Removing Superfluous Versions in Polyvariant Specialization of Prolog Programs

Claudio Ochoa[1], Germán Puebla[1], and Manuel Hermenegildo[1,2]

[1] School of Computer Science, Technical U. of Madrid ,
{cochoa,german,herme}@fi.upm.es
[2] Depts. of Comp. Sci. and El. and Comp. Eng., U. of New Mexico , herme@unm.edu

Abstract. Polyvariant specialization allows generating multiple versions of a procedure, which can then be separately optimized for different uses. Since allowing a high degree of polyvariance often results in more optimized code, polyvariant specializers, such as most partial evaluators, can generate a large number of versions. This can produce unnecessarily large residual programs. Also, large programs can be slower due to cache miss effects. A possible solution to this problem is to introduce a minimization step which identifies sets of *equivalent* versions, and replace all occurrences of such versions by a single one. In this work we present a unifying view of the problem of superfluous polyvariance. It includes both partial deduction and abstract multiple specialization. As regards partial deduction, we extend existing approaches in several ways. First, previous work has dealt with pure logic programs and a very limited class of builtins. Herein we propose an extension to traditional characteristic trees which can be used in the presence of calls to external predicates. This includes all builtins, libraries, other user modules, etc. Second, we propose the possibility of collapsing versions which are not strictly equivalent. This allows trading time for space and can be useful in the context of embedded and pervasive systems. This is done by residualizing certain computations for external predicates which would otherwise be performed at specialization time. Third, we provide an experimental evaluation of the potential gains achievable using minimization which leads to interesting conclusions.

1 Introduction and Motivation

Partial evaluation (PE) of logic programs [13, 3] aims at obtaining code which is as optimized as possible by performing aggressive unfolding at the local control level, and by being as accurate as possible (generalize the least possible) at the global control level, as long as termination is guaranteed. We refer to [7] for a survey on control issues. In particular, given a fixed local control rule, different global control rules will have different effects on the polyvariance level of partial evaluation, i.e., the number of versions produced for each procedure. In general, a common heuristic is to produce as many different versions as possible, as long as termination is not compromised, the idea being that by considering different versions separately, further optimizations may be uncovered. This heuristic makes sense from the point of view of optimizing programs in terms of resolution

```
(1) main(A,B,C,D,E,F,G,H,I,J,K,L,M,N,O):-      (2) addlists([],[],[]).
      write(C),                                (3) addlists([A|B],[C|D],[H|T]):-
      addlists([4,4|A],[0,3|B],[4,7|C]),             H is A+C,
      addlists([3,3|D],[1,4|E],[4,7|F]),             addlists(B,D,T).
      addlists([3,3|G],[1,4|H],I),
      addlists([1,1|J],[3,6|K],L),
      addlists([7,1|M],[1,5|N],O).
```

Fig. 1. Adding pairs of lists.

steps, but it can produce unnecessarily large results, and may even slow down programs due to cache miss effects.

Example 1. Fig. 1 shows our running example. Predicate `addlists/3` adds the contents of two lists, using the builtin `is/2`. Clauses are numbered for later reference. A possible result of partial evaluation for the initial query `main/15` is shown in Fig. 2. Unfolding of `main/15` only performs one step since the leftmost literal `write(C)` has side-effects, and performing non-leftmost unfolding of any other literal may backpropagate bindings (as variables may be aliases) onto `write(C)`. Note that one version has been generated for each call to `addlists/3` within the body of `main/15`, plus one version for the general case. However, the four versions `addlists_2` through `addlists_5` are indeed equivalent and could be replaced by a single one, resulting in the program shown in Fig. 3.

The problem of superfluous polyvariance has been studied both in the context of abstract multiple specialization [18, 16] and in the context of partial evaluation of normal logic programs [9]. The common idea is to identify sets of versions which are *equivalent* and replace all occurrences of such versions by a single, canonical, one. This poses two questions which we address in this work: under which conditions can we consider two given versions as equivalent? And, how can we efficiently check for equivalence?

In this work, we provide a thorough analysis of these questions, comparing different approaches for controlling polyvariance, and we also extend previous approaches in two ways. First, we tackle in an accurate way the case in which

```
main(A,B,C,D,E,F,G,H,I,J,K,L,M,N,O) :- write(C),
  addlists_2([4,4|A],[0,3|B],[4,7|C]), addlists_3([3,3|D],[1,4|E],[4,7|F]),
  addlists_4([3,3|G],[1,4|H],I),       addlists_5([1,1|J],[3,6|K],L),
  addlists_6([7,1|M],[1,5|N],O).

addlists_1([],[],[]).
addlists_1([A|B],[C|D],[E|F]) :-             E is A+C, addlists_1(B,D,F).

addlists_2([4,4],[0,3],[4,7]).
addlists_2([4,4,A|B],[0,3,C|D],[4,7,E|F]) :- E is A+C, addlists_1(B,D,F).

addlists_3([3,3],[1,4],[4,7]).
addlists_3([3,3,A|B],[1,4,C|D],[4,7,E|F]) :- E is A+C, addlists_1(B,D,F).

addlists_4([3,3],[1,4],[4,7]).
addlists_4([3,3,A|B],[1,4,C|D],[4,7,E|F]) :- E is A+C, addlists_1(B,D,F).

addlists_5([1,1],[3,6],[4,7]).
addlists_5([1,1,A|B],[3,6,C|D],[4,7,E|F]) :- E is A+C, addlists_1(B,D,F).

addlists_6([7,1],[1,5],[8,6]).
addlists_6([7,1,A|B],[1,5,C|D],[8,6,E|F]) :- E is A+C, addlists_1(B,D,F).
```

Fig. 2. Specialization of addlists/3 via partial evaluation.

```
main(A,B,C,D,E,F,G,H,I,J,K,L,M,N,O) :- write(C),
  addlists_5([4,4|A],[0,3|B],[4,7|C]), addlists_5([3,3|D],[1,4|E],[4,7|F]),
  addlists_5([3,3|G],[1,4|H],I), addlists_5([1,1|J],[3,6|K],L),
  addlists_6([7,1|M],[1,5|N],O) .

addlists_1([],[],[]).
addlists_1([A|B],[C|D],[E|F]) :-  E is A+C, addlists_1(B,D,F) .

addlists_5([A,A],[_1,_2],[4,7]).
addlists_5([A,A,B|C],[_1,_2,D|E],[4,7,F|G]) :- F is B+D, addlists_1(C,E,G).

addlists_6([7,1],[1,5],[8,6]).
addlists_6([7,1,A|B],[1,5,C|D],[8,6,E|F]) :- E is A+C, addlists_1(B,D,F).
```

Fig. 3. Specialization of addlists/3 after minimization.

programs contain *external predicates*, i.e., predicates whose code is not defined in the program being specialized, and thus it is not available to the specializer. This includes predicates defined in other user modules, library predicates, builtins, predicates implemented in other languages, etc. Note that external predicates may have *impure* features. The minimization shown in Figure 3 is not possible in previous works such as [9] as it involves calls to the builtin predicate `is/2`, which is not *safe* in the sense that it may produce bindings during its execution.

Second, previously proposed minimization techniques do not provide any degrees of freedom at the minimization stage. We propose the possibility of collapsing versions which are not *strictly* equivalent. This is achieved by residualizing certain computations for external predicates which would otherwise be performed at specialization time. This allows automatically trading time for space and can be of interest in the context of embedded and pervasive systems, where computing resources and storage are often limited.

A completely different approach to that studied in this paper is to incorporate within the global control certain heuristics which limit polyvariance based for example on characteristic trees [2, 6, 12]. Such approach has both advantages and disadvantages. The advantage is that there is no need to perform a post minimization phase, such as that discussed in this paper. On the other hand, the disadvantage of that approach is that it sometimes produces results which are suboptimal, since the fact that characteristic trees are equal not always means that the corresponding versions should be merged.

We argue that a minimization phase is important in specialization algorithms, since it allows using very accurate global control rules while limiting the risk of generating large residual code. Rather than deciding *a priori* the best global control possible, this technique allows using aggressive control strategies. We can minimize the program *a posteriori* and eliminate those specialized versions which are redundant.

2 Background

We assume some basic knowledge on the terminology of logic programming. See for example [14] for details. Very briefly, an *atom* A is a syntactic construction of the form $p(t_1, \ldots, t_n)$, where p/n, with $n \geq 0$, is a predicate symbol and $t_1, \ldots, t_n$ are terms. The function *pred* applied to atom A, i.e., $pred(A)$, returns the predicate symbol p/n for A. A *clause* is of the form $H \leftarrow B$ where its head

H is an atom and its body B is a conjunction of atoms. A *definite program* is a finite set of clauses. A *goal* (or query) is a conjunction of atoms. We denote by $\{X_1 \mapsto t_1, \ldots, X_n \mapsto t_n\}$ the *substitution* σ with $\sigma(X_i) = t_i$ for all $i = 1, \ldots, n$ (with $X_i \neq X_j$ if $i \neq j$) and $\sigma(X) = X$ for any other variable X, where t_i are terms. We denote with ϵ the empty substitution. Also, $dom(\sigma)$ denotes the set of variables affected by substitution σ, i.e., $dom(\{X_1 \mapsto t_1, \ldots, X_n \mapsto t_n\}) = \{X_1, \ldots, X_n\}$.

A term t is *more general* than s (or s is an *instance* of t), in symbols $t \leq s$, if $\exists\sigma.\ t\sigma = s$. Two terms t and t' are *variants*, denoted $t \approx t'$, if there exists a renaming ρ such that $t\rho = t'$. A *unifier* of a pair of terms $\{t_1, t_2\}$ is a substitution σ such that $t_1\sigma = t_2\sigma$. 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 another term t such that $\exists\theta_1, \ldots, \theta_n$ with $t_i = t\theta_i$, $i = 1, \ldots, n$. A generalization t is the *most specific generalization* (*msg*) of $\{t_1, \ldots, t_n\}$ if for every other term t' s.t. t' is a generalization of $\{t_1, \ldots, t_n\}$, $t' \leq t$. Given a set of clauses $\{Cl_1 = H_1 \leftarrow B_1, \ldots, Cl_n = H_n \leftarrow B_n\}$, $n \geq 0$, we denote by $instantiate(\{Cl_1, \ldots, Cl_n\}, A)$ the set of clauses $\{Cl_1\theta_1, \ldots, Cl_n\theta_n\}$ where each $\theta_i = mgu(H_i, A)$.

2.1 Basics of Partial Evaluation

Traditional algorithms for on-line partial evaluation of logic programs (known as *partial deduction* (PD) [13, 3]) usually include two control levels: *local control* and *global control* [3]. Local control defines an unfolding rule. Given an atom A, an *unfolding rule* computes a set of finite SLD derivations $D_1, \ldots, D_n$ (i.e., a possibly incomplete SLD tree) of the form $D_i = A, \ldots, G_i$ with computed answer substitution θ_i for $i = 1, \ldots, n$. We use $U(P, G) = \tau$ to denote the fact that the unfolding rule U when applied to goal G in program P returns the SLD tree τ. The global control rule decides when and how to generalize atoms before applying the unfolding rule to them. Such generalization steps are necessary in order to guarantee that the number of atoms to which the unfolding rule is applied remains finite. We refer to [7] for a survey on both control issues.

3 A General View of Polyvariance and Minimization

We now present a very general description of a polyvariant specialization process which includes both partial evaluation [13, 3, 7] and abstract multiple specialization [16]. Given a program P and a set of atoms $\mathcal{Q} = \{A_1, \ldots, A_m\}$, which describe the possible initial queries to P, polyvariant specialization performs the following three steps:

1. *Analysis.* In this phase, we compute a set of call patterns $\{A_1, \ldots, A_n\} \supseteq \mathcal{Q}$ which cover all calls in the specialized program. We write $Analysis(P, \mathcal{Q}) = \{A_1, \ldots, A_n\}$ to denote that the result of analysis for P and $\mathcal{Q}$ is the set of call patterns $\{A_1, \ldots, A_n\}$.
2. *Code Generation.* The aim of this phase is, for each call pattern $A_i \in Analysis(P, \mathcal{Q})$, to compute properly optimized residual code. We denote by $code(A_i)$ the code (set of clauses) associated to A_i. In partial evaluation, an unfolding rule U is used for generating code, i.e., $code(A_i) = U(P, A_i)$.

3. *Renaming.* In this phase we assign a fresh predicate name to each atom in $\{A_1, \ldots, A_n\}$. Then, for each $code(A_i)$, we perform appropriate renamings in the head and body atoms so that each program point uses a correct (and as optimized as possible) version. *Ren* denotes the renaming function.

The polyvariant specialized program P_Q is then defined as:

$$P_Q = \bigcup_{A_i}^{Analysis(P,Q)} Ren(code(A_i))$$

3.1 Minimizing the Results of Polyvariant Specialization

The aim of minimization is to group the call patterns (or *versions*) in $\{A_1, \ldots, A_n\}$ into *equivalence classes*, obtaining a minimal program that allows the same set of optimizations, and that can be implemented without introducing run-time tests to select amongst different versions of a predicate.

Deciding whether two versions A_i and A_j with $pred(A_i) = pred(A_j)$ are equivalent is not straightforward, as we have to consider not only the code of A_i and A_j, but also the code of all other versions which are reachable from them. In the case of the *main* predicate in a program, we would have to take the code of all the specialized program into account. Thus, we will split the notion of equivalence into a *local equivalence* and a *global equivalence* level. Local equivalence concentrates on comparing the code for A_i and A_j only, without worrying about the other predicates which are reachable from them. Global equivalence will only hold if A_i and A_j are locally equivalent and all reachable versions for the corresponding program points are also locally equivalent.

The minimization algorithm (called *Minimize* from now on) consists of two phases. In [17], the first phase is called *reunion* and the second phase is called *splitting.* The reunion phase is concerned with local equivalence only and it places together all versions for the same predicate which are considered locally equivalent according to some criteria. The splitting phase is concerned with global equivalence. It splits sets of versions which are not globally equivalent until no more splitting is needed, i.e., until we have reached a partition where all sets contain versions which are globally equivalent. This minimization process is isomorphic to the minimization of deterministic finite automata (DFA) [5], by considering each call pattern A_i as a *state* and each program point in $code(A_i)$ as a *symbol.*

A crucial point thus is, given a pair of atoms A and A', to decide whether they can be safely considered locally equivalent. The decision criteria has to satisfy two properties: (1) it must produce correct results, and (2) it must be effective, i.e. it must be possible to efficiently decide whether A and A' are candidates for equivalence based on syntactic, local conditions. For this purpose, in this work we introduce *structural equivalence.*

Definition 1 (structurally equivalent). *Let A_1 and A_2 be two call patterns such that $pred(A_1) = pred(A_2)$. We say that A_1 and A_2 are* structurally equivalent *iff*

$$\begin{aligned} & C = msg(code(A_1), code(A_2)) \\ \wedge\ & instantiate(C, A_1) \approx code(A_1) \\ \wedge\ & instantiate(C, A_2) \approx code(A_2) \end{aligned}$$

Clearly, if $code(A_1) \approx code(A_2)$ then A_1 and A_2 are structurally equivalent. However the definition above allows also considering as structurally equivalent call patterns whose code only differs in constants which are input arguments to the predicate but which do not play an important role for local optimization. Note that structural equivalence is just a syntactic characterization which guarantees that two call patterns are locally equivalent. In fact, there can be call patterns which are locally equivalent in the sense that their behaviours under the semantics of interest are identical but which our definition of structural equivalence would not capture. Also, structural equivalence in particular, and local equivalence in general do not guarantee global equivalence. It often happens that two call patterns which are structurally equivalent end up in different equivalence classes after the splitting phase. Only after this phase terminates we can be sure that two call patterns are globally equivalent.

The polyvariant specialized program with minimization P_Q^{Min} is defined as:

$$P_Q^{Min} = \bigcup_{V_i}^{Minimize(Analysis(P,Q))} Ren_{\equiv}(code(V_i))$$

where given a set of atoms $\{A_1, \ldots, A_n\}$, we partition them in equivalence classes $\{V_1, \ldots, V_k\}$, $k \geq n$ s.t. $\forall A, A' \in V_i$. A and A' are structurally equivalent. We use $code(\{A_1, \ldots, A_i\})$ to denote $msg(\{code(A_1), \ldots, code(A_i)\})$. Also, $Ren_{\equiv}$ is a new renaming function which always uses the same (*canonical*) predicate name for any atom in $\{A_1, \ldots, A_i\}$.

Our definition of structural equivalence plays several roles. It underlies the notions of local equivalence used both in abstract multiple specialization and partial deduction, thus allowing us to present a unified view of both minimization processes. Furthermore, it can also be used in order to determine whether two versions are locally equivalent. Existing approaches to minimization do not compare the syntactic structure of the residual code directly (as this definition would require) but rather use the specialization history in order to decide local equivalence. In [16] two call patterns are considered locally equivalent iff (1) they correspond to the same predicate in the original program and (2) the set of optimizations in both call patterns is the same. In [9] two call patterns are locally equivalent iff they have the same characteristic tree.

4 Characteristic Trees with External Predicates

A *characteristic tree* [2] is a data structure which encapsulates the evaluation behaviour of an atom, i.e., a trace of the unfolding process. The following definitions are taken from [9], which in turn were derived from [2].

Definition 2 (characteristic path). *Let G_0 be a goal, and let P be a definite program whose clauses are numbered. Let $G_0, \ldots, G_n$ be the goals of a finite, possibly incomplete SLD-derivation D of $P \cup \{G_0\}$. The characteristic path of the derivation D is the sequence $\langle l_0 : c_0, \ldots, l_{n-1} : c_{n-1}\rangle$, where l_i is the position of the selected atom in G_i, and c_i is the number of the clause chosen to resolve with G_i.*

Now that we have characterized derivations, we can characterize goals through the derivations in their associated SLD-trees.

Definition 3 (characteristic tree). *Let G be a goal, P a definite program, and τ a finite SLD-tree for $P \cup \{G\}$. Then the characteristic tree $\hat{\tau}$ of τ is the set containing the characteristic paths of the nonfailing SLD-derivations associated with the branches of τ.*
Let U be an unfolding rule such that $U(P, G) = \tau$. Then $\hat{\tau}$ is also called the characteristic tree of G (in P) via U. We introduce the notation $ch\_tree(G, P, U) = \hat{\tau}$.

Although existing partial evaluation systems such as SP [1] and ECCE [10] perform some limited handling of builtins within characteristic trees, the existing formal definitions of characteristic trees do not contemplate the existence of builtins nor of external predicates. We now extend the standard definitions in order to accurately include external predicates.

Definition 4 (chpath with external predicates). *Let G_0 be a goal, and let P be a program whose clauses are numbered. Let $G_0, \ldots, G_n$ be the goals of a finite, possibly incomplete SLD-derivation D of $P \cup \{G_0\}$. Let $A_0, \ldots, A_{n-1}$ be the selected atoms in D. The* characteristic path with external predicates *of the derivation D is the sequence $\langle l_0 : c_0, \ldots, l_{n-1} : c_{n-1} \rangle$, where l_i is the position of A_i in G_i, and c_i is defined as follows:*

- *if $pred(A_i)$ is defined in P, then c_i is the number of the clause in P chosen to resolve with G_i;*
- *if $pred(A_i)$ is an external predicate, then let θ be a computed answer generated when performing $exec(A_i)$. Then, c_i is a pair (A_i, θ).*

In the definition above, $exec(A_i)$ represents the execution of A_i. For this, the external call A_i has to be *evaluable* [15], i.e., A_i is both well-moded and well-typed, it does not produce any side-effect, and it universally terminates. Note that $exec(A_i)$ can succeed more than once and possibly with different computed answers. Reconsidering characteristic paths, each pair $(l_i : c_i)$ in a characteristic path must uniquely identify: (1) the position of the selected atom A_i, (2) the bindings introduced by this step on the current goal, and (3) the atoms which must be introduced in the goal in place of the selected atom A_i. An important obvious difference between external and regular predicates is that the code for external predicates may not be available, so it is not possible, as done with regular predicates, to assign clause numbers to them or to unfold them. Instead of unfolding external predicates, we will fully execute them. As a result, no atoms will be introduced in the current goal and, thus, (3) is not needed in this case.

In the case of external predicates, we introduce in the characteristic tree an *external success*, i.e., a pair (A_i, θ) containing the call pattern A_i and the bindings θ generated during evaluation for each external predicate. Note that, in contrast to the handling of builtins within characteristic trees in SP and ECCE, this makes it possible to reconstruct the residual code for an atom without the need for (re-)evaluating external predicates, even if the external predicates succeed several times with (possibly) different computed answers. The notion

$$\begin{aligned}
\tau_2 = \{&\langle 1:3, 1:(4\ is\ 4+0, \epsilon), 1:3, 1:(7\ is\ 4+3, \epsilon), 1:2\rangle,\\
&\langle 1:3, 1:(4\ is\ 4+0, \epsilon), 1:3, 1:(7\ is\ 4+3, \epsilon), 1:3\rangle\},\\
\tau_3 = \{&\langle 1:3, 1:(4\ is\ 3+1, \epsilon), 1:3, 1:(7\ is\ 3+4, \epsilon), 1:2\rangle,\\
&\langle 1:3, 1:(4\ is\ 3+1, \epsilon), 1:3, 1:(7\ is\ 3+4, \epsilon), 1:3\rangle\},\\
\tau_4 = \{&\langle 1:3, 1:(A\ is\ 3+1, \{A \mapsto 4\}), 1:3, 1:(B\ is\ 3+4, \{B \mapsto 7\}), 1:2\rangle,\\
&\langle 1:3, 1:(C\ is\ 3+1, \{C \mapsto 4\}), 1:3, 1:(D\ is\ 3+4, \{D \mapsto 7\}), 1:3\rangle\},\\
\tau_5 = \{&\langle 1:3, 1:(E\ is\ 1+3, \{E \mapsto 4\}), 1:3, 1:(F\ is\ 1+6, \{F \mapsto 7\}), 1:2\rangle,\\
&\langle 1:3, 1:(G\ is\ 1+3, \{G \mapsto 4\}), 1:3, 1:(H\ is\ 1+6, \{H \mapsto 7\}), 1:3\rangle\}.\\
\tau_6 = \{&\langle 1:3, 1:(I\ is\ 7+1, \{I \mapsto 8\}), 1:3, 1:(J\ is\ 1+5, \{J \mapsto 6\}), 1:2\rangle,\\
&\langle 1:3, 1:(L\ is\ 7+1, \{L \mapsto 8\}), 1:3, 1:(M\ is\ 1+5, \{M \mapsto 6\}), 1:3\rangle\}.
\end{aligned}$$

Fig. 4. Characteristic trees for addlists/3 versions.

of characteristic paths with external predicates is indeed consistent with traditional characteristic paths. In the case of regular predicates, the same *implicit* representation as in traditional characteristic paths is used. This representation is efficient in space since rather than introducing (an instantiated version of) the clause chosen for resolving the selected atom directly in the characteristic tree, only the number of the clause used for unfolding is stored. This suffices since the actual instantiation can be performed later if needed using the actual clause. In the case of external predicates, this implicit representation is no longer possible, since the clauses are not available. Instead, the call pattern and the corresponding bindings are *explicitly* stored.

Characteristic trees are extended to handle external predicates by simply considering characteristic paths with external predicates. Fig. 4 shows the characteristic trees with external predicates τ_2, τ_3, τ_4, τ_5 and τ_6 for versions `addlists_2/3`, `addlists_3/3`, `addlists_4/3`, `addlists_5/3`, and `addlists_6/3`, respectively.

5 Isomorphic Characteristic Trees

In this section we define the notion of *isomorphic* characteristic trees with external predicates, which guarantees that the corresponding code is structurally equivalent. We assume that predicate names cannot be numbers, as is the case in most existing logic programming systems. Also, *number(X)* succeeds iff X is a number.

First, we introduce the concept of *quasi-isomorphic characteristic trees*, for identifying characteristic trees which only (possibly) differ in the input and/or output values of arguments in calls to external predicates:

Definition 5 (quasi-isomorphic characteristic trees). *Two characteristic paths* $\delta^1 = \langle l_0 : c_0^1, \ldots, l_m : c_m^1\rangle$ *and* $\delta^2 = \langle l_0 : c_0^2, \ldots, l_m : c_m^2\rangle$ *are* quasi-isomorphic *and we denote it* $\delta^1 \approx_q \delta^2$ *iff* $\forall i \in \{1..m\}\ .\ number(c_i^1) \Rightarrow c_i^1 = c_i^2$. *Two characteristic trees* τ_1 *and* τ_2 *are* quasi-isomorphic*, denoted* $\tau_1 \approx_q \tau_2$*, iff* $\forall \delta^1 \in \tau_1\ .\ \exists \delta^2 \in \tau_2$ *s.t.* $\delta^1 \approx_q \delta^2$ *and* $\forall \delta^2 \in \tau_2\ .\ \exists \delta^1 \in \tau_1$ *s.t.* $\delta^2 \approx_q \delta^1$.

Note that quasi-isomorphic characteristic paths must have the same length and the selected atom must be in the same position in each resolution step. Furthermore, if the atom is not for an external predicate, then the atom must have been resolved against the same clause. In Fig. 4, $\tau_2 \approx_q \tau_3 \approx_q \tau_4 \approx_q \tau_5 \approx_q \tau_6$.

Now we define some relationships among external successes, after some auxiliary definitions. A *position* uniquely determines a subterm within a term.

Definition 6 (Position). *A* position ω *is either the empty position* ε, *or* $n.\omega'$, *where* n *is a natural number and* ω' *is a position.*

Definition 7 (getval, Pos, and Allpos). *Let* $A = f(\overline{t_n})$ *be a term. Let* ω *be a position. Let* X *be a variable s.t.* $X \in vars(A)$. *Let* θ *be a substitution.*

- *We define* $getval(\omega, A)$ *as* A *if* $\omega = \varepsilon$ *and as* $getval(\omega', t_i)$ *if* $\omega = i.\omega'$.
- *We define* $Pos(A, X)$ *as* $\{\omega \mid getval(\omega, A) = X\}$.
- *We define* $Allpos(A, \theta)$ *as* $\cup_{X \in dom(\theta)}\{\omega\}, s.t.\ \omega \in Pos(A, X)$.

Example 2. $getval(2.1.\varepsilon, f(a, g(b, c))) = b$, and $Pos(f(a, g(b, Y)), Y) = \{2.2.\varepsilon\}$. If A is not linear, then for some X, the set $Pos(A, X)$ may have more than one element. E.g., $Pos(f(Z, g(Z)), Z) = \{1.\varepsilon, 2.1.\varepsilon\}$. In such case, any $\omega \in Pos(A, X)$ can be used for our purposes. Also $Allpos(A\ is\ 3+1, \{A \mapsto 4\}) = \{1.\varepsilon\}$.

Definition 8 (isomorphic external successes). *Let* $c = (A, \theta)$ *and* $c' = (A', \theta')$ *be external successes. Then* c *and* c' *are* isomorphic external successes, *denoted by* $c \simeq c'$, *iff* $\forall \omega \in Allpos(A, \theta) \cup Allpos(A', \theta')\ .\ getval(\omega, A\theta) = getval(\omega, A'\theta')$.

Example 3. This definition tries to consider as isomorphic as many pairs of external successes as possible. A particular subcase of this definition corresponds to the case where the calls to external predicates generate no bindings. For example, the pair $(4\ is\ 4{+}0, \epsilon)$ and $(4\ is\ 3{+}1, \epsilon)$ is isomorphic, whereas the notion of equivalence in [9] cannot capture this since the builtin predicate `is/2` potentially generates bindings, though in this case it does not. Note that $(4 is 4+0, \epsilon)$ and $(8 is 2*4, \epsilon)$ are also considered as isomorphic although their syntactic structure is very different. Another interesting subcase is when the external successes have different levels of instantiation but on success they are variants. This happens with $(A\ is\ 3{+}1, \{A \mapsto 4\})$ and $(4\ is\ 3{+}1, \epsilon)$. Furthermore, it allows considering as isomorphic external successes which have the same values in all positions which are instantiated in either external success. For example $(A\ is\ 3+1, \{A \mapsto 4\})$ and $(4\ is\ 4+0, \epsilon)$ are considered isomorphic since $Allpos(A\ is\ 3+1, \{A \mapsto 4\}) = \{1.\varepsilon\} \wedge Allpos(4\ is\ 4{+}0, \epsilon\}) = \emptyset \wedge getval(1.\varepsilon, 4\ is\ 3{+}1) = getval(1.\varepsilon, 4\ is\ 4{+}0) = 4$. However, $(E\ is\ 1{+}3, \{E \mapsto 4\}) \not\simeq (I\ is\ 7{+}1, \{I \mapsto 8\})$, since $Allpos(E\ is\ 1{+}3, \{E \mapsto 4\}) = Allpos(I\ is\ 7{+}1, I) = \{1.\varepsilon\}$, but $getval(1.\varepsilon, 4\ is\ 1{+}3) = 4 \neq getval(1.\varepsilon, 8\ is\ 7{+}1) = 8$.

Definition 9 (isomorphic characteristic trees). *Two characteristic paths* $\delta^1 = \langle l_0 : c^1_0, \ldots, l_m : c^1_m\rangle$ *and* $\delta^2 = \langle l_0 : c^2_0, \ldots, l_m : c^2_m\rangle$ *are* isomorphic *and we denote it* $\delta^1 \approx \delta^2$ *iff* $\delta^1 \approx_q \delta^2 \wedge \forall i \in \{1..m\}\ .\ c^1_i = (A^1_i, \theta^1_i) \Rightarrow c^2_i = (A^2_i, \theta^2_i) \wedge c^1_i \simeq c^2_i$. *Two characteristic trees* τ_1 *and* τ_2 *are* isomorphic, *denoted* $\tau_1 \approx \tau_2$, *iff* $\forall \delta^1 \in \tau_1\ .\ \exists \delta^2 \in \tau_2$ *s.t.* $\delta^1 \approx \delta^2$ *and* $\forall \delta^2 \in \tau_2\ .\ \exists \delta^1 \in \tau_1$ *s.t.* $\delta^2 \approx \delta^1$.

The following proposition provides the basis for our minimization approach.

Proposition 1 (structural equivalence). *Let* P *be a program with external predicates, let* U *be an unfolding rule, let* A_1 *and* A_2 *be two call patterns such that* $\tau_1 = ch\_tree(A_1, P, U)$ *and* $\tau_2 = ch\_tree(A_2, P, U)$. *If* $\tau_1 \approx \tau_2$ *then* A_1 *and* A_2 *are structurally equivalent.*

A difficulty with our notion $\approx$ of isomorphic characteristic trees and its usage as a condition for local equivalence is that though the $\approx$ relation is reflexive and symmetric, it is not transitive. This means that $(\tau_1 \approx \tau_2 \wedge \tau_2 \approx \tau_3) \not\to \tau_1 \approx \tau_3$. As a result, in order to be able to state that all characteristic trees in a set $\{\tau_1, \ldots, \tau_n\}$ are isomorphic we have to check that $\forall \tau, \tau' \in \{\tau_1, \ldots, \tau_n\} . \tau \approx \tau'$.

Example 4. Let us consider again the characteristic trees in Fig. 4. We have already noticed that all of them are quasi-isomorphic. If we take the quasi-isomorphic paths of τ_2, τ_3, τ_4 and τ_5, and extract their external successes, we can see that they are isomorphic. For example, if we take $c_{21} = (4\ is\ 4+0, \epsilon)$, $c_{31} = (4\ is\ 3+1, \epsilon)$, $c_{41} = (A\ is\ 3+1, \{A \mapsto 4\})$ and $c_{51} = (C\ is\ 1+3, \{C \mapsto 4\})$, we can compute $\cup_{i \in \{2 \ldots 5\}} Allpos(c_{i1}) = \{1.\epsilon\}$. Since $getval(1.\epsilon, 4\ is\ 4+0) = getval(1.\epsilon, 4\ is\ 3+1) = getval(1.\epsilon, 4\ is\ 1+3) = 4$, we can conlude that they are isomorphic.

Finally, note that even though $\tau_5 \approx_q \tau_6$, they are not (fully) isomorphic since, for instance, $(E\ is\ 1+3, \{E \mapsto 4\}) \not\approx (I\ is\ 7+1, \{I \mapsto 8\})$. Indeed, `addlists_5/3` and `addlists_6/3` are not structurally equivalent. As a result, the sets which are identified as locally equivalent during the reunion phase are: {{`main/15`}, {`addlists_1/3`},{`addlists_2/3,addlists_3/3,addlists_4/3,addlists_5/3`}, {`addlists_6/3`}}. This is also the final partition after applying the splitting phase. This produces the minimized program which was shown in Fig. 3.

6 Minimization via Residualization of External Calls

There are situations in which even the minimized program is too large and/or where we would like to trade space for time efficiency. This would mean achieving programs which are smaller, but at the cost of introducing some efficiency penalty. In cases like this, we propose as candidates for minimization, call patterns with *quasi-isomorphic* characteristic trees. An important observation is that if $\delta^1 \approx_q \delta^2$ then the associated resultants have the same structure. However, this is not a sufficient condition for structural equivalence. This is because part of the bindings needed for structural equivalence cannot be achieved by the operation *instantiate*, as in Def. 1, but rather they originate from the execution of calls to external predicates. Thus, the second important observation is that if the calls to external predicates involved succeed only once, i.e. they are deterministic, such missing bindings can be recovered at run-time by residualizing (part of the) calls to external predicates which had in principle taken place during specialization time. Note that for detecting determinacy, no static analysis is actually required. We can simply check whether the calls which are to be residualized succeed just once by directly executing the calls as they appear in the different characteristic trees, i.e., before applying the msg to them. After the required external predicates have been residualized, the corresponding versions will be structurally equivalent.
The strategy we propose is the following: for any pair of versions A_1 and A_2 with $\tau_1 = ch\_tree(A_1, P, U)$ and $\tau_2 = ch\_tree(A_2, P, U)$ s.t. $\tau_1 \approx_q \tau_2$ we

$$\frac{msg\begin{pmatrix}\{\mathit{addlists}([4,4],[0,3],[4,7]).,\langle 1:(4\ \mathit{is}\ 4+0),1:(7\ \mathit{is}\ 4+3)\rangle\}\\ \{\mathit{addlists}([3,3],[1,4],[4,7]).,\langle 1:(4\ \mathit{is}\ 3+1),1:(7\ \mathit{is}\ 3+4)\rangle\}\\ \{\mathit{addlists}([3,3],[1,4],[4,7]).,\langle 1:(4\ \mathit{is}\ 3+1),1:(7\ \mathit{is}\ 3+4)\rangle\}\\ \{\mathit{addlists}([1,1],[3,6],[4,7]).,\langle 1:(4\ \mathit{is}\ 1+3),1:(7\ \mathit{is}\ 1+6)\rangle\}\end{pmatrix}}{\{\mathit{addlists}([X,X],[Y,Z],[4,7]).,\langle 1:(4\ \mathit{is}\ X+Y),1:(7\ \mathit{is}\ X+Z)\rangle\}}$$

$$\frac{msg\begin{pmatrix}\begin{array}{r}\{\mathit{addlists}([4,4,A|B],[0,3,C|D],[4,7,E|F]):-E\ \mathit{is}\ A+C,\mathit{addlists}(B,D,F).,\\ \langle 1:(4\ \mathit{is}\ 4+0),1:(7\ \mathit{is}\ 4+3)\rangle\}\\ \{\mathit{addlists}([3,3,A|B],[1,4,C|D],[4,7,E|F]):-E\ \mathit{is}\ A+C,\mathit{addlists}(B,D,F).,\\ \langle 1:(4\ \mathit{is}\ 3+1),1:(7\ \mathit{is}\ 3+4)\rangle\}\\ \{\mathit{addlists}([3,3,A|B],[1,4,C|D],[4,7,E|F]):-E\ \mathit{is}\ A+C,\mathit{addlists}(B,D,F).,\\ \langle 1:(4\ \mathit{is}\ 3+1),1:(7\ \mathit{is}\ 3+4)\rangle\}\\ \{\mathit{addlists}([1,1,A|B],[3,6,C|D],[4,7,E|F]):-E\ \mathit{is}\ A+C,\mathit{addlists}(B,D,F).,\\ \langle 1:(4\ \mathit{is}\ 1+3),1:(7\ \mathit{is}\ 1+6)\rangle\}\end{array}\end{pmatrix}}{\begin{array}{r}\{\mathit{addlists}([X,X,R|S],[Y,Z,T|U],[4,7,V|W]):-V\ \mathit{is}\ R+T,\mathit{addlists}(S,U,W).,\\ \langle 1:(4\ \mathit{is}\ X+Y),1:(7\ \mathit{is}\ X+Z)\rangle\}\end{array}}$$

Fig. 5. *msg* of versions `addlists_2`, `addlists_3`, `addlists_4` and `addlists_5`.

1. Compute $(C, T) = msg((code(A_1), \overline{\tau}_1), (code(A_2), \overline{\tau}_2))$, where $\forall i \in \{1..2\}.\overline{\tau}_i$ is obtained from τ_i by *evaluating* all external successes, i.e., $\forall(B, \theta)$ we replace it by $B\theta$.
2. **If** $\forall i \in \{1..2\}$. $instantiate(C, A_i) \approx code(A_i)$
   - **then** A_1 and A_2 are structurally equivalent. No need to residualize.
   - **else** if for every evaluated external success $c \in T$ such that c is no longer sufficiently instantiated to be executed we can determine that its corresponding $c_1 \in \tau_1$ and $c_2 \in \tau_2$ are both *deterministic*,
     - then *residualize* all $c \in T$ being no longer sufficiently instantiated.
     - otherwise we cannot collapse A_1 and A_2.

Note that without such residualization, the code generated by the *msg* is not directly usable, since there are bindings in the original versions which are lost if we apply the code produced by the *msg*.

Example 5. As we have already mentioned, all characteristic trees in Fig. 4 are quasi-isomorphic. Therefore, they can be collapsed into one version. In Fig. 5 we show the *msg* of both the code and the characteristic trees for versions `addlists_2`, `addlists_3`, `addlists_4` and `addlists_5`. In this figure, the scope of variables is local to each clause. Since $\tau_2 \approx \tau_3 \approx \tau_4 \approx \tau_5$, the *msg* does not produce any information loss. This can be easily verified by instantiating back the *msg* with any of the call patterns. For instance, if we take `addlists([X,X],[Y,Z],[4,7])` and instantiate it with `addlists([3,3|G],[1,4|H],I)` we obtain the original clause (eighth clause of Fig. 2).

Example 6. Now, let us now compute the *msg* of the generalized code and characteristic tree obtained in Example 5 with `addlists_6`.

$$\frac{msg\begin{pmatrix}\{\mathit{addlists}([X,X],[Y,Z],[4,7]).,\langle 1:(4\ \mathit{is}\ X+Y),1:(7\ \mathit{is}\ X+Z)\rangle\}\\ \{\mathit{addlists}(\ [7,\ 1],[\ 1,5],[8,6]).,\langle 1:(8\ \mathit{is}\ \ 7+\ 1),1:(6\ \ \mathit{is}\ \ 1+5)\rangle\}\end{pmatrix}}{\{\mathit{addlists}([A,B],[C,D],[E,F]).,\langle 1:(E\ \mathit{is}\ A+C),1:(F\ \mathit{is}\ B+D)\rangle\}}$$

$$\frac{msg\begin{pmatrix}\begin{array}{r}\{\mathit{addlists}([X,X,R|S],[Y,Z,T|U],[4,7,V|W]):-V\ \mathit{is}\ R+T,\mathit{addlists}(S,U,W).,\\ \langle 1:(4\ \mathit{is}\ X+Y),1:(7\ \mathit{is}\ X+Z)\rangle\}\\ \{\mathit{addlists}([7,\ 1,R|S],[\ 1,5,T|U],[8,6,V|W]):-V\ \mathit{is}\ R+T,\mathit{addlists}(S,U,W).,\\ \langle 1:(8\ \mathit{is}\ \ 7+1),1:(6\ \mathit{is}\ \ 1+5)\rangle\}\end{array}\end{pmatrix}}{\begin{array}{r}\{\mathit{addlists}([A,B,G|H],[C,D,I|J],[E,F,K|L]):-K\ \mathit{is}\ G+I,\mathit{addlists}(H,J,L).,\\ \langle 1:(E\ \mathit{is}\ A+C),1:(F\ \mathit{is}\ B+D)\rangle\}\end{array}}$$

```
main(A,B,C,D,E,F,G,H,I,J,K,L,M,N,O) :- write(A),
   addlists_6([4,4|A],[0,3|B],[4,7|C]), addlists_6([3,3|D],[1,4|E],[4,7|F]),
   addlists_6([3,3|G],[1,4|H],I), addlists_6([1,1|J],[3,6|K],L),
   addlists_6([7,1|M],[1,5|N],O) .

addlists_1([],[],[]).
addlists_1([A|B],[C|D],[E|F]) :- E is A+C, addlists_1(B,D,F) .

addlists_6([A,B],[C,D],[E,F]) :-  E is A+C, F is B+D.
addlists_6([A,B,G|H],[C,D,I|J],[E,F,K|L]) :- E is A+C, F is B+D,
   K is G+I, addlists_1(H,J,L) .
```

Fig. 6. Specialization of addlists/3 after minimization with residualization.

Since `addlists_6` is not (fully) isomorphic with the other versions, the *msg* introduces some information loss through the variables E and F in the new heads $addlists([A, B], [C, D], [E, F])$ and $addlists([A, B, G|H], [C, D, I|J], [E, F, K|L])$. This information loss cannot be recovered by *instantiate*, since, for example, when instantiating the *msg* `addlists([A,B],[C,D],[E,F])` with the call pattern `addlists([3,3|G],[1,4|H],I)` we obtain `addlists([3,3],[1,4],[E,F])`, in which E and F are unbound variables. If we take the external successes which correspond to `E is A+C` and `F is B+D` we can verify that the original external successes were deterministic (indeed, all calls to `is/2` are deterministic). Thus, it is possible to collapse by residualization. As both external calls are no longer sufficiently instantiated, they are residualized. Residualized atoms are always placed before any other atom in the generalized clause, guaranteeing that after execution of such residual atoms at run-time, the clause as a whole is actually a variant of the original definition of the clause. The resulting minimized program is shown in Fig. 6. Residual atoms are underlined to distinguish them from the rest of atoms in body clauses.

7 Experimental Results

In this section we assess experimentally the impact of our proposed minimization. Most of the benchmarks considered contain calls to builtins which possibly generate bindings, such as `is/2`, and thus the existing partial evaluators which perform minimization [10, 11] would not be able to minimize them optimally.

In our experiments we use an unfolding rule based on homeomorphic embedding (see, e.g., [7]) and which performs leftmost unfolding steps only. This guarantees the correctness of the partial evaluation process even in the presence of impure predicates. Note that the issue of redundant polyvariance may occur for any unfolding rule. The global control rule is based on homeomorphic embedding and global trees [8]. All benchmarks have been run on an Intel Pentium 4, 3.4 GHz processor, with 512 Mb of RAM, and running a 2.6 Linux kernel.

7.1 The Benefits of Minimization

Table 1 shows the size reduction introduced by the minimization step after partial evaluation. Each benchmark program is evaluated using five different minimization criteria, as shown in the *Min Crit* column. Specialization history is used in *pure*, *nobinds*, and *bindings*, in order to consider two versions as locally equivalent, while *codemsg* directly applies the definition of structural equivalence for

Benchmark	**Min Crit**	**Orig Preds**	**Minimization**					
			Versions			**Size (bytes)**		
			PE	**Min**	**Ratio**	**PE**	**Min**	**Ratio**
datetime	pure	15	56/31	36/36	1.78	131377	102651	1.28
	nobinds			36/36	1.78		102836	1.28
	bindings			34/35	1.83		102331	1.28
	codemsg			34/35	1.83		102295	1.28
	residual			31/33	1.94		100976	1.30
flattrees	pure	2	33/16	22/22	1.50	226390	223320	1.01
	nobinds			22/22	1.50		223435	1.01
	bindings			22/22	1.50		223389	1.01
	codemsg			17/19	1.74		221513	1.02
	residual			16/18	1.83		220796	1.03
freeof	pure	3	93/8	35/35	2.66	292642	245262	1.19
	nobinds			35/35	2.66		245442	1.19
	bindings			32/35	2.66		245370	1.19
	codemsg			18/35	2.66		245334	1.19
	residual			8/35	2.66		245370	1.19
mmatrix_2	pure	3	70/11	18/34	2.06	58323	37061	1.57
	nobinds			18/34	2.06		37236	1.57
	bindings			18/34	2.06		37166	1.57
	codemsg			18/34	2.06		37131	1.57
	residual			11/30	2.33		31781	1.84
nrev_38	pure	2	41/3	3/3	13.67	25115	5261	4.77
	nobinds			3/3	13.67		5281	4.76
	bindings			3/3	13.67		5273	4.76
	codemsg			3/3	13.67		5269	4.77
	residual			3/3	13.67		5273	4.76
qsort_33	pure	3	168/50	68/68	2.47	232079	166288	1.40
	nobinds			50/50	3.36		131650	1.76
	bindings			50/50	3.36		131548	1.76
	codemsg			50/50	3.36		131497	1.76
	residual			50/50	3.36		131548	1.76
sublists	pure	4	29/19	27/27	1.11	101969	99986	1.02
	nobinds			27/27	1.11		100121	1.02
	bindings			19/19	1.58		95815	1.06
	codemsg			19/19	1.58		95795	1.06
	residual			19/19	1.58		95815	1.06
Overall				2.88 / 2.96			1.32 / 1.33	

Table 1. Benchmarks (Minimization Ratios)

the same purpose. In particular, *pure* considers two versions as locally equivalent when their characteristic trees are identical. Of course, if external successes are included, these must be identical too. The criteria *nobinds* and *bindings* check for isomorphism of external successes instead. *Nobinds* only considers two external successes c and c' as isomorphic when they generate no bindings, i.e., when $\mathit{Allpos}(c) = \mathit{Allpos}(c) = \emptyset$, while *bindings* applies the full power of Def. 8. Finally, *residual* considers two versions as candidates for minimization when their characteristic trees are quasi-isomorphic, possibly residualizing calls to external predicates in the resulting program.

The number of predicates in the original program is shown in the column *Orig Preds*. The number of predicates in the specialized programs are shown under the column *Versions*. *PE* shows both the number of versions which are generated after partial evaluation (i.e., the effects of polyvariance) and the number of sets of predicates with quasi-isomorphic characteristic trees. The latter provides a lower bound on the number of predicates which the minimized program may have. *Min* shows the number of elements in the partition generated by the reunion phase of the minimization algorithm (local equivalence) and the number of elements in the partition after the splitting phase (global equivalence). Finally, *Ratio* shows the reduction ratio for each criteria compared to the number of versions produced by partial evaluation. The column *Size* compares the sizes of the compiled bytecode of programs minimized using the different criteria.

The last row, *Overall*, shows the weighted geometric mean (*wgm*) for ratios in terms of number of versions and size. Weights are number of versions and size of the *PE* column, respectively. In both cases, under the column *Min* we find the *wgm* of the *codemsg* criterion, which achieves the best results while still producing programs of maximal optimization. Under the column *Ratio* we find the *wgm* of the *residual* criterion, which achieves highest ratio.

As can be seen in the table, in most of the benchmarks considered, minimization is capable of considerably reducing the specialized program, both in terms of number of versions and of bytecode size. As it is to be expected, out of the four criteria which are guaranteed to produce programs of maximal optimization, i.e., *pure*, *bindings*, *nobinds*, and *codemsg*, the one which produces the best results is the latter. Among the three of them which take the minimization history into account—and which are more efficient in terms of specialization time—, the best is *bindings*, but it sometimes does not produce as good results as *codemsg*. The effects of the splitting phase are clear in many benchmarks, showing that, in effect, local equivalence does not imply global equivalence. Finally, for `datetime`, `flattrees` and `mmatrix_2`, *residual* is able to further reduce code size.

7.2 The Cost of Minimization

In Table 2 we can observe the cost, in terms of specialization time, introduced by minimization, expressed in milliseconds. The (*Total*) time of the whole specialization process is shown, including the time taken by the partial evaluation (*Analysis*), minimization (*Minim*), and code generation (*Codegen*) steps. A new minimization criteria is introduced, *nomin*, showing the time employed by partial evaluation without minimization. The *Slowdown* column shows the cost of performing this minimization post-processing.

Interestingly, the table shows that when minimization is employed, the code generation phase takes less time in most cases, since fewer versions need to be generated. This lowers the burden introduced by minimization post-processing. However, even in the worst case the slowdown introduced is reasonable (1.85). As expected, using specialization history makes minimization faster than just applying the definition of structural equivalence. Given the fact that employing structural equivalence generates fewer versions than other criteria based on the specialization history, the *codemsg* criterion emerges as a very interesting one.

Benchmark	Minimization Criteria	Minimization Times (msec)				
		Total	Analysis	Minim	Codegen	Slowdown
datetime	nomin	556.52	475.33	0	81.19	1
	pure	632.90	486.13	61.19	85.59	1.14
	nobinds	634.30	476.13	72.79	85.39	1.14
	bindings	640.10	479.93	73.99	86.19	1.15
	codemsg	642.30	478.13	79.19	84.99	1.15
	residual	687.30	479.93	77.59	129.78	1.23
flattrees	nomin	299.55	232.56	0	66.99	1
	pure	395.14	230.97	107.78	56.39	1.32
	nobinds	396.34	231.57	108.58	56.19	1.32
	bindings	400.19	230.21	113.48	56.49	1.34
	codemsg	412.74	231.36	125.98	55.39	1.38
	residual	424.94	231.36	125.78	67.79	1.42
freeof	nomin	5732.93	5583.15	0	149.78	1
	pure	5833.11	5589.95	118.98	124.18	1.02
	nobinds	5844.11	5589.15	131.38	123.58	1.02
	bindings	5858.31	5573.35	160.38	124.58	1.02
	codemsg	5948.90	5595.15	230.97	122.78	1.04
	residual	6113.47	5613.95	221.97	277.56	1.07
mmatrix_2	nomin	316.15	271.76	0	44.39	1
	pure	356.55	272.76	48.39	35.39	1.13
	nobinds	367.14	274.56	57.39	35.19	1.16
	bindings	364.34	272.76	55.99	35.59	1.15
	codemsg	373.34	274.96	63.19	35.19	1.18
	residual	435.53	270.76	60.79	103.98	1.38
nrev_38	nomin	898.26	877.07	0	21.20	1
	pure	886.67	861.27	13.20	12.20	0.99
	nobinds	901.86	872.67	16.80	12.40	1.00
	bindings	898.86	870.27	16.40	12.20	1.00
	codemsg	903.86	874.67	17.20	12.00	1.01
	residual	916.26	873.87	17.20	25.20	1.02
qsort_33	nomin	9983.68	9745.12	0	238.56	1
	pure	10267.64	9778.91	282.96	205.77	1.03
	nobinds	10303.83	9768.12	337.75	197.97	1.03
	bindings	10339.03	9771.91	368.94	198.17	1.04
	codemsg	10401.82	9764.92	441.73	195.17	1.04
	residual	11241.69	9732.72	371.14	1137.83	1.13
sublists	nomin	401.94	293.56	0	108.38	1
	pure	647.70	295.35	278.56	73.79	1.61
	nobinds	651.90	297.75	281.36	72.79	1.62
	bindings	679.50	295.56	280.16	103.78	1.69
	codemsg	681.30	297.56	278.76	104.98	1.70
	residual	744.09	296.95	284.56	162.57	1.85

Table 2. Benchmarks (Minimization Times)

Also, for the *residual* minimization criterion, the time spent in code generation is greater than for the rest of criteria, since it requires deciding which external successes need to be residualized.

Benchmark	PE Time	Speedup				
		Pure	No Binds	Bindings	CodeMsg	Residual
datetime*	167.77	1.01	1.02	1.01	1.01	1.01
flattrees*	81.39	1.03	1.01	1.01	1.03	1.01
freeof	246.96	1.04	1.04	1.05	1.04	1.05
mmatrix_2*	1920.11	1.02	1.02	1.02	1.02	1.00
nrev_38	141.38	1.20	1.18	1.18	1.19	1.19
qsort_33	457.33	1.05	1.04	1.04	1.05	1.04
sublists	15501.44	1.00	1.00	1.00	1.00	1.00

Table 3. Benchmarks (Speedup)

7.3 Benefits of Minimization in Runtime

Table 3 shows how specialized programs behave in terms of runtime. Benchmark programs having residualized external predicates (for the *residual* minimization criterion) are marked with * in the table. Column *PE Time* shows the absolute run-time for the partially evaluated program. The rest of the columns show the speedup achieved for the minimized programs (for each different minimization criteria) w.r.t. *PE Time*. As can be seen in the table, in most benchmarks a small speedup is achieved (1.00 – 1.20), and no slowdown is produced in any case. As expected, in the case of programs with residualized external predicates, the speedup achieved is usually smaller than for the other minimization criteria.

8 Discussion and Related Work

The problem of superfluous polyvariance has been tackled in the context of abstract multiple specialization in [18, 16], and in the context of partial evaluation of normal logic programs in [9]. This work presents a unifying view under which the minimization problems in both contexts are isomorphic.

The work in [9], reflected in the ECCE [10] partial evaluator, uses an internal table of *safe* builtins which basically correspond to instantiation and type tests and which are guaranteed (1) not to generate any bindings, and (2) to be deterministic. The minimization phase then would only allow collapsing two predicates in the same version if their characteristic trees are quasi-isomorphic and all the builtins executed are listed in the table of pure predicates.

The approach presented herein, and implemented in the Ciao system preprocesor, CiaoPP [4], can handle any external predicate, including non-safe builtins, and the notion of isomorphic external predicates can be satisfied for builtins which generate bindings and which are non-deterministic. Also, there is no need for a static table of builtins. Additionally, the technique automatically applies to any external predicates, for example other modules written by the user.

To the best of our knowledge, this work presents the first experimental evaluation of the benefits of post-minimization in partial deduction. We have compared several criteria, with different cost and potential benefit. We have also applied directly the definition of structural equivalence and discovered that it is also applicable in practice, in addition to the other criteria based on the specialization history. Finally, we have proposed a criteria which allows residualizing external calls. The experiments show that it is also applicable in practice and provides some further program reduction.

References

1. J. Gallagher. A system for specialising logic programs. Technical Report TR-91-32, University of Bristol, November 1991.
2. J. Gallagher and M. Bruynooghe. The derivation of an algorithm for program specialisation. *New Generation Computing*, 9(1991):305–333, 1991.
3. J.P. Gallagher. Tutorial on specialisation of logic programs. In *Proc. of PEPM'93*, pages 88–98. ACM Press, 1993.
4. M. Hermenegildo, F. Bueno, G. Puebla, and P. López-García. Program Analysis, Debugging and Optimization Using the Ciao System Preprocessor. In *1999 ICLP*, pages 52–66. MIT Press, Nov 1999.
5. J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.
6. M. Leuschel. Ecological partial deduction: Preserving characteristic trees without constraints. In *Proc. of LOPSTR'95*, LNCS 1048, pages 1–16. Springer, 1995.
7. M. Leuschel and M. Bruynooghe. Logic program specialisation through partial deduction: Control issues. *TPLP*, 2(4 & 5):461–515, July & September 2002.
8. M. Leuschel and B. Martens. Global control for partial deduction through characteristic atoms and global trees. In *1996 Dagstuhl Seminar on Partial Evaluation*, LNCS 1110, pages 263–283, Schloß Dagstuhl, 1996.
9. M. Leuschel, B. Martens, and D. De Schreye. Controlling generalisation and polyvariance in partial deduction of normal logic programs. *ACM TOPLAS*, 20(1):208–258, 1998.
10. Michael Leuschel. The ECCE partial deduction system and the DPPD library of benchmarks. Obtainable via `http://www.ecs.soton.ac.uk/~mal`, 1996-2002.
11. Michael Leuschel. *Advanced Techniques for Logic Program Specialisation*. PhD thesis, K.U. Leuven, May 1997.
12. Michael Leuschel and Danny De Schreye. Constrained partial deduction and the preservation of characteristic trees. *New Generation Computing*, 16:283–342, 1998.
13. J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *The Journal of Logic Programming*, 11:217–242, 1991.
14. J.W. Lloyd. *Foundations of Logic Programming*. Springer, 2nd Ext. Ed., 1987.
15. G. Puebla, E. Albert, and M. Hermenegildo. Efficient Local Unfolding with Ancestor Stacks for Full Prolog. In *Proc. of LOPSTR'04*, pages 149–165. Springer LNCS 3573, 2005.
16. G. Puebla and M. Hermenegildo. Implementation of Multiple Specialization in Logic Programs. In *Proc. of PEPM'95*, pages 77–87. ACM Press, June 1995.
17. G. Puebla and M. Hermenegildo. Abstract Multiple Specialization and its Application to Program Parallelization. *JLP*, 41(2&3):279–316, November 1999.
18. W. Winsborough. Multiple Specialization using Minimal-Function Graph Semantics. *Journal of Logic Programming*, 13(2 and 3):259–290, July 1992.