-step narrowing* relation as follows. A term s narrows to t in $\mathcal{R}$, in symbols $s \leadsto_{p,l\to r,\sigma} t$ (or simply $s \leadsto_\sigma t$), iff there exists a position $p \in \varphi(s)$, a (standardized apart) rule $l \to r \in \mathcal{R}$, and a substitution σ such that $\sigma = mgu(\{s|_p, l\})$ and $t = (s[r]_p)\sigma$. The *selection strategy* $\varphi(t)$ is responsible for computing the set of *demanded* positions of a given term t. A formal definition of this strategy in terms of an inference system is shown in [1]. Lazy narrowing is *strong complete* w.r.t. constructor substitutions in CB, weakly-orthogonal TRS's [25, 14]. This means that the interpreter is free to disregard from $\varphi(t)$ all components of each conjunction which may occur in t except one, even if all arguments are demanded by the predefined rules of '$\wedge$' (that is, completeness holds for all scheduling policies). A formal definition can be found in [3].

If $s_0 \leadsto_{\sigma_1} s_1 \leadsto_{\sigma_2} \ldots \leadsto_{\sigma_n} s_n$ (in symbols, $s_0 \leadsto^*_\sigma s_n$, $\sigma = \sigma_1\sigma_2 \ldots \sigma_n$), we speak of a lazy narrowing *derivation* for the *goal* s_0 with (partial) *result* s_n. A lazy narrowing derivation $s \leadsto^*_\sigma t$ is *successful* iff $t \in \mathcal{T}(\mathcal{C} \cup \mathcal{X})$, where $\sigma_{|\mathcal{V}ar(s)}$ is the *computed answer substitution*.

3 The Generalized Specialization Algorithm

In this section, we generalize some basic concepts and techniques for the NPE of (lazy) functional logic programs (as presented in [3]). These extended notions will prove to be extremely useful for formulating new unfolding and abstraction operators which are well-suited to cope with primitive function symbols. In the original NPE framework, no distinction is made between primitive and defined function symbols during specialization. For instance, a conjunction $b_1 \wedge b_2$ is

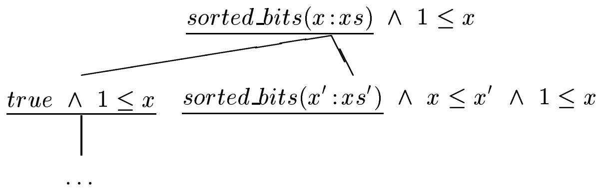

Fig. 1. Incomplete narrowing tree for $sorted\_bits(x:xs) \,\wedge\, 1 \leq x$

considered as a single entity when checking whether it is covered by the set of specialized calls. This commonly implies a drastic generalization of the involved calls, which causes losing all specialization, as the following example illustrates.

Example 1. Let us consider the program excerpt:

$sorted\_bits(x:[\,]) \;\rightarrow\; true$
$sorted\_bits(x_1:x_2:xs) \;\rightarrow\; sorted\_bits(x_2:xs) \;\wedge\; x_1 \leq x_2$
$0 \leq 0 \;\rightarrow\; true \qquad 0 \leq 1 \;\rightarrow\; true \qquad 1 \leq 1 \;\rightarrow\; true$

and the call $sorted\_bits(x : xs) \;\wedge\; 1 \leq x$. The lazy narrowing tree depicted[1] in Fig. 1 is built up by using the *nonembedding* unfolding rule of [3], which expands derivations while new redexes are not "greater" (with the *homeomorphic embedding ordering*, see e.g. [4, 28]) than previous, *comparable* redexes in the branch (i.e., redexes with the same outermost function symbol). From this tree, we can identify two main weaknesses of the plain NPE algorithm:

- The rightmost branch stops because the leftmost redex $sorted\_bits(x':xs')$ "embeds" the previous redex $sorted\_bits(x:xs)$, even if no reductions have been performed on the other elements of the conjunction.
- At the global level, since the call $sorted\_bits(x':xs') \wedge x \leq x' \wedge 1 \leq x$ in the leaf of the tree embeds (but does not cover) the specialized call $sorted\_bits(x:xs) \;\wedge\; 1 \leq x$ (and they are comparable), the msg $sorted\_bits(x : xs) \;\wedge\; z$ is computed, which gives up the intended specialization.

The first drawback pointed out in this example motivates the definition of more sophisticated unfolding rules which are able to achieve a *balanced* evaluation of the given expression by narrowing appropriate redexes (e.g., by using some kind of dynamic scheduling strategy which takes into account the ancestors narrowed in the same branch). The second drawback suggests the definition of a more flexible abstraction operator which is able to automatically split complex terms before attempting folding or generalization. In the following, we refine the framework of [3] in order to overcome these problems.

The PE of a term is obtained by constructing a (partial) narrowing tree and then extracting the resultants associated to the root-to-leaf paths of the tree.

Definition 1 (resultant). *Let s be a term and $\mathcal{R}$ be a program. Given a lazy narrowing derivation $s \leadsto^*_\sigma t$, its associated* resultant *is the rewrite rule $s\sigma \rightarrow t$.*

[1] We assume a fixed left-to-right selection of components within conjunctions and underline the selected redex at each step.

Definition 2 (partial evaluation). *Let $\mathcal{R}$ be a program and s be a term. Let τ be a finite (possibly incomplete) narrowing tree for s in $\mathcal{R}$ such that no goal in the tree is narrowed beyond its head normal form. Let $\{t_1, \ldots, t_n\}$ be the terms in the leaves of τ. Then, the set of resultants for the narrowing sequences $\{s \leadsto^{+}_{\sigma_i} t_i \mid i = 1, \ldots, n\}$ is called a* partial evaluation *of s in $\mathcal{R}$.*

The partial evaluation of a set of terms S in $\mathcal{R}$ is defined as the union of the partial evaluations for the terms in S.

Intuitively, the reason for requiring that the PE of a term s do not surpass its head normal form is that, at runtime, the evaluation of a (nested) call $C[s]_p$ containing the partially evaluated term s at some position p might not *demand* evaluating s beyond its head normal form. Since the "contexts" $C[\ ]$ in which s will appear are not known at PE time, we avoid to interfering with the "lazy nature" of computations in the specialized program by imposing this condition.

A recursive *closedness* condition is formalized by inductively checking that the different calls in the rules are sufficiently covered by the specialized functions.

Definition 3 (closedness). *Let S be a finite set of terms. A term t is S-closed if $closed(S, t)$ holds, where the predicate closed is defined inductively as follows:*

$$closed(S,t) \;\Leftrightarrow\; \begin{cases} true & \text{if } t \in \mathcal{X} \\ closed(S,t_1) \wedge \ldots \wedge closed(S,t_n) & \text{if } t \equiv c(t_1,\ldots,t_n) \\ \exists s \in S^{+}.\; s\theta = t \wedge \bigwedge_{x/t' \in \theta} closed(S,t') & \text{if } t \equiv f(t_1,\ldots,t_n) \end{cases}$$

where $c \in \mathcal{C}$, $f \in (\mathcal{D} \cup \mathcal{P})$, and $S^{+} = S \cup \{p(x,y) \mid p \in \mathcal{P}\}$. 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 program $\mathcal{R}$ is S-closed if $closed(S, \mathcal{R}_{calls})$ holds. Here we denote by $\mathcal{R}_{calls}$ the set of terms in the rhs's of the rules in $\mathcal{R}$.

The novelty w.r.t. [4, 3] is that a complex expression headed by a primitive function symbol, such as a conjunction, is proved closed w.r.t. S either by checking that it is an instance of a call in S (followed by an inductive test of the subterms), or by splitting it into two conjuncts and then trying to match with "simpler" terms in S (which happens when matching is first attempted w.r.t. one of the 'flat' calls $p(x, y)$ in S^{+}). This extension is safe since the rules which define the primitive functions are automatically added to each program.

The way in which a concrete PE is made is given by an *unfolding rule* (which decides how to stop the construction of lazy narrowing trees) and an *abstraction operator* (which ensures the finiteness of the set of specialized calls).

Definition 4 (unfolding rule [3]). *An* unfolding rule *U is a mapping which, when given a program $\mathcal{R}$ and a term s, returns a concrete PE for s in $\mathcal{R}$ (a set of resultants). By $U(S, \mathcal{R})$ we denote the union of $U(s, \mathcal{R})$ for all $s \in S$.*

Definition 5 (abstraction operator). *Given a finite set of terms T and a set of terms S, an* abstraction operator *is a function which returns a finite set of terms $abstract(S, T)$ such that: i) if $s \in abstract(S, T)$, then there exists $t \in (S \cup T)$ such that $t|_p = s\theta$ for some position p and substitution θ; ii) for all $t \in (S \cup T)$, t is closed w.r.t. the set of terms in $abstract(S, T)$.*

Roughly speaking, the first condition guarantees that the abstraction operator does not introduce new function symbols, while the second condition ensures that the resulting set of terms "covers" the calls previously specialized and that closedness is preserved throughout successive abstractions.

The following basic algorithm for NPE is parameterized by the unfolding rule U and the abstraction operator *abstract* in the style of [10].

Algorithm 1.
Input: a program $\mathcal{R}$ and a set of terms T
Output: a set of terms S
Initialization: $i := 0;\ T_0 := T$
Repeat
1. $\mathcal{R}' := U(T_i, \mathcal{R})$;
2. $T_{i+1} := abstract(T_i, \mathcal{R}'_{calls})$;
3. $i := i + 1$;

Until $T_i = T_{i-1}$ (modulo renaming)
Return $S := T_i$

The output of the NPE algorithm, given a program $\mathcal{R}$, is not a PE, but a set of terms S from which the partial evaluations $U(S, \mathcal{R})$ are automatically derived. Note that, whenever the specialized call is not a pattern, lhs's of resultants are not patterns either and hence resultants are not (CB) program rules. In [3], we introduced a post-processing renaming which is useful for producing CB rules. Roughly speaking, we construct an "independent renaming" S' of S as follows: for each term s in S, we define its independent renaming $s' = f_s(x_1, \ldots, x_n)$, where $x_1, \ldots, x_n$ are the distinct variables in s in the order of their first occurrence and f_s is a new fresh function symbol. Then, we fold each call t in the rules of $U(S, \mathcal{R})$ by replacing the old call t by a call to the corresponding term t' in S' (details can be found in [3]). After the algorithm terminates, the specialized program is obtained by applying this post-processing renaming to $U(S, \mathcal{R})$.

The (partial) correctness of the NPE algorithm is stated as follows.

Theorem 2. *Given a program $\mathcal{R}$ and a term t, if Algorithm 1 terminates by computing the set of terms S, then $\mathcal{R}'$ and t are S-closed, where $\mathcal{R}' = U(S, \mathcal{R})$.*

The correctness of the generic algorithm is stated in the following theorem, which generalizes Theorem 4.5 of [3].

Theorem 3. *Let $\mathcal{R}$ be a program, t a term, and S a finite set of terms. Let $\mathcal{R}'$ be a PE of $\mathcal{R}$ w.r.t. S such that $\mathcal{R}'$ and t are S-closed. Let S' be an independent renaming of S, and t'' (resp. $\mathcal{R}''$) be a renaming of t (resp. $\mathcal{R}'$) w.r.t. S'. Then t computes in $\mathcal{R}$ the result d with computed answer θ iff t'' computes in $\mathcal{R}''$ the result d with computed answer θ' and $\theta' \leq_{\mathcal{V}ar(t)} \theta$.*

4 Improving Control of NPE

In Sect. 4.1 we improve control in functional logic specialization by fixing an unfolding strategy which is specifically designed for "conjunctive specialization".

$$\underline{app(x,y)} \approx w \ \wedge \ app(w,z) \approx r$$

$$\vdots \qquad \underline{x' : app(xs',y) \approx w} \ \wedge \ app(w,z) \approx r$$

$$\underline{x' \approx w'} \ \wedge \ app(xs',y) \approx ws' \ \wedge \ app(w' : ws', z) \approx r$$

$$\underline{true \ \wedge \ app(xs',y) \approx ws' \ \wedge \ app(w' : ws', z) \approx r}$$

$$\underline{app(xs',y)} \approx ws' \ \wedge \ app(w' : ws', z) \approx r$$

Fig. 2. Naïve local control for $app(x,y) \approx w \ \wedge \ app(w,z) \approx r$

As for global control, a specific treatment of the primitive function symbols '$\approx$', '$\wedge$' and '$\Rightarrow$' is introduced in Sect. 4.2 which produces more effective and powerful, polygenetic specializations, as compared to classical NPE.

4.1 Local Control

The unfolding rule of [3] simply exploits the redexes selected by the lazy narrowing strategy φ (using a static selection rule which determines the next conjunct to be reduced) whenever none of them embed a previous (comparable) redex. The following example reveals that this strategy is not elaborated enough for specializing calls which may contain primitive symbols like conjunctions.

Example 2. Consider the well-known program *append*:

$$app([\,], y) \rightarrow y$$
$$app(x : xs, y) \rightarrow x : app(xs, y)$$

with the input goal "$app(x,y) \approx w \wedge app(w,z) \approx r$". Using the nonembedding unfolding rule of [3], we obtain the tree depicted[2] in Fig. 2 (using a fixed left-to-right selection rule for conjunctions). From this local tree, no appropriate specialized definition for the initial goal can be obtained, since the leaf cannot be folded into the input call in the root of the tree and generalization is required (which causes losing all specialization, as in Example 1).

We note that the NPE method [4, 3] succeeds with this example when the specialized call is written as a nested expression $app(app(x,y),z)$. This is because exploiting the nesting capability of the functional syntax allows us to transform the original tupling[3] problem illustrated by Example 2 into a simpler, deforestation problem, which is easily solved by the original NPE method.

[2] We adopt the standard optimization which makes use of the built-in unification to solve strict equalities $s \approx t$ (provided that only constructor bindings are produced).

[3] Here we refer to tupling of logic programs, which subsumes both deforestation and tupling of functional programs [27].

Now we introduce a dynamic lazy unfolding rule which attempts to achieve a fair evaluation of the complete input term, rather than a deeper evaluation of some given subterm. This novel concrete unfolding rule dynamically selects the positions to be reduced within a given conjunction, by exploiting some dependency information between redexes gathered along the derivation.

Definition 6 (dependent positions). *Let $D \equiv (s \leadsto_{p,l\to r,\sigma} t)$ be a narrowing step. The set of* dependent positions *of a position q of s by D, written $q\backslash\backslash D$, is:*

$$q\backslash\backslash D = \begin{cases} \{q.u \mid u \in \mathcal{FP}os(r) \wedge \mathcal{H}ead(r|_u) \notin \mathcal{C}\} & \text{if } q = p \\ \{q\} & \text{if } q \not\leq p \\ \{p.u'.v \mid r|_{u'} = x\} & \text{if } q = p.u.v \text{ and } l|_u = x \in \mathcal{X} \end{cases}$$

This notion can be naturally lifted to narrowing derivations.

The notion of dependency for terms stems directly from the corresponding notion for positions. Note that the above definition is a strict extension of the standard notion of *descendant* in functional programming. Intuitively, the descendants of the subterm $s|_q$ are computed as follows: underline the root symbol of $s|_q$ and perform the narrowing step $s \leadsto t$. Then, every subterm of t with and underlined root symbol is a descendant of $s|_q$ [17]. Intuitively, a position q' of t depends on a position q of s (by D) if q' is a descendant of q (second and third cases), or if the position q' has been introduced by the rhs of the rule applied in the reduction of the former position q and it addresses a subterm headed by a defined function symbol (first case). Note that this notion is an extension of the standard PD concept of (covering) ancestor to the functional logic framework. By abuse, we also say that the term addressed by q is an *ancestor* of the term addressed by q' in D. If s is an ancestor of t and $\mathcal{H}ead(s) = \mathcal{H}ead(t)$, we say that s is a *comparable ancestor* of t in D.

Now we formalize the way in which the dynamic selection is performed.

Definition 7 (dynamic narrowing selection strategy).
Let $D \equiv (t_0 \leadsto t_1 \leadsto \ldots \leadsto t_n)$, $n \geq 0$ be a lazy narrowing derivation. We define the dynamic selection rule $\varphi_{dynamic}$ as: $\varphi_{dynamic}(t_n, D) = select(t_n, \Lambda, D)$, where the auxiliary function select is:

$$\begin{array}{l} select(t,p,D) = \textit{if } p \in \varphi(t) \\ \qquad \textit{then if } dependency\_clash(t|_p, D) \textit{ then } \{\bot\} \textit{ else } \{p\} \\ \qquad \textit{else case } t|_p \textit{ of} \\ \qquad\quad \begin{array}{ll} x \in \mathcal{V}\textit{:} & \emptyset \\ s_1 \wedge s_2\textit{:} & \textit{let } O_i = select(t, p.i, D),\ i \in \{1,2\}, \textit{ in} \\ & [\textit{if } \exists i.\ (\bot \notin O_i \wedge O_i \not\equiv \emptyset) \textit{ then } O_i \\ & \textit{else if } (O_1 \equiv O_2 \equiv \emptyset) \textit{ then } \emptyset \textit{ else } \{\bot\}] \\ \textit{otherwise:} & \textit{let } t|_p = f(s_1, \ldots, s_n) \textit{ and} \\ & O_{args} = \bigcup_{i=1}^{n} select(t, p.i, D) \textit{ in} \\ & [\textit{if } \bot \in O_{args} \textit{ then } \{\bot\} \textit{ else } O_{args}] \end{array} \end{array}$$

where $dependency\_clash(t, D)$ is a generic Boolean function that looks at the ancestors of t in D to determine whether there is a risk of nontermination.

$$\underline{app(x,y)} \approx w \ \wedge \ app(w,z) \approx r$$

$$\vdots \qquad \underline{x' : app(xs',y) \approx w} \ \wedge \ app(w,z) \approx r$$

$$x' \approx w' \ \wedge \ app(xs',y) \approx ws' \ \wedge \ \underline{app(w' : ws',z)} \approx r$$

$$x' \approx w' \ \wedge \ app(xs',y) \approx ws' \ \wedge \ \underline{w' : app(ws',z) \approx r}$$

$$x' \approx w' \ \wedge \ app(xs',y) \approx ws' \ \wedge \ w' \approx r' \ \wedge \ app(ws',z) \approx rs'$$

Fig. 3. Improved local control for $app(x,y) \approx w \wedge app(w,z) \approx r$

For the sake of simplicity, in the remainder of this section we consider that $dependency\_clash(t,D)$ holds whenever there is a comparable ancestor of the selected redex t in D. Another approach, that we investigate in the experiments, is to additionally test homeomorphic embedding on comparable ancestors.

Roughly speaking, the dynamic selection strategy recurs over the structure of the goal and determines the set of positions to be unfolded by a don't-care selection within each conjunction of exactly one of the components (among those that do not incur into a $dependency\_clash$). We introduce a dynamic unfolding rule $U_{dynamic}(t,\mathcal{R})$ which simply expands lazy narrowing trees according to the dynamic lazy narrowing strategy $\varphi_{dynamic}$. The "mark" $\perp$ of Definition 7 is used as a whistle to warn us that the derivation must be cut off since it runs into a $dependency\_clash$, i.e., each branch D of the tree is stopped whenever $\varphi_{dynamic}(t,D) = \{\perp\}$ or the term t (of the leaf) is in head normal form.

Example 3. Consider again the program and goal of Example 2. Using the dynamic unfolding rule $U_{dynamic}$, we get the tree depicted in Fig. 3. From this tree, an optimal (recursive) specialized definition for the initial call can be derived, provided there is a suitable splitting mechanism to extract, from the leaf of the tree, an appropriate subconjunction such as $app(xs',y) \approx ws' \wedge app(ws',z) \approx rs'$, which is covered by the initial call in the root of the tree (see Example 4).

4.2 Global Control

In the presence of primitive functions like '$\wedge$' or '$\approx$', using an abstraction operator which respects the structure of the terms (as in [3]) is not very effective, since the generalization of two conjunctions (resp. equations) might be a term of the form $x \approx y \wedge z$ (resp. $x \approx y$) in most cases. The drastical solution of decomposing the term into subterms containing just one function call can avoid the problem, but has the negative consequence of losing nearly all specialization. In this section, we introduce a more concerned abstraction operator which is inspired by the partitioning techniques of conjunctive PD [12, 21], and which uses the homeomorphic embedding relation $\trianglelefteq$ as defined in [28].

The following notion of *best matching terms*, which is aimed at avoiding loss of specialization due to generalization, is a proper generalization of the notion of *best matching conjunction* in [12].

Definition 8 (best matching terms). *Let $S = \{s_1, \ldots, s_n\}$ be a set of terms, t a term, and consider the set of terms $W = \{w_i \mid \langle w_i, \{\theta_{i1}, \theta_{i2}\}\rangle = msg(\{s_i, t\}),$ $i = 1, \ldots, n\}$. The* best matching terms *$BMT(S, t)$ for t in S are those terms $s_j \in S$ such that the corresponding w_j in W is a minimally general element.*

The notion of BMT is used in the abstraction process at two stages: i) when selecting the more appropriate term in S which covers a new call t, and ii) when determining whether a call t headed by a primitive function symbol could be (safely) added to the current set of specialized calls or should be split.

Definition 9 (concrete abstraction operator). *Let S and T be sets of terms. We define $abstract_{\trianglelefteq}(S, T)$ inductively as follows: $abstract_{\trianglelefteq}(S, T)$ =*

$$\begin{cases} S & \text{if } T \equiv \emptyset \text{ or } T \equiv \{t\},\ t \in \mathcal{X} \\ abstract_{\trianglelefteq}(\ldots abstract_{\trianglelefteq}(S, t_1), \ldots, t_n) & \text{if } T \equiv \{t_1, \ldots, t_n\}, n > 0 \\ abstract_{\trianglelefteq}(S, \{t_1, \ldots, t_n\}) & \text{if } T \equiv \{t\},\ t \equiv c(t_1, \ldots, t_n),\ c \in \mathcal{C} \\ abs\_def(S, T', t) & \text{if } T \equiv \{t\},\ \mathcal{H}ead(t) \in \mathcal{D} \\ abs\_prim(S, T', t) & \text{if } T \equiv \{t\},\ \mathcal{H}ead(t) \in \mathcal{P} \end{cases}$$

where $T' = \{s \in S \mid \mathcal{H}ead(s) = \mathcal{H}ead(t) \wedge s \trianglelefteq t\}$. The functions abs_def and abs_prim are defined as follows:

$$\begin{array}{lll} abs\_def(S, \emptyset, t) & = abs\_prim(S, \emptyset, t) & = S \cup \{t\} \\ abs\_def(S, T, t) & = abstract_{\trianglelefteq}(S \setminus \{s\}, \{w\} \cup \mathcal{R}an(\theta_1) \cup \mathcal{R}an(\theta_2)) & \\ & \quad \text{if } \langle w, \{\theta_1, \theta_2\}\rangle = msg(\{s, t\}),\ \text{with } s \in BMT(T, t) & \end{array}$$

$$abs\_prim(S, T, t) = \begin{cases} abs\_def(S, T, t) & \text{if } \exists s \in BMT(T, t) \text{ s.t. } def(t) = def(s) \\ abstract_{\trianglelefteq}(S, T, \{t_1, t_2\}) & \text{otherwise, where } t \triangleq p(t_1, t_2) \end{cases}$$

where $def(t)$ denotes a sequence with the defined *function symbols of t in lexicographical order, and $\triangleq$ is equality up to reordering of elements in a conjunction.*

Essentially, the way in which the abstraction operator proceeds is simple. We distinguish the cases when the considered term i) is a variable, ii) is headed by a constructor symbol, iii) by a defined function symbol, or iv) by a primitive function symbol. The actions that the abstraction operator takes, respectively, are: i) to ignore it, ii) to recursively inspect the subterms, iii) to generalize the given term w.r.t. some of its best matching terms (recursively inspecting the msg w and the subterms of θ_1, θ_2 not covered by the generalization), and iv) the same as in iii), but considering the possibility of splitting the given expression before generalizing it when $def(t) \neq def(s)$ (which essentially states that some defined function symbols would be lost due to the application of msg). The function $abstract_{\trianglelefteq}$ is an abstraction operator in the sense of Definition 5 [1]. The following result establishes the termination of the global specialization process

Theorem 4. *Algorithm 1 terminates for the unfolding rule $U_{dynamic}$ and the abstraction operator $abstract_{\trianglelefteq}$.*

Our final example witnesses that $abstract_{\trianglelefteq}$ behaves well w.r.t. Example 3.

Example 4. Consider again the tree depicted in Fig. 3. By applying Algorithm 1, the following call to $abstract_{\trianglelefteq}$ is undertaken:

$$abstract_{\trianglelefteq}(\{app(x,y) \approx w \ \wedge \ app(w,z) \approx r\},$$
$$\{x' \approx w' \ \wedge \ app(xs',y) \approx ws' \ \wedge \ w' \approx r' \ \wedge \ app(ws',z) \approx rs'\}).$$

Following Definition 9, by two recursive calls to *abs_prim*, we get:

$$\{app(x,y) \approx w \ \wedge \ app(w,z) \approx r, \ x' \approx w'\}.$$

By considering the independent renaming $dapp(x,y,w,z,r)$ for the specialized call $app(x,y) \approx w \wedge app(w,z) \approx r$, the method derives a (recursive) rule of the form: $dapp(x\!:\!xs, y, w\!:\!ws, z, r\!:\!rs) \ \rightarrow \ x \approx w \ \wedge \ w \approx r \ \wedge \ dapp(xs,y,ws,z,rs)$, which embodies the intended optimal specialization for this example.

5 Experiments

The refinements presented so far have been incorporated into the NPE prototype implementation system Indy (Integrated Narrowing-Driven specialization system [2]). Indy is written in SICStus Prolog v3.6 and is publicly available [2].

In order to assess the practicality of our approach, we have benchmarked the speed and specialization achieved by the extended implementation. The benchmarks used for the analysis are: `applast`, which appends an element at the end of a given list and returns the last element of the resulting list; `double_app`, see Example 2; `double_flip`, which flips a tree structure twice, then returning the original tree back; `fibonacci`, fibonacci's function; `heads&legs`, which computes the number of heads and legs of a given number of birds and cats; `match-app`, the extremely naïve string pattern matcher based on using `append`; `match-kmp`, a semi-naïve string pattern matcher; `maxlength`, which returns the maximum and the length of a list; `palindrome`, a program to check whether a given list is a palindrome; and `sorted_bits`, see Example 1. Some of the examples are typical PD benchmarks (see [20]) adapted to a functional logic syntax, while others come from the literature of functional program transformations, such as PS [29], fold/unfold transformations [7], and deforestation [30].

We have considered the following settings to test the benchmarks:

- Evaluation strategy: All benchmarks were executed by lazy narrowing.
- Unfolding rule: We have tested three different alternatives: (1) `emb_goal`: it expands derivations while new goals do not embed a previous comparable goal in the same branch; (2) `emb_redex`: the concrete unfolding rule of Sect. 4.1 which implements the *dependency_clash* test using homeomorphic embedding on comparable ancestors of selected redexes to ensure finiteness (note that it differs from `emb_goal` in that `emb_redex` implements dynamic scheduling on conjunctions and that homeomorphic embedding is checked on simple redexes rather than on whole goals); (3) `comp_redex`: the unfolding rule of Sect. 4.1 which uses the simpler definition of *dependency_clash* based on comparable ancestors of selected redexes as a whistle.
- Abstraction operator: Abstraction is always done as explained in Def. 9.

Table 1. Benchmark results

	Original		emb_goal		emb_redex		comp_redex	
Benchmarks	Rw	RT	Rw	Speedup	Rw	Speedup	Rw	Speedup
applast	10	90	13	1.32	28	2.20	13	1.10
double_app	8	106	39	1.63	61	1.28	15	3.12
double_flip	8	62	26	1.51	17	1.55	17	1.55
fibonacci	5	119	11	1.19	7	1.08	7	1.08
heads&legs	8	176	24	4.63	22	2.41	21	2.48
match-app	8	201	12	1.25	20	2.75	23	2.79
match-kmp	12	120	14	3.43	14	3.64	13	3.43
maxlength	14	94	51	1.17	20	1.27	18	1.25
palindrome	10	119	19	1.25	10	1.35	10	1.35
sorted_bits	8	110	16	1.15	31	2.89	10	2.68
Average	9.1	119.7	22.5	1.85	23	2.04	14.7	2.08
TMix average			1881		7441		5788	

Table 1 summarizes our benchmark results. The first two columns measure the number of rewrite rules (Rw) and the absolute runtimes (RT) for each original program. The other columns show the number of rewrite rules and the speedups achieved for the specialized programs obtained by using the three considered unfolding rules. The row at the bottom of the table (TMix) indicates the average specialization time for each considered unfolding rule. Times are expressed in milliseconds and are the average of 10 executions. Speedups were computed by running the original and specialized programs under the publicly available lazy functional logic language Toy [8]. Runtime input goals were chosen to give a reasonably long overall time. The complete code for benchmarks, the specialized calls, and the partially evaluated programs can be found in [1].

The figures in Table 1 demonstrate that the control refinements that we have incorporated into the Indy system provide satisfactory speedups on all benchmarks (which is very encouraging, given the fact that no partial input data were provided in any example, except for `match-app`, `match-kmp`, and `sorted_bits`). On the other hand, our extensions are *conservative* in the sense that there is no penalty w.r.t. the specialization achieved by the original system on non-conjunctive goals (although some specialization times are slightly higher due to the more complex processing being done). Let us note that, from the speedup results in Table 1, it can appear that there is no significant difference between the strategies `emb_redex` and `comp_redex`. However, when we also consider the specialization times (TMix) and the size of the specialized programs (Rw), we find out that `comp_redex` has a better overall behaviour. A detailed comparison between the considered unfolding strategies can be found in [1].

6 Discussion

In functional logic languages, expressions can be written by exploiting the *nesting* capability of the functional syntax, as in $app(app(x, y), z) \approx r$, but in many cases it can be appropriate (or necessary) to decompose nested expressions as in logic programming, and write $app(x, y) \approx w \wedge app(w, z) \approx r$ (for instance, if some test such as $sorted\_bits(w)$ on the intermediate list w were necessary). The original INDY system behaves well on programs written with the "pure" functional syntax [5]. However, INDY is not able to produce good specialization on the benchmarks of Table 1 when they are written as conjunctions of subgoals. For this we could not achieve some of the standard, difficult transformations such as tupling [7, 27] within the classical NPE framework. As opposed to the classical PD framework (in which only folding on single atoms can be done), the NPE algorithm is able to perform folding on complex expressions (containing an arbitrary number of function calls). This does not suffice to achieve tupling in practice, since complex expressions are often generalized and specialization is lost. We have shown that the NPE general framework can be supplied with appropriate control options to specialize complex expressions containing primitive functions, thus providing a powerful polygenetic specialization framework with no ad-hoc setting.

To the best of our knowledge, this is the first practical framework for the specialization of modern functional logic languages with partitioning techniques and dynamic scheduling. As future research, there is room for further improvement in performance by introducing more powerful abstraction operators based on better analyses to determine the optimal way to split expressions (trying not to endanger the communication of data structures with shared variables), and by considering in practice the problem of controlling particular algebraic laws for primitive symbols.

References

1. E. Albert, M. Alpuente, M. Falaschi, P. Julián, and G. Vidal. Improving Control in Functional Logic Program Specialization. Technical Report DSIC-II/2/97, UPV, 1998. Available from URL: `http://www.dsic.upv.es/users/elp/papers.html`.
2. E. Albert, M. Alpuente, M. Falaschi, and G. Vidal. INDY User's Manual. Technical Report, available from `http://www.dsic.upv.es/users/elp/papers.html`.
3. M. Alpuente, M. Falaschi, P. Julián, and G. Vidal. Specialization of Lazy Functional Logic Programs. In *Proc. of PEPM'97*, volume 32(12) of *Sigplan Notices*, pages 151–162, New York, 1997. ACM Press.
4. M. Alpuente, M. Falaschi, and G. Vidal. Narrowing-driven Partial Evaluation of Functional Logic Programs. In H. Riis Nielson, editor, *Proc. of the 6th European Symp. on Programming, ESOP'96*, pages 45–61. Springer LNCS 1058, 1996.
5. M. Alpuente, M. Falaschi, and G. Vidal. Partial Evaluation of Functional Logic Programs. *ACM TOPLAS*, 1998. To appear.
6. M. Alpuente, M. Falaschi, and G. Vidal. A Unifying View of Functional and Logic Program Specialization. *ACM Computing Surveys*, 1998. To appear.
7. R.M. Burstall and J. Darlington. A Transformation System for Developing Recursive Programs. *Journal of the ACM*, 24(1):44–67, 1977.
8. R. Caballero-Roldán, F.J. López-Fraguas, and J. Sánchez-Hernández. User's manual for Toy. Technical Report SIP-5797, UCM, Madrid (Spain), April 1997.
9. 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.

10. J. Gallagher. Tutorial on Specialisation of Logic Programs. In *Proc. of PEPM'93*, pages 88–98. ACM, New York, 1993.
11. E. Giovannetti, G. Levi, C. Moiso, and C. Palamidessi. Kernel Leaf: A Logic plus Functional Language. *J. of Computer and System Sciences*, 42:363–377, 1991.
12. R. Glück, J. Jørgensen, B. Martens, and M.H. Sørensen. Controlling Conjunctive Partial Deduction of Definite Logic Programs. In *Proc. of PLILP'96*, pages 152–166. Springer LNCS 1140, 1996.
13. R. Glück and M.H. Sørensen. A Roadmap to Metacomputation by Supercompilation. In *Partial Evaluation, Int'l Seminar, Dagstuhl Castle, Germany*, pages 137–160. Springer LNCS 1110, February 1996.
14. M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, 19&20:583–628, 1994.
15. M. Hanus, H. Kuchen, and J.J. Moreno-Navarro. Curry: A Truly Functional Logic Language. In *Proc. ILPS'95 Workshop on Visions for the Future of Logic Programming*, pages 95–107, 1995.
16. N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Englewood Cliffs, NJ, 1993.
17. J.W. Klop and A. Middeldorp. Sequentiality in Orthogonal Term Rewriting Systems. *Journal of Symbolic Computation*, pages 161–195, 1991.
18. J. Komorowski. An Introduction to Partial Deduction. In A. Pettorossi, editor, *Meta-Programming in Logic*, pages 49–69. Springer LNCS 649, 1992.
19. L. Lafave and J.P. Gallagher. Partial Evaluation of Functional Logic Programs in Rewriting-based Languages. Technical Report CSTR-97-001, Department of Computer Science, University of Bristol, Bristol, England, March 1997.
20. M. Leuschel. The ECCE partial deduction system and the DPPD library of benchmarks. Tech. Rep., accessible via `http://www.cs.kuleuven.ac.be/~lpai`, 1998.
21. 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.
22. J.W. Lloyd and J.C. Shepherdson. Partial Evaluation in Logic Programming. *Journal of Logic Programming*, 11:217–242, 1991.
23. 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.
24. B. Martens and J. Gallagher. Ensuring Global Termination of Partial Deduction while Allowing Flexible Polyvariance. In L. Sterling, editor, *Proc. of ICLP'95*, pages 597–611. MIT Press, 1995.
25. J.J. Moreno-Navarro and M. Rodríguez-Artalejo. Logic programming with functions and predicates: the language Babel. *J. Logic Program.*, 12(3):191–224, 1992.
26. P. Padawitz. *Computing in Horn Clause Theories*, volume 16 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Berlin, 1988.
27. A. Pettorossi and M. Proietti. Rules and Strategies for Transforming Functional and Logic Programs. *ACM Computing Surveys*, 28(2):360–414, 1996.
28. M.H. Sørensen and R. Glück. An Algorithm of Generalization in Positive Supercompilation. In *Proc. of ILPS'95*, pages 465–479. The MIT Press, 1995.
29. M.H. Sørensen, R. Glück, and N.D. Jones. A Positive Supercompiler. *Journal of Functional Programming*, 6(6):811–838, 1996.
30. P.L. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.