Polygenetic Partial Evaluation of Lazy Functional Logic Programs

E. Albert M. Alpuente M. Falaschi P. Julián G. Vidal

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 carefully considers primitive functions and is the key to achieving accurate polygenetic specialization. It adapts and extends some recent partitioning techniques defined in the framework of *conjunctive partial deduction*. We provide experimental results obtained from an implementation using the INDY system.

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. Narrowing provides completeness in the sense of logic programming —computation of answers— as well as functional programming —computation of normal forms— (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* [15, 22, 24].

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 have been widely applied to the optimization of functional (see [9, 16] and references therein) and

E. Albert, M. Alpuente, G. Vidal: DSIC, UPV, Camino de Vera s/n, 46022 Valencia, Spain.
M. Falaschi: DIMI, Via delle Scienze 206, 33100 Udine, Italy.
P. Julián: Dep. de Informática, Ronda de Calatrava s/n, 13071 Ciudad Real, Spain.
This work has been partially supported by CICYT TIC 95-0433-C03-03, by HCM project CONSOLE and by Acción Integrada HA1997-0073.

logic programs [11, 18, 21, 26]. These methods do not transfer easily to functional logic languages, since logical variables in function calls place new technical demands.

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 [21, 23] 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 [23, 27]. 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) [28], whose basic transformation operation is *driving*, a unification-based transformation mechanism which is similar to (lazy) narrowing.

Classical PD computes partial evaluations for separate atoms independently. Recently, [12, 20] have introduced a technique for the PD 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.

The NPE method of [4] is able to produce *polygenetic* specializations, i.e. it is able to extract specialized definitions which combine several function definitions of the original program (see, e.g., [13]). 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 considering the equality and conjunction operators as *primitive* function symbols of the language. However, the use of primitive functions such as conjunctions may encumber the nature of the specialization problems and it often turns out that some form of *tupling* (as defined in [26] for logic programs) is required for specializing expressions which contain conjunctive calls.

At the local level of control, the reduction of the elements in a conjunction can be made don't care nondeterministically, and the order in which elements are chosen during the construction of the local narrowing trees is crucial to achieving good specialization. As we will see in Section 4, naïve selection may lose all specialization. Some kind of dynamic selection strategy which keeps track of the ancestors reduced in the same derivation is necessary to significantly speed-up execution. At the global level of control, enhancing the general NPE algorithm requires particular techniques for carefully splitting complex terms containing primitive symbols before checking whether they are covered by the set of (already) specialized functions, to avoid an unsuitable generalization of the calls to be partial evaluated that may result in monogenetic specializations in many interesting cases (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) in a natural way, which provides for polygenetic specialization without any ad-hoc artifice; and iii) our method is applicable

to modern lazy functional logic languages such as Babel [24], 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 prototype implementation INDY [2].

The structure of the paper is as follows. Section 2 contains basic definitions. Section 3 extends slightly the generic NPE algorithm of [3] to care for the appropriate handling of primitive function symbols. In Section 4, the concrete control options are described by formalizing some appropriate unfolding and abstraction operators. Preliminary performance results, given in Section 5, show the practical importance of the proposed strategies. Finally, Section 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 programming [10, 14, 17]. The definitions below are given in the homogeneous case. The extension to many-sorted signatures is straightforward [25].

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. The identity of syntactic objects is denoted by $\equiv$. $\mathcal{V}ar(s)$ is the set of variables occurring in the syntactic object s.

A *substitution* is a mapping from $\mathcal{X}$ to $\mathcal{T}(\mathcal{F}, \mathcal{X})$ such that 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, and $\sigma_{|V}$ denotes the *restriction* of a substitution σ to a set V of variables. We consider the usual preorder on substitutions $\leq$: θ is *more general* than σ (in symbols $\theta \leq \sigma$) iff $\exists\gamma.\ \sigma \equiv \theta\gamma$.

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 (possibly empty) of natural numbers used to address subterms of t, and they are ordered by the prefix ordering $p \leq q$, if there exists w such that $pw = q$. 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. $t \rightarrow_{p,l\rightarrow r} s$ if there exists a position $p \in \mathcal{P}os(t)$, a rewrite rule $l \rightarrow r$, 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 $t \rightarrow^* s$. A term t is *irreducible* or in *normal form* if there is no term s with $t \rightarrow s$. A TRS $\mathcal{R}$ is called terminating (confluent) if the induced rewrite relation $\rightarrow$ is terminating (confluent).

Functional Logic Programming

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 [22, 24], Toy [8], and Curry [15], a modern integrated language which has been recently proposed to become a standard in the area.

A TRS $\mathcal{R}$ is *constructor-based* (CB) if for each rule $l \rightarrow r \in \mathcal{R}$ 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 $l \rightarrow r \in \mathcal{R}$, the lhs l does not contain multiple occurrences of the same variable) and $\mathcal{R}$ contains only trivial overlaps (i.e., if $l \rightarrow r$ and $l' \rightarrow r'$ 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 assume that the following *predefined rules* belong to any given program:

$$
\begin{array}{lll}
c \approx c \;\rightarrow\; true & & \%\ c/0 \in \mathcal{C} \\
c(x_1, \ldots, x_n) \approx c(y_1, \ldots, y_n) \;\rightarrow\; (x_1 \approx y_1) \wedge \ldots \wedge (x_n \approx y_n) & & \%\ c/n \in \mathcal{C} \\
true \wedge x \;\rightarrow\; x \qquad x \wedge true \;\rightarrow\; x \qquad (true \Rightarrow x) \;\rightarrow\; x & &
\end{array}
$$

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 [15, 24]. A *solution* to an equation $s \approx t$ is a substitution σ such that $(s \approx t)\sigma$ rewrites to *true* using the rules of the program.

Note that it is still adequate to support logic programs since conditional rewrite rules $l \rightarrow r \Leftarrow C$ can be encompassed by guarded unconditional rules $l \rightarrow (C \Rightarrow r)$ [24]. 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 [22, 24]. 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 $s \leadsto_{p,l\rightarrow r,\sigma} t$ (or simply $s \leadsto_\sigma t$), iff there exists a position $p \in \varphi(s)$, a (standardized apart) rule $l \rightarrow r \in \mathcal{R}$, 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 is shown in [1]. Lazy narrowing is *strong complete* w.r.t. constructor substitutions in CB, weakly-orthogonal TRS's [24, 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]. This will be useful when defining the concrete unfolding rule of Section 4, where a fair evaluation of redexes within conjunctions is crucial to achieving an effective specialization.

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 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*.

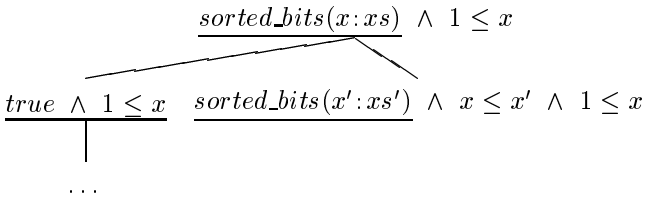
3 The Generalized Specialization Algorithm

In 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 considered as a block 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. The following example illustrates this point.

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 following lazy narrowing tree[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, 27]) 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 $sorted\_bits(x' : xs')$ of the leaf "embeds" the previous redex $sorted\_bits(x{:}xs)$, even if no reductions have been performed on the other elements of the conjunction, which does not seem very equitable.
- According to the NPE algorithm in [3], since the call $sorted\_bits(x'{:}xs') \wedge x \leq x' \wedge 1 \leq x$ in the leaf of the tree embeds (but is not covered by) 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. 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 slightly generalize some basic concepts and techniques for the NPE of (lazy) functional logic programs (as presented in [3]) in order to properly deal with primitive function symbols.

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

The PE of a term s is obtained by constructing a (partial) narrowing tree for s, and then extracting the specialized definitions —the resultants— from the root-to-leaf paths of the tree.

Definition 3.1 (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 $s\sigma \to t$.*

Definition 3.2 (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.*

Roughly speaking, the reason for requiring partial evaluations to not "surpass" head normal forms is that, at run time, the evaluation of an expression $C[t]_p$ containing a partially evaluated term t might not demand evaluating t beyond its head normal form. Since this is 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 3.3 (closedness) *Let S be a finite set of terms and t a term. We say that a term t is S-closed if $closed(S,t)$ holds, where the predicate closed is defined as follows:*

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

where $S^+ = S \cup \{p(x,y) \mid p \in \mathcal{P}\}$. A set of terms T is S-closed, written $closed(S,T)$, if $closed(S,t)$ holds for all $t \in T$, and 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 $\mathcal{R}$.

Informally, a term t headed by a defined function symbol is closed w.r.t. a set of calls S, if it is an instance of a term of S and the terms in the matching substitution are recursively closed by S. 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 easy extension of the closedness condition allows us to formulate refined abstraction operators in which terms containing primitive symbols are (possibly) partitioned (cf. Section 4.2) before attempting folding or generalization.

The way in which concrete partial evaluations are constructed is given by an *unfolding rule*, which determines the expressions to be narrowed (regarding the narrowing strategy φ) and which decides how to stop the construction of lazy narrowing trees.

Definition 3.4 (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 $s \in S$.*

The *abstraction operator* guarantees the termination of the NPE process by ensuring the finiteness of the set of terms for which partial evaluations are produced.

Definition 3.5 (abstraction operator) *Given a finite set of terms T and a set of terms S, an* abstraction operator *returns a finite set of terms $abstract(S,T)$ such that: i) if $s \in abstract(S,T)$, then there exists $t \in T$ 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 $abstract(S,T)$.*

Intuitively, the first condition guarantees that the abstraction operator does not introduce new function symbols not appearing in the input sets S and T, 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 [11].

Algorithm 3.6
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 $S := T_i$

Similarly to [23], by applying *abstract* at every iteration of Algorithm 3.6 we can tune the control of polyvariance (i.e. the ability to produce several specialized definitions for a single original function) as much as needed. The output of the algorithm, given a program $\mathcal{R}$, is not a partial evaluation, but a set of terms S from which the partial evaluations $U(S,\mathcal{R})$ are automatically derived, as is usual. Note that, if 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 transformation which is useful for producing CB rules and guarantees the completeness of the transformation. Informally, for each term s in S, we define the "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 the f_s's are new fresh function symbols. Then, we fold each call t in the resultants which derive from $U(S,\mathcal{R})$ 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 3.7 *Given a program $\mathcal{R}$ and a term t, if Algorithm 3.6 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 3.8 *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 $\theta' \leq_{\mathcal{V}ar(t)} \theta$.*

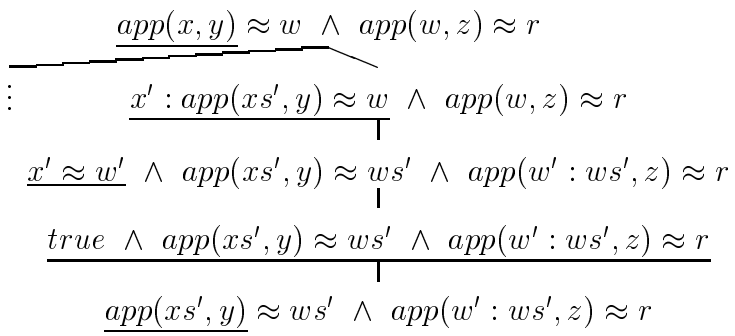


Figure 1: Naïve local control for $app(x, y) \approx w \ \wedge \ app(w, z) \approx r$.

4 Improving Control of NPE

In the following subsection, we improve control in functional logic specialization by fixing an unfolding strategy which is specifically designed for "conjunctive specialization". As for global control, a specific treatment of the primitive function symbols '$\approx$', '$\wedge$' and '$\Rightarrow$' is introduced in subsection 4.2 which produces more effective and powerful, polygenetic specializations, as compared to classical NPE.

4.1 Local Control

The unfolding rule introduced in [3] simply exploits the redexes selected by the lazy narrowing strategy φ (using a fixed, static selection rule which determines the next conjunct to be reduced) whenever none of them embed a previous (comparable) redex of the same branch. 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*:

$$\begin{array}{rcl} app([\,], y) & \rightarrow & y \\ app(x : xs, y) & \rightarrow & x : app(xs, y) \end{array}$$

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 in Figure 1 (using a fixed left-to-right selection rule for conjunctions). From this 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).

Now we introduce a refined, dynamic lazy unfolding rule which attempts to achieve a fair, balanced 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 by exploiting some dependency information between redexes gathered along the derivation. The notion of *dependent positions* is used to trace the *functional dependencies* between redexes of the local narrowing tree being constructed by PE.

Definition 4.1 (dependent positions) *Let $\mathcal{D} \equiv (s \leadsto_{p,l \to r,\sigma} t)$ be a narrowing step. The set of* dependent positions *of a position q of s by $\mathcal{D}$, denoted $q \backslash\backslash \mathcal{D}$, is:*

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

This notion can be naturally lifted to narrowing derivations.

Roughly speaking, a position q' of a term t in $\mathcal{D}$ *depends* on another position q in a previous term s, if q' and q address subterms which are "descendants" (see, e.g., [17]) of each other (second and third cases), or if the position q' has been introduced by the rhs of a rule applied in the reduction of the former position q and it does not address a subterm headed by a constructor symbol (first case). We also say that the term addressed by q is an *ancestor* of the term addressed by q' in $\mathcal{D}$. 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 $\mathcal{D}$. Note that this notion is an extension of the standard PD concept of (covering) ancestor to the functional logic framework.

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

Definition 4.2 *Let $\mathcal{D} \equiv (t_0 \leadsto t_1 \leadsto \ldots \leadsto t_n)$, $n \geq 0$, be a lazy narrowing derivation. We define the dynamic selection rule $\varphi_{dynamic}$ as: $\varphi_{dynamic}(t_n, \mathcal{D}) = select(t_n, \Lambda, \mathcal{D})$, where the auxiliary function select is:*

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

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

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

Informally, 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 just one of the components (among those that do not inccur into a *dependency_clash*). This is safe since all scheduling policies are admissible for an interpreter implementing lazy narrowing. We introduce a concrete, 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" $\bot$ of Definition 4.2 is used as a whistle to warn us that the derivation must be cut off because it runs into a *dependency_clash*. That is, each branch $\mathcal{D}$ of the tree is stopped whenever $\varphi_{dynamic}(t, \mathcal{D}) = \{\bot\}$ or the term t (of the leaf) is in head normal form. $U_{dynamic}(t, \mathcal{R})$ produces a finite lazy narrowing tree [1].

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. 2. ¿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 (see Example 4).

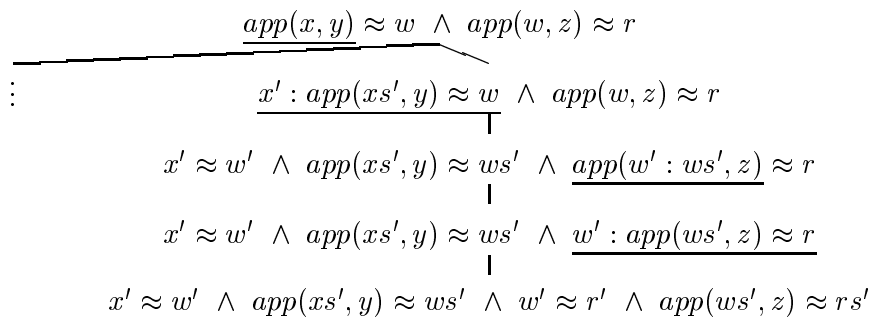


Figure 2: Improved local control for $app(x,y) \approx w \wedge app(w,z) \approx r$.

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, 20], and which uses the homeomorphic embedding relation "$\trianglelefteq$" as defined in [27].

During the abstraction process, terms may require being split in order to find the best way of continuing the specialization process without risking nontermination. The following notion, 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 4.3 (best matching terms) *Given a set of terms $S = \{s_1, \ldots, s_n\}$ and a term t, 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 4.4 *Let S and T be sets of terms. Then,* $abstract_{\trianglelefteq}(S,T) =$

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

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:

$$abs\_def(S,\emptyset,t) \ = \ abs\_prim(S,\emptyset,t) \ = S \cup \{t\}$$

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

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

where $def(t)$ denotes a sequence with the defined *function symbols of t in lexicographical order, and $\stackrel{\wedge}{=}$ 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 3.5 [1].

The following result establishes the termination of the global specialization process.

Theorem 4.5 *Algorithm 3.6 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 Figure 2. By applying Algorithm 3.6, 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 4.4, by two recursive calls to *abs_prim*, we get:

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

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. A detailed description of the benchmarks used for the analysis can be found in [1]. Some of them are typical PD benchmarks (see [19]) adapted to a functional logic syntax, while others come from the literature of functional program transformations, such as positive supercompilation [28], fold/unfold transformations [7], and deforestation [29].

We have considered the following settings to test the benchmarks:

– Evaluation strategy: All benchmarks were executed by lazy narrowing.

– Unfolding rule: We have three alternatives: a) `emb_goal`: it expands derivations while new goals do not embed a previous comparable goal in the same branch; b) `emb_redex`: the concrete unfolding rule of Section 4.1 which implements the *dependency_clash* test

	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: Benchmark results.

using homeomorphic embedding on comparable ancestors of selected redexes; and c) `comp_redex`: the concrete unfolding rule of Section 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. 4.4.

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]. The complete code for benchmarks as well as the specialized calls 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 "$append(append(x, y), z) \approx r$", but in many cases it can be appropriate (or necessary) to decompose nested expressions as in logic programming, and write "$append(x, y) \approx w \ \wedge \ append(w, z) \approx r$". 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 (and a slowdown is commonly produced). For this we could not achieve some of the standard, difficult transformations such as tupling [7] 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). However, this does not suffice to achieve tupling in practice.

The results in this paper demonstrate that it is possible to supply the NPE general framework with appropriate control options to specialize complex expressions containing primitive functions, thus providing a powerful polygenetic specialization framework

with no ad-hoc setting. The figures in Table 1 demonstrate that the control refinements that we have incorporated into the INDY system provide satisfactory speedups on all benchmarks. Our extensions are *conservative* in the sense that there is no penalty w.r.t. the specialization achieved by the original system on programs written in a pure functional style (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, although the speedups achieved by these strategies are somewhat similar, `emb_redex` is inherently more complex and it very often expands local narrowing trees beyond the "optimal" point.

There is still room for further improvement in performance within our framework, such as 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).

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 DSIC-II/12/98, UPV, 1998. Available from URL: `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*, pages 151–162. ACM, New York, 1997.

[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. Technical Report DSIC-II/11/98, UPV, 1998.

[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] C. Consel and O. Danvy. Tutorial notes on Partial Evaluation. In *Proc. of 20th Annual ACM Symp. on Principles of Programming Languages*, pages 493–501. ACM, 1993.

[10] 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.

[11] J. Gallagher. Tutorial on Specialisation of Logic Programs. In *Proc. of PEPM'93*, pages 88–98. ACM, New York, 1993.

[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 O. Danvy, R. Glück, and P. Thiemann, editors, *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. Term Rewriting Systems. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume I, pages 1–112. Oxford University Press, 1992.

[18] J. Komorowski. An Introduction to Partial Deduction. In A. Pettorossi, editor, *Meta-Programming in Logic, Uppsala, Sweden*, pages 49–69. Springer LNCS 649, 1992.

[19] M. Leuschel. The ECCE partial deduction system and the DPPD library of benchmarks. Technical report, Accessible via `http://www.cs.kuleuven.ac.be/~lpai`, 1998.

[20] 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.

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

[22] 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, Tallinn (Estonia)*, pages 184–200. Springer LNCS 714, 1993.

[23] B. Martens and J. Gallagher. Ensuring Global Termination of Partial Deduction while Allowing Flexible Polyvariance. In *Proc. of ICLP'95*, pages 597–611. MIT Press, 1995.

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

[25] P. Padawitz. *Computing in Horn Clause Theories*, volume 16 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Berlin, 1988.

[26] A. Pettorossi and M. Proietti. Transformation of Logic Programs: Foundations and Techniques. *Journal of Logic Programming*, 19,20:261–320, 1994.

[27] M.H. Sørensen and R. Glück. An Algorithm of Generalization in Positive Supercompilation. In J.W. Lloyd, editor, *Proc. of ILPS'95*, pages 465–479. The MIT Press, 1995.

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

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