Redundant Argument Filtering of Logic Programs

Michael Leuschel[1] & Morten Heine Sørensen[2]

[1] Department of Computer Science, Katholieke Universiteit Leuven, Celestijnenlaan 200A, B-3001, Heverlee, Belgium, michael@cs.kuleuven.ac.be.
[2] Department of Computer Science, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen, Denmark. rambo@diku.dk.

Abstract. This paper is concerned with the problem of removing, from a given logic program, *redundant arguments*. These are arguments which can be removed without affecting correctness. Most program specialisation techniques, even though they perform argument filtering and redundant clause removal, fail to remove a substantial number of redundant arguments, yielding in some cases rather inefficient residual programs. We formalise the notion of a redundant argument and show that one cannot decide effectively whether a given argument is redundant. We then give a safe, effective approximation of the notion of a redundant argument and describe several simple and efficient algorithms calculating based on the approximative notion. We conduct extensive experiments with our algorithms on mechanically generated programs illustrating the practical benefits of our approach.

1 Introduction

Automatically generated programs often contain redundant parts. For instance, programs produced by standard partial deduction [20] often have *useless clauses* and *redundant structures*, see e.g. [7]. This has motivated uses of *regular approximations* to detect useless clauses [9, 6, 5] and the *renaming* (or *filtering*) transformation [8, 2] that removes redundant structures. In this paper we are concerned with yet another notion of redundancy which may remain even after these transformations have been applied, viz. *redundant arguments*. These seem to appear particularly often in programs produced by *conjunctive partial deduction* [17, 11], a recent extension of standard partial deduction which can perform tupling and deforestation.

For example, consider the goal $\leftarrow$ *doubleapp*(*Xs*, *Ys*, *Zs*, *R*) and the program:

$$\begin{array}{ll} \mathit{doubleapp}(\mathit{Xs}, \mathit{Ys}, \mathit{Zs}, R) & \leftarrow \mathit{app}(\mathit{Xs}, \mathit{Ys}, \underline{T}), \mathit{app}(\underline{T}, \mathit{Zs}, R) \\ \mathit{app}([\,], \mathit{Ys}, \mathit{Ys}) & \leftarrow \\ \mathit{app}([H|\mathit{Xs}], \mathit{Ys}, [H|\mathit{Zs}]) & \leftarrow \mathit{app}(\mathit{Xs}, \mathit{Ys}, \mathit{Zs}) \end{array}$$

Given *Xs*, *Ys*, *Zs*, the goal $\leftarrow$ *doubleapp*(*Xs*, *Ys*, *Zs*, *R*) concatenates the three lists *Xs*, *Ys*, *Zs* yielding as result *R*. This is achieved via two calls to *app* and the local variable *T*. The first call to *app* constructs from the lists *Xs* and *Ys*

an intermediate list T, which is then traversed when appending Zs. While the goal $doubleapp(Xs, Ys, Zs, R)$ is simple and elegant, it is rather inefficient since construction and traversal of such intermediate data structures is expensive.

Partial deduction within the framework of Lloyd and Shepherdson [20] cannot substantially improve the program since the atoms $app(Xs, Ys, T)$, $app(T, Zs, R)$ are transformed independently. However, as shown in [17, 11], conjunctive partial deduction of $\leftarrow doubleapp(Xs, Ys, Zs, R)$ gives the following equivalent program:

$$\begin{array}{ll}
doubleapp(Xs, Ys, Zs, R) & \leftarrow da(Xs, Ys, \underline{T}, Zs, R) \\
da([\,], Ys, \underline{Ys}, Zs, R) & \leftarrow a(Ys, Zs, R) \\
da([H|Xs'], Ys, \underline{[H|T']}, Zs, [H|R']) & \leftarrow da(Xs', Ys, \underline{T'}, Zs, R') \\
a([\,], Ys, Ys) & \leftarrow \\
a([H|Xs'], Ys, [H|Zs']) & \leftarrow a(Xs', Ys, Zs')
\end{array}$$

Here the concatenation of the lists Xs and Ys is still stored in T, but is not used to compute the result in R. Instead the elements encountered while traversing Xs and Ys are stored directly in R. Informally, the third argument of da is redundant. Thus, although this program represents a step in the right direction, we would rather prefer the following program:

$$\begin{array}{ll}
doubleapp(Xs, Ys, Zs, R) & \leftarrow da'(Xs, Ys, Zs, R) \\
da'([\,], Ys, Zs, R) & \leftarrow a'(Ys, Zs, R) \\
da'([H|Xs'], Ys, Zs, [H|R']) & \leftarrow da'(Xs', Ys, Zs, R') \\
a'([\,], Ys, Ys) & \leftarrow \\
a'([X|Xs], Ys, [X|Zs]) & \leftarrow a'(Xs, Ys, Zs)
\end{array}$$

Automation of the step from $da/5$ to $da'/4$ was left open in [17, 11] (although correctness conditions were given in [17]). The step cannot be obtained by the renaming operation in [8, 2] which only improves programs where some atom in some body contains functors or multiple occurrences of the same variable. In fact, this operation has *already* been employed by conjunctive partial deduction to arrive at the program with $da/5$. The step also cannot be obtained by other transformation techniques, such as partial deduction itself, or the more specific program construction of [22] which calculates more specific versions of programs. Indeed, any method which preserves the least Herbrand model, or the computed answer semantics for all predicates, is incapable of transforming $da/5$ to $da'/4$.

Redundant arguments also appear in a variety of other situations. For instance, they appear in programs generated by standard partial deduction when conservative unfolding rules are used.

As another example, redundant arguments arise when one re-uses general predicates for more specific purposes. For instance, define the $member/2$ predicate by re-using a general $delete/3$ predicate:

$$\begin{array}{ll}
member(X, L) & \leftarrow delete(X, L, DL) \\
delete(X, [X|T], T) & \leftarrow \\
delete(X, [Y|T], [Y|DT]) & \leftarrow delete(X, T, DT)
\end{array}$$

Here the third argument of $delete$ is redundant but cannot be removed by any of the techniques cited above.

In this paper we rigorously define the notion of a redundant argument, and show that the problem of removing all redundant arguments is undecidable. We then present an efficient algorithm which computes a safe approximation of the redundant arguments and removes them. Correctness of the technique is also established. On a range of programs produced by conjunctive partial deduction (with renaming), an implementation of our algorithm reduces code size and execution time by an average of approximately 20%. The algorithm never increases code size nor execution time.

2 Correct Erasures

In the remainder we adopt the terminology and notation from [19]. Moreover, $Pred(P)$ denotes the set of predicates occurring in a logic program P, $arity(p)$ denotes the arity of a predicate p, $Clauses(P)$ denotes the set of clauses in P, $Def(p, P)$ denotes the definitions of p in P, and $vars(U)$ denotes the set of variables occurring in U, where U may be a term, an atom, a conjunction, a goal, or a program. An atom, conjunction, goal, or program, in which every predicate has arity 0 is called *propositional*. In this paper all goals and programs are *definite*, except when explicitly stated otherwise.

In this section we formalise *redundant arguments* in terms of *correct erasures*.

Definition 1. Let P be a program.

1. An *erasure* of P is a set of tuples (p, k) with $p \in Pred(P)$, and $1 \leq k \leq arity(p)$.
2. The *full erasure* for P is $\top_P = \{(p, k) \mid p \in Pred(P) \wedge 1 \leq k \leq arity(p)\}$.

The effect of applying an erasure to a program is to erase a number of arguments in every atom in the program. For simplicity of the presentation we assume that, for every program P and goal G of interest, each predicate symbol occurs only with one particular arity (this will later ensure that there are no unintended name clashes after erasing certain argument positions).

Definition 2. Let G be a goal, P a program, and E an erasure of P.

1. For an atom $A = p(t_1, \ldots, t_n)$ in P, let $1 \leq j_1 < \ldots < j_k \leq n$ be all the indexes such that $(p, j_i) \notin E$. We then define $A|E = p(t_{j_1}, \ldots, t_{j_k})$.
2. $P|E$ and $G|E$ arise by replacing every atom A by $A|E$ in P and G, respectively.

How are the semantics of P and $P|E$ of Def. 2 related? Since the predicates in P may have more arguments than the corresponding predicates in $P|E$, the two programs have incomparable semantics. Nevertheless, the two programs may have the same semantics for some of their arguments.

Example 1. Consider the program P:

$$p(0, 0, 0) \leftarrow$$
$$p(s(X), f(Y), g(Z)) \leftarrow p(X, Y, Z)$$

The goal $G = \leftarrow p(s(s(0)), B, C)$ has exactly one SLD-refutation, with computed answer $\{B/f(f(0)), C/g(g(0))\}$. Let $E = \{(p, 3)\}$, and hence $P|E$ be:

$$p(0, 0) \leftarrow$$
$$p(s(X), f(Y)) \leftarrow p(X, Y)$$

Here $G|E = \leftarrow p(s(s(0)), B)$ has exactly one SLD-refutation, with computed answer $\{B/f(f(0))\}$. Thus, although we have erased the third argument of p, the computed answer for the variables in the remaining two arguments is not affected. Taking finite failures into account too, this suggests a notion of equivalence captured in the following definition.

Definition 3. An erasure E is *correct* for a program P and a goal G iff

1. $P \cup \{G\}$ has an SLD-refutation with computed answer θ with $\theta' = \theta\,|_{vars(G|E)}$ iff $P|E \cup \{G|E\}$ has an SLD-refutation with computed answer θ'.
2. $P \cup \{G\}$ has a finitely failed SLD-tree iff $P|E \cup \{G|E\}$ has.

Given a goal G and a program P, we may now say that the i'th argument of a predicate p is *redundant* if there is an erasure E which is correct for P and G and which contains (p, i). However, we will continue to use the terminology with correct erasures, rather than redundant arguments.

Usually there is a certain set of argument positions I which we do not want to erase. For instance, for $G = app([a], [b], R)$ and the append program, the erasure $E = \{(app, 3)\}$ is correct, but applying the erasure will also make the result of the computation invisible. In other words, we wish to retain some arguments because we are interested in their values (see also the examples in Sect. 4). Therefore we only consider subsets of $\top_P \setminus I$ for some I. Not all erasures included in $\top_P \setminus I$ are of course correct, but among the correct ones we will prefer those that remove more arguments. This motivates the following definition.

Definition 4. Let G be a goal, P a program, $\mathcal{E}$ a set of erasures of P, and $E, E' \in \mathcal{E}$.

1. E is *better than* E' iff $E \supseteq E'$.
2. E is *strictly better than* E' iff E is better than E' and $E \neq E'$.
3. E is *best* iff no other $E' \in \mathcal{E}$ is strictly better than E.

Proposition 5. *Let G be a goal, P a program and $\mathcal{E}$ a collection of erasures of P. Among the correct erasures for P and G in $\mathcal{E}$ there is a best one.*

Proof. There are only finitely many erasures in $\mathcal{E}$ that are correct for P and G. Just choose one which is not contained in any other. □

Best correct erasures are not always unique. For $G = \leftarrow p(1, 2)$ and P:

$$p(3, 4) \leftarrow q$$
$$q \leftarrow$$

both $\{(p,1)\}$ and $\{(p,2)\}$ are best correct erasures, but $\{(p,1),(p,2)\}$ is incorrect.

The remainder of this section is devoted to proving that best correct erasures are uncomputable. The idea is as follows. It is decidable whether $P \cup \{G\}$ has an SLD-refutation for propositional P and G, but not for general P and G. The full erasure of any P and G yields propositional $P|\top_P$ and $G|\top_P$. The erasure is correct iff both or none of $P \cup \{G\}$ and $P|\top_P \cup \{G|\top_P\}$ have an SLD-refutation. Thus a test to decide correctness, together with the test for SLD-refutability of propositional formulae, would give a general SLD-refutability test.

Lemma 6. *There is an effective procedure that decides, for propositional program P and goal G, whether $P \cup \{G\}$ has an SLD-refutation.*
Proof. By a well-known [19, Cor 7.2, Thm. 8.4] result, $P \cup \{G\}$ has an SLD-refutation iff $P \cup \{G\}$ is unsatisfiable. The latter problem is decidable, since P and G are propositional. □

Lemma 7. *Let G be a goal, P a program, and E an erasure of P. If $P \cup \{G\}$ has an SLD-refutation, then so has $P|E \cup \{G|E\}$.*
Proof. By induction on the length of the SLD-derivation of $P \cup \{G\}$. □

Lemma 8. *Let P be a program and G a goal. If $P \cup \{G\}$ has an SLD-refutation, then $P \cup \{G\}$ has no finitely failed SLD-tree.*
Proof. By [19, Thm. 10.3]. □

Proposition 9. *There is no effective procedure that tests, for a program P and goal G, whether $\top_P$ is correct for P and G.*
Proof. Suppose such an effective procedure exists. Together with the effective procedure from Lemma 6 this would give an effective procedure to decide whether $P \cup \{G\}$ has an SLD-refutation, which is known to be an undecidable problem:[3]

1. If $P|\top_P \cup \{G|\top_P\}$ has no SLD-refutation, by Lemma 7 neither has $P \cup \{G\}$.
2. If $P|\top_P \cup \{G|\top_P\}$ has an SLD-refutation then:
   (a) If $\top_P$ is correct then $P \cup \{G\}$ has an SLD-refutation by Def. 3.
   (b) If $\top_P$ is incorrect then $P \cup \{G\}$ has no SLD-refutation. Indeed, if $P \cup \{G\}$ had an SLD-refutation with computed answer θ, then Def. 3(1) would be satisfied with $\theta' = \theta\,|_{vars(G|\top_P)} = \emptyset$. Moreover, by Lemma 8 none of $P \cup \{G\}$ and $P|\top_P \cup \{G|\top_P\}$ would have a finitely failed SLD-tree, so Def. 3(2) would also be satisfied. Thus $\top_P$ would be correct, a contradiction. □

Corollary 10. *There is no effective function that maps any program P and goal G to a best, correct erasure for P and G.*
Proof. $\top_P$ is the best among all erasures of P, so such a function f would satisfy:

$$f(P,G) = \top_P \quad \Leftrightarrow \quad \top_P \text{ is correct for } P \text{ and } G$$

giving an effective procedure to test correctness of $\top_P$, contradicting Prop. 9. □

[3] $G|\top_P$ may contain variables, namely those occurring in atoms with predicate symbols not occurring in P. However, such atoms are equivalent to propositional atoms not occurring in P.

3 Computing Correct Erasures

In this section we present an algorithm which computes correct erasures. Corollary 10 shows that we cannot hope for an algorithm that computes *best* correct erasures. We therefore derive an approximate notion which captures some interesting cases. For this purpose, the following examples illustrate some aspects of correctness.

The first example shows what may happen if we try to erase a variable that occurs several times in the body of a clause.

Example 2. Consider the following program P:

$$\begin{array}{ll} p(X) & \leftarrow r(X,Y), q(Y) \\ r(X,1) & \leftarrow \\ q(0) & \leftarrow \end{array}$$

If $E = \{(r,2)\}$ then $P|E$ is the program:

$$\begin{array}{l} p(X) \leftarrow r(X), q(Y) \\ r(X) \leftarrow \\ q(0) \leftarrow \end{array}$$

In P the goal $G = \leftarrow p(X)$ fails finitely, while in $P|E$ the goal $G|E = \leftarrow p(X)$ succeeds. Thus E is not correct for P and G. The source of the problem is that the existential variable Y links the calls to r and q with each other. By erasing Y in $\leftarrow r(X,Y)$, we also erase the synchronisation between r and q.

Also, if $E = \{(q,1),(r,2)\}$ then $P|E$ is the program:

$$\begin{array}{l} p(X) \leftarrow r(X), q \\ r(X) \leftarrow \\ q \quad \leftarrow \end{array}$$

Again, $G|E = \leftarrow p(X)$ succeeds in $P|E$, so the problem arises independently of whether the occurrence of Y in $q(Y)$ is itself erased or not.

In a similar vein, erasing a variable that occurs several times within the same call, but is not linked to other atoms, can also be problematic.

Example 3. If P is the program:

$$\begin{array}{ll} p(a,b) & \leftarrow \\ p(f(X),g(X)) & \leftarrow p(Y,Y) \end{array}$$

and $E = \{(p,2)\}$ then $P|E$ is the program:

$$\begin{array}{ll} p(a) & \leftarrow \\ p(f(X)) & \leftarrow p(Y) \end{array}$$

Here $G = \leftarrow p(f(X),Z)$ fails finitely in P, while $G|E = \leftarrow p(f(X))$ succeeds (with the empty computed answer) in $P|E$.

Note that, for $E = \{(p,1),(p,2)\}$, $P|E$ is the program:

$p.$
$p \leftarrow p.$

Again $G|E =\leftarrow p$ succeeds in $P|E$ and the problem arises independently of whether the second occurrence of Y is erased or not.

Still another problem is illustrated in the next example.

Example 4. Consider the following program P:

$p([],[]) \leftarrow$
$p([X|Xs],[X|Ys]) \leftarrow p(Xs,[0|Ys])$

If $E = \{(p,2)\}$ then $P|E$ is the program:

$p([]) \leftarrow$
$p([X|Xs]) \leftarrow p(Xs)$

In P, the goal $G =\leftarrow p([1,1],Y)$ fails finitely, while in $P|E$ the goal $G|E =\leftarrow p([1,1])$ succeeds. This phenomenon can occur when erased arguments of predicate calls contain non-variable terms.

Finally, problems may arise when erasing in the body of a clause a variable which also occurs in a non-erased position of the head of a clause:

Example 5. Let P be the following program:

$p(a,b) \leftarrow$
$p(X,Y) \leftarrow p(Y,X)$

If $E = \{(p,2)\}$ then $P|E$ is the program:

$p(a) \leftarrow$
$p(X) \leftarrow p(Y)$

Here $G =\leftarrow p(c,Y)$ fails (infinitely) in P while $G|E =\leftarrow p(c)$ succeeds in $P|E$. The synchronisation of the alternating arguments X and Y is lost by the erasure.

The above criteria lead to the following sufficient definition, adapted from the definition of correct renamings in [17], in which (1) rules out Example 4, (2) rules out Examples 2 and 3, and (3) rules out Example 5.

Definition 11. Let P be a program and E an erasure of P. E is *safe* for P iff for all $(p,k) \in E$ and all $H \leftarrow C, p(t_1,\ldots,t_n), C' \in \mathit{Clauses}(P)$, it holds that:

1. t_k is a variable X.
2. X occurs only once in $C, p(t_1,\ldots,t_n), C'$.
3. X does not occur in $H|E$.

This in particular applies to goals:

Definition 12. Let P be a program and E an erasure of P. E is *safe* for a goal G iff for all $(p,k) \in E$ where $G =\leftarrow C, p(t_1,\ldots,t_n), C'$ it holds that:

1. t_k is a variable X.
2. X occurs only once in in $C, p(t_1, \ldots, t_n), C'$.

These conditions occur, in a less obvious formulation, among the conditions for Tamaki-Sato folding (see [25]). The method of this paper can be seen as a novel application of Tamaki-Sato folding using a particular control strategy.

Proposition 13. *Let G be a goal, P a program, and E an erasure of P. If E is safe for P and for G then E is correct for P and G.*

Proof. The conditions in Def. 11 and Def. 12 are equivalent to the conditions on correct renamings in [17]. Therefore, Theorem 3.7 from [17] can be invoked as follows (where we extensively use terminology from [17]). Let S be an *independent* set of *maximally general* atoms using the predicate symbols occurring in P, let p be a *partitioning function* such that $p(A_1 \wedge \ldots A_n) = \{A_1, \ldots, A_n\}$ and let the *atomic renaming* σ be such that for $A \in S$: $\alpha(A) = A|E$. Since S is independent there is only one *renaming function* $\rho_{\alpha,p}$ based on α and p. We now have that the *conjunctive partial deduction* $P_{\rho_{\alpha,p}}$, using an *unfolding rule* which just performs one single unfolding step for every $A \in S$, is identical to $P|E$. Note that the thus obtained trees are *non-trivial* wrt p and also that trivially $P_{\rho_{\alpha,p}}$ is *S-closed* wrt p. As already mentioned, safety of E implies that ρ_σ is a *correct renaming* for $P_{\rho_{\alpha,p}} \cup \{G\}$. Theorem 3.7 of [17] then shows that $P|E$ is correct for P and G. □

The following algorithm constructs a safe erasure for a given program.

Algorithm 1 (RAF)

Input: a program P, an initial erasure E_0.
Output: an erasure E with $E \subseteq E_0$.
Initialisation: $i := 0$;
while there exists a $(p, k) \in E_i$ and a $H \leftarrow C, p(t_1, \ldots, t_n), C' \in \mathit{Clauses}(P)$ s.t.:
1. t_k is not a variable; **or**
2. t_k is a variable that occurs more than once in $C, p(t_1, \ldots, t_n), C'$; **or**
3. t_k is a variable that occurs in $H|E_i$

do $E_{i+1} := E_i \setminus \{(p, k)\}$; $i := i + 1$;
return E_i

The above algorithm starts out from an initial erasure E_0, usually contained in $\top_P \setminus I$, where I are positions of interest (i.e. we are interested in the computed answers they yield). Furthermore E_0 should be so as to be safe for any goal of interest (see the example in the next section).

Proposition 14. *With input E_0, RAF terminates, and output E is a unique erasure, which is the best safe erasure for P contained in E_0.*

Proof. The proof consists of four parts: *termination* of RAF, *safety* of E for P, *uniqueness* of E, and *optimality* of E. The two first parts are obvious; termination follows from the fact that each iteration of the while loop decreases the size of E_i, and safety is immediate from the definition.

To prove uniqueness, note that the non-determinism in the algorithm is the choice of which (p,k) to erase in the while loop. Given a logic program P, let the *reduction* $E(p,k)F$ denote the fact that E is not safe for P and that an iteration of the while loop may chose to erase (p,k) from E yielding $F = E\backslash\{(p,k)\}$.

Now suppose $E(p,k)F$ and $E(q,j)G$. Then by analysis of all the combinations of reasons that (p,k) and (q,j) could be removed from E it follows that $F(q,j)H$ and $G(p,k)H$ with $H = E\backslash\{(p,k),(q,j)\}$.

This property implies that for any two sequences

$$E(p_1,k_1)F_1 \ldots F_{n-1}(p_n,k_n)F_n \quad \text{and} \quad E(q_1,j_1)G_1 \ldots G_{m-1}(q_m,j_m)G_m$$

there are sequences:

$$F_n(q_1,j_1)G'_1 \ldots G'_{m-1}(q_m,j_m)H \quad \text{and} \quad G_m(p_1,k_1)F'_1 \ldots F'_{n-1}(p_n,k_n)H$$

with $H = F_n \cap G_m$. In particular, if F_n and G_m are safe, so that no reductions apply, it follows that $F_n = G_m$. Hence the output is a unique erasure.

To see that this is the best one among the safe erasures contained in E_0, note that $E(p,k)F$ implies that no safe erasure contained in E contains (p,k). □

4 Applications, Implementation and Benchmarks

We first illustrate the usefulness of the RAF algorithm in the transformation of double-append from Sect. 1. Recall that we want to retain the semantics (and so all the arguments) of *doubleapp*, but want to erase as many arguments in the auxiliary calls to *app* and *da* as possible. Therefore we start RAF with

$$E_0 = \{(da,1),(da,2),(da,3),(da,4),(da,5),(a,1),(a,2),(a,3)\}$$

Application of RAF to E_0 yields $E = \{(da,3)\}$, representing the information that the third argument of *da* can be safely removed, as desired. By construction of E_0, we have that $E \subseteq E_0$ is safe for any goal which is an instance of $\leftarrow$ *doubleapp*(*Xs*, *Ys*, *Zs*, *R*). Hence, as long as we consider only such goals, we get the same answers from the program with $da'/4$ as we get from the one with $da/5$.

Let us also treat the member-delete problem from Sect. 1. If we start RAF with

$$E_0 = \{(delete,1),(delete2),(delete,3)\}$$

indicating that we are only interested in computed answers to $member/2$, then we obtain $E = \{(delete,3)\}$ and the following more efficient program $P|E$:

$$\begin{array}{ll} member(X,L) & \leftarrow delete(X,L) \\ delete(X,[X|T]) & \leftarrow \\ delete(X,[Y|T]) & \leftarrow delete(X,T) \end{array}$$

To investigate the effects of Algorithm 1 more generally, we have incorporated it into the ECCE partial deduction system [18]. This system is based on work in [15, 16] and was extended to perform conjunctive partial deduction based on [17, 11, 13]. We ran the system *with* and *without* redundant argument filtering (but always *with* renaming in the style of [8]) on a series of benchmarks of the DPPD library [18] (a brief description can also be found in [13]). An unfolding rule allowing determinate unfolding and leftmost "indexed" non-determinate unfolding (using the homeomorphic embedding relation on covering ancestors to ensure finiteness) was used.[4] For further details, see [13]. The timings were obtained via the *time*/2 predicate of Prolog by BIM 4.0.12 (on a Sparc Classic under Solaris) using the "benchmarker" files generated by ECCE. The compiled code size was obtained via *statistics*/4 and is expressed in units, were 1 unit corresponds to approximatively 4.08 bytes (in the current implementation of Prolog by BIM).

The results are summarised in Table 1. The weighted speedup is obtained by the formula

$$\frac{n}{\sum_{i=1}^{n} \frac{spec_i}{orig_i}}$$

where $n = 29$ is the number of benchmarks and $spec_i$ and $orig_i$ are the absolute execution times of the specialised and original programs respectively. As can be seen, RAF reduced code size by an average of 21% while at the same time yielding an average additional speedup of 18%. Note that 13 out of the 29 benchmarks benefited from RAF, while the others remained unaffected (i.e. no redundant arguments where detected). Also, none of the programs were deteriorated by RAF. Except for extremely large residual programs, the execution time of the RAF algorithm was insignificant compared to the total partial deduction time. Note that the RAF algorithm was also useful for examples which have nothing to do with deforestation and, when running the same benchmarks with standard partial deduction based on e.g. determinate unfolding, RAF also turned out to be useful, albeit to a lesser extent. In conclusion, RAF yields a practically significant reduction of code size and a practically significant speedup (e.g. reaching a factor of 4.29 for *depth*).

5 Poly-variance, Negation and Further Extensions

In this section we discuss some natural extensions of our technique.

Polyvariant Algorithm

The erasures computed by RAF are *mono-variant*: an argument of some predicate has to be erased in all calls to the predicate or not at all. It is sometimes desirable that the technique be more precise and erase a certain argument only

[4] The full system options were: Abs:j, InstCheck:a, Msv:s, NgSlv:g, Part:f, Prun:i, Sel:l, Whistle:d, Poly:y, Dpu: yes, Dce:yes, MsvPost: no.

Benchmark	Code Size		Execution Time			
	w/o RAF	with RAF	Original	w/o RAF	with RAF	Extra Speedup
advisor	809 u	809 u	0.68	0.21	0.21	1.00
applast	188 u	145 u	0.44	0.17	0.10	1.70
contains.kmp	2326 u	1227 u	1.03	0.28	0.10	2.80
contains.lam	2326 u	1227 u	0.53	0.15	0.11	1.36
depth.lam	5307 u	1848 u	0.47	0.30	0.07	4.29
doubleapp	314 u	277 u	0.44	0.42	0.35	1.20
ex_depth	874 u	659 u	1.14	0.37	0.32	1.16
flip	573 u	493 u	0.61	0.66	0.58	1.14
grammar.lam	218 u	218 u	1.28	0.18	0.18	1.00
groundunify.simple	368 u	368 u	0.28	0.07	0.07	1.00
liftsolve.app	1179 u	1179 u	0.81	0.04	0.04	1.00
liftsolve.db1	1326 u	1326 u	1.00	0.01	0.01	1.00
liftsolve.lmkng	2773 u	2228 u	0.45	0.54	0.44	1.23
map.reduce	348 u	348 u	1.35	0.11	0.11	1.00
match.kmp	543 u	543 u	2.28	1.49	1.49	1.00
match.lam	543 u	543 u	1.60	0.95	0.95	1.00
maxlength	1083 u	1023 u	0.10	0.14	0.12	1.17
model_elim.app	444 u	444 u	1.43	0.19	0.19	1.00
regexp.r1	457 u	457 u	1.67	0.33	0.33	1.00
regexp.r2	831 u	799 u	0.51	0.25	0.18	1.39
regexp.r3	1229 u	1163 u	1.03	0.45	0.30	1.50
relative.lam	261 u	261 u	3.56	0.01	0.01	1.00
remove	2778 u	2339 u	4.66	3.83	3.44	1.11
rev_acc_type	242 u	242 u	3.39	3.39	3.39	1.00
rev_acc_type.inffail	1475 u	1475 u	3.39	0.96	0.96	1.00
rotateprune	4088 u	3454 u	5.84	6.07	5.82	1.04
ssuply.lam	262 u	262 u	0.65	0.05	0.05	1.00
transpose.lam	2312 u	2312 u	1.04	0.18	0.18	1.00
Weighted Speedup			1	2.11	2.50	1.18
Average Size	1204.91 u	952.64 u				

Table 1. Code size (in units) and Execution times (in s)

in certain contexts (this might be especially interesting when a predicate also occurs inside a negation, see the next subsection below).

Example 6. Consider the following program P:

$$\begin{aligned} p(a,b) &\leftarrow \\ p(b,c) &\leftarrow \\ p(X,Y) &\leftarrow p(X,Z), p(Z,Y) \end{aligned}$$

For $E_0 = \{(p,2)\}$ (i.e. we are only interested in the first argument to p), RAF returns $E = \emptyset$ and hence $P|E = P$. The reason is that the variable Z in the call $p(X,Z)$ in the third clause of P cannot be erased. Therefore no optimisation

can occur at all. To remedy this, we need a *poly-variant algorithm* which, in the process of computing a safe erasure, generates duplicate versions of some predicates, thereby allowing the erasure to behave differently on different calls to the same predicate. Such an algorihtm might return the following erased program:

$$\begin{array}{ll} p(a) & \leftarrow \\ p(b) & \leftarrow \\ p(X) & \leftarrow p(X,Z), p(Z) \\ p(a,b) & \leftarrow \\ p(b,c) & \leftarrow \\ p(X,Y) & \leftarrow p(X,Z), p(Z,Y) \end{array}$$

The rest of this subsection is devoted to the development of such a *poly-variant* RAF algorithm.

First, the following, slightly adapted, definition of erasing is needed. The reason is that several erasures might now be applied to the same predicate, and we have to avoid clashes between the different specialised versions for the same predicate.

Definition 15. Let E be an erasure of P. For an atom $A = p(t_1, \ldots, t_n)$, we define $A||E = p_E(t_{j_1}, \ldots, t_{j_k})$ where $1 \leq j_1 < \ldots < j_k \leq n$ are all the indexes such that $(p, j_i) \notin E$ and where p_E denotes a predicate symbol of arity j_k such that $\forall p, q, E_1, E_2$ ($p_{E_1} = q_{E_2}$ iff $(p = q \wedge E_1 = E_2)$).

For example we might have that $p(X,Y)||\{(p,1)\} = p'(X)$ together with $p(X,Y)||\{(p,2)\} = p''(Y)$, thereby avoiding the name clash that occurs when using the old scheme of erasing.

Algorithm 2 (poly-variant RAF)

Input: a program P, an initial erasure E_p for a particular predicate p.
Output: a new program P' which can be called with $\leftarrow p(t_1, \ldots, t_n)||E_p$ and which is correct[5] if E_p is safe for $\leftarrow p(t_1, \ldots, t_n)$.
Initialisation: $New := \{(E_p, p)\}$, $S := \emptyset$, $P' = \emptyset$;
while not $New \subseteq S$ **do**
  let $S := S \cup New$, $S' := New \setminus S$ and $New := \emptyset$
  for every element (E_p, p) of S' **do**
    for every clause $H \leftarrow A_1, \ldots, A_n \in Def(p, P)$
      let $E_{A_i} = \{(q_i, k) \mid A_i = q_i(t_1, \ldots, t_m) \wedge 1 \leq k \leq m \wedge (q_i, k)$ satisfies
        1. t_k is a variable X; **and**
        2. X occurs exactly once in $A_1, \ldots, A_n$; **and**
        3. X does not occur in $H||E_p$ $\}$
      let $New := New \cup \{(E_{A_i}, q_i) \mid 1 \leq i \leq n\}$
      let $P' := P' \cup \{H||E_p \leftarrow A_1||E_{A_1}, \ldots, A_n||E_{A_n}\}$
return P'

[5] In the sense of Def. 3, by simply replacing $P|E$ by P' and $|$ by $||$.

Note that, in contrast to mono-variant RAF, in the poly-variant RAF algorithm there is no operation that removes a tuple from the erasure E_p. So one may wonder at how the poly-variant algorithm is able to produce a correct program. Indeed, if an erasure E_p contains the tuple (p, k) this means that this particular version of p will only be called with the k-th argument being an existential variable. So, it is always correct to erase the position k in the head of a clause C for that particular version of p, because no bindings for the body will be generated by the existential variable and because we are not interested in the computed answer bindings for that variable. However the position k in a call to p somewhere else in the program, e.g. in the body of C, might not be existential. But in contrast to the mono-variant RAF algorithm, we do not have to remove the tuple (p, k): we simply generate another version for p where the k-th argument is not existential.

Example 7. Let us trace Algorithm 2 by applying it to the program P of Example 6 above and with the initial erasure $E_p = \{(p, 2)\}$ for the predicate p. For this example we can suppose that p_{E_p} is the predicate symbol p with arity 1 and $p_\emptyset$ is simply p with arity 2.

1. After the first iteration we obtain $New = \{(\emptyset, p), (\{(p, 2)\}, p)\}$, as well as $S = \{(\{(p, 2)\}, p)\}$ and $P' =$
   $p(a) \leftarrow$
   $p(b) \leftarrow$
   $p(X) \leftarrow p(X, Z), p(Z)$
2. After the second iteration we have $New = \{(\emptyset, p)\}$, $S = \{(\{(p, 2)\}, p), (\emptyset, p)\}$, meaning that we have reached the fixpoint. Furthermore P' is now the desired program of Example 6 above, i.e. the following clauses have been added wrt the previous iteration:
   $p(a, b) \leftarrow$
   $p(b, c) \leftarrow$
   $p(X, Y) \leftarrow p(X, Z), p(Z, Y)$

The erasure E_p is safe for e.g. the goal $G = \leftarrow p(a, X)$, and the specialised program P' constructed for E_p is correct for $G||E_p = \leftarrow p(a)$ (in the sense of Def. 3, by simply replacing $P|E$ by P' and $|$ by $||$). For instance $P \cup \{\leftarrow p(a, X)\}$ has the computed answer $\{X/b\}$ with $\theta' = \{X/b\}|_\emptyset = \emptyset$ and indeed $P' \cup \{\leftarrow p(a)\}$ has the computed answer $\emptyset$.

Termination of the algorithm follows from the fact that there are only finitely many erasures for every predicate. The result of Algorithm 2 is identical to the result of Algorithm 1 applied to a suitably duplicated and renamed version of the original program. Hence correctness follows from correctness of Algorithm 1 and of the duplication/renaming phase.

Handling Normal Programs

When treating *normal* logic programs an extra problem arises: erasing an argument in a negative goal might modify the floundering behaviour wrt SLDNF. In

fact, the conditions of safety of Def. 11 or Def. 12 would ensure that the negative call will always flounder! So it does not make sense to remove arguments to negative calls (under the conditions of Def. 11, Def. 12) and in general it would even be incorrect to do so. Take for example the goal $\leftarrow ni$ and program P:

$$\begin{array}{ll} int(0) & \leftarrow \\ int(s(X)) & \leftarrow int(X) \\ ni & \leftarrow \neg int(Z) \\ p(a) & \leftarrow \end{array}$$

By simply ignoring the negation and applying the RAF Algorithm 1 for $E_0 = \{(int, 1)\}$ we obtain $E = E_0$ and the following program $P|E$ which behaves incorrectly for the query $G = \leftarrow ni$ (i.e. $G|E$ fails and thereby falsely asserts that everything is an integer)[6]:

$$\begin{array}{ll} int & \leftarrow \\ int & \leftarrow int \\ ni & \leftarrow \neg int \\ p(a) & \leftarrow \end{array}$$

This problem can be solved by adopting the pragmatic but safe approach of keeping all argument positions for predicates occuring inside negative literals. Hence, for the program P above, we would obtain the correct erasure $E = \emptyset$. This technique was actually used for the benchmark programs with negation of the previous section.

Further Improvements

Finally we mention that, in some cases, the conditions of Def. 12 can be relaxed. For instance, the erasure $\{(p, 1), (q, 1)\}$ is safe for the goal $p(X)$ and program:

$$\begin{array}{ll} p(X) & \leftarrow q(f(X)) \\ q(Z) & \leftarrow \end{array}$$

The reason is that, although the erased argument of $q(f(X))$ is a non-variable, the value is never used. So, whereas the RAF Algorithm 1 detects existential arguments (which might return a computed answer binding), the above is an argument which is non-existential and non-ground but whose value is never used (and for which no computed answer binding will be returned).

Those kind of arguments can be detected by another post-processing phase, executing in a similar fashion as RAF, but using reversed conditions.

Algorithm 3 (FAR)

[6] For instance in the programming language Gödel, the query $\leftarrow ni$ flounders in P while $\leftarrow ni|E = \leftarrow ni$ fails in $P|E$. Note however that in Prolog, with its unsound negation, the query $\leftarrow ni$ fails both in P and $P|E$. So this approach to erasing inside negation is actually sound wrt unsound Prolog.

Input: a program P.
Output: a correct erasure E for P (and any G).
Initialisation: $i := 0$; $E_0 = \top_P$;
while there exists a $(p,k) \in E_i$ and a $p(t_1,\ldots,t_n) \leftarrow B \in Clauses(P)$ such that

1. t_k is not a variable; **or**
2. t_k is a variable that occurs more than once in $p(t_1,\ldots,t_n)$; **or**
3. t_k is a variable that occurs in $B|E_i$

do $E_{i+1} := E_i \setminus \{(p,k)\}$; $i := i+1$;
return E_i

The justifications for the points 1–3 in the FAR algorithm are as follows:

1. If t_k is a non-variable term this means that the value of the argument will be unified with t_k. This might lead to failure or to a computed answer binding being returned. So the value of the argument is used after all and might even be instantiated.
2. If t_k is repeated variable in the head of a clause it will be unified with another argument leading to the same problems as in point 1.
3. If t_k is a variable which occurs in non-erased argument in the body of a clause then it is passed as an argument to another call in which the value might be used after all and even be instantiated.

These conditions guarantee that an erased argument is never inspected or instantiated and is only passed as argument to other calls in positions in which it is neither inspected nor instantiated.

Note that this algorithm looks very similar to the RAF Algorithm 1, except that the roles of the head and body of the clauses have been reversed. This has as consequence that, while RAF detects the arguments which are existential (and in a sense propagates unsafe erasures top-down, i.e. from the head to the body of a clause), FAR detects arguments which are never used (and propagates unsafe erasures bottom-up, i.e. from the body to the head of a clause). Also, because the erasures calculated by this algorithm do *not* change the computed answers, we can safely start the algorithm with the complete erasure $E_0 = \top_P$. It can again be seen that the outcome of the algorithm is unique.

Also note that the two algorithms RAF and FAR cannot be put into one algorithm in a straightforward way, because erasures have different meanings in the two algorithms. We can however get an optimal (mono-variant) result by running sequences of FAR and RAF alternately — until a fix-point is reached (this process is well-founded as only finitely many additional argument positions can be erased). Unfortunately, as the following examples show, one application each of RAF and FAR is not sufficient to get the optimal result.

Example 8. Let P be the following program:

$$\begin{array}{ll} p & \leftarrow q(a,Z) \\ q(X,X) & \leftarrow \end{array}$$

Applying FAR does not give any improvement because of the multiple occurrence of the variable X in the head of the second clause. After RAF we obtain:

$$p \leftarrow q(a)$$
$$q(X) \leftarrow$$

Now applying FAR we get the optimally erased program:

$$p \leftarrow q$$
$$q \leftarrow$$

So in this example the FAR algorithm benefitted from erasure performed by the RAF algorithm. The following example shows that the converse can also hold.

Example 9. Take the following program:

$$p \leftarrow q(X, X)$$
$$q(a, Z) \leftarrow$$

Applying RAF does not give any improvement because of the multiple occurrence of the variable X (but this time inside a call and not as in the Example 8 above inside the head). However applying FAR gives the following:

$$p \leftarrow q(X)$$
$$q(a) \leftarrow$$

And now RAF can give an improvement, leading to the optimal program:

$$p \leftarrow q$$
$$q \leftarrow$$

The reason that each of the algorithms can improve the result of the other is that RAF cannot erase multiply occurring variables in the body while FAR cannot erase multiply occurring variables in the head. So, one can easily extend Examples 8 and 9 so that a sequence of applications of RAF and FAR is required for the optimal result. We have not yet examined whether the RAF and FAR algorithm can be combined in a more refined way, e.g. obtaining the optimal program in one pass and maybe also weakening the respective safety conditions by using information provided by the other algorithm.

Poly-variance for FAR

The RAF algorithm looks at *every call* to a predicate p to decide which arguments can be erased. Therefore, the poly-variant extension was based on producing specialised (but still safe) erasures for *every distinct use* of the predicate p. The FAR algorithm however looks at *every head of a clause* defining p to decide which arguments can be erased. This means that an argument might be erasable wrt one clause while not wrt another. We clearly cannot come up with a poly-variant extension of FAR by generating different erasures for every clause.

But one could imagine detecting for every call the clauses that match this call and then derive different erased versions of the same predicate. In the context of optimising residual programs produced by (conjunctive) partial deduction this does not seem to be very interesting. Indeed, every call will usually match every clause of the specialised predicate (especially for partial deduction methods which preserve characteristic trees like [15, 16]).

Negation and FAR

In contrast to RAF, the erasures obtained by FAR *can* be applied inside negative calls. The conditions of the algorithm ensure that any erased variable never returns any interesting[7] computed binding. Therefore removing such arguments, in other words allowing the selection of negative goals even for the case that this argument is non-ground, is correct wrt the completion semantics by correctness of the weaker safeness conditions for SLDNF (see page 94 of [19]). Take for example the following program P:

$$r(X) \leftarrow \neg p(X)$$
$$p(X) \leftarrow q(f(X))$$
$$q(Z) \leftarrow$$

By ignoring the negation and applying the FAR algorithm, we get the erasure $E = \{(q,1),(p,1),(r,1)\}$ and thus $P|E$:

$$r \leftarrow \neg p$$
$$p \leftarrow q$$
$$q \leftarrow$$

Using $P|E \cup \{G|E\}$ instead of $P \cup \{G\}$ is correct. In addition $P|E \cup \{G|E\}$ will never flounder when using standard SLDNF, while P will flounder for any query $G = \leftarrow r(t)$ for which t is not ground. In other words, the FAR algorithm not only improves the efficiency of a program, but also its "floundering behaviour" under standard SLDNF.

Implementation

The FAR algorithm has also been implemented (by slightly re-writing the RAF algorithm) and incorporated into the ECCE system [18]. Preliminary experiments indicate that, when executed once after RAF, it is able to remove redundant arguments much less often than RAF, although in some cases it can be highly beneficial (e.g. bringing execution time of the final specialised program from 6.3 s down to 4.1 s for the memo-solve example of the DPPD library [18]). Also, it seems that an optimisation similar to FAR has recently been added to the Mercury compiler, where it is e.g. useful to get rid of arguments carrying unused type information.

[7] An erased variable V might only return bindings of the form V/F where F is a fresh existential variable.

6 Related Work and Conclusion

It would seem that our algorithm RAF for removal of redundant arguments is related to Proietti and Pettorossi's *Elimination Procedure* (EP) for removal of *unnecessary variables*. However, it would be a mistake to compare RAF and EP directly. RAF is intended as a simple, efficient post-processing phase for program transformers, in particular for conjunctive partial deduction, whereas EP is a less efficient, but far more powerful unfold/fold-based transformation which can remove intermediate data structures from programs. For instance, it can transform the naive double-append program to the version with $da'/4$ directly. Thus one should rather compare EP to the composition of conjunctive partial deduction with RAF.

The work in [17], among other work, helped to bridge the gap between the partial deduction and the unfold/fold areas. Roughly, the proofs in [17] show that, for every conjunctive partial deduction program specialisation, there exists an equivalent transformation sequence consisting of Tamaki-Sato definition steps and unfolding steps, followed by Tamaki-Sato folding steps. There are however some subtle differences between control in conjunctive partial deduction and control in the unfold/fold approach. Indeed, an unfold/fold transformation is usually described as doing the definition steps first (and at that point one knows which arguments are existential because existentiality can be propagated top-down — but one does not yet have the whole specialised program available) while conjunctive partial deduction can be seen as doing the definition introduction and the corresponding folding steps only at the very end (when producing the residual code). Therefore the use of an algorithm like RAF is required for conjunctive partial deduction to detect the existential variables for the respective definitions. But on the other hand this also gives conjunctive partial deduction the possibility to base its choice on the *entire* residual program. For instance one may use a mono-variant algorithm (to limit the code size) or an algorithm like FAR which, due to its bottom-up nature, has to examine the entire residual program.

Another related work is [4], which provides some pragmatics for removing unnecessary variables in the context of optimising binarized Prolog programs.

Yet another related work is that on *slicing* [29], useful in the context of debugging. RAF can also be used to perform a simple form of program slicing; for instance, one can use RAF to find the sub-part of a program which affects a certain argument. However, the slice so obtained is usually less precise than the one obtained by the specific slicing algorithm of [29] which takes Prolog's left-to-right execution strategy into account and performs a data-dependency analysis.

Similar work has also been considered in other settings than logic programming. Conventional compiler optimisations use data-flow analyses to detect and remove *dead code*, i.e. commands that can never be reached and assignments to variables whose values are not subsequently required, see [1]. These two forms of redundancy are similar to useless clauses and redundant variables.

Such techniques have also appeared in functional programming. Chin [3] describes a technique to remove *useless variables*, using an abstract interpretation (forwards analysis). A concrete program is translated into an abstract one working on a two-point domain. The least fix-point of the abstract program is computed, and from this an approximation of the set of useless variables can be derived.

Hughes [12] describes a backwards analysis for *strictness analysis*. Such analyses give for each parameter of a function the information either that the parameter *perhaps is not* used, or that the parameter *definitely is* used. The analysis in [12] can in addition give the information that a parameter *definitely is not* used, in which case it can be erased from the program.

Another technique can be based on Seidl's work [30]. He shows that the corresponding question for higher-level grammars, *parameter-reducedness*, is decidable. The idea then is to approximate a functional program by means of a higher-level grammar, and decide parameter-reducedness on the grammar.

Most work on program slicing has been done on imperative programs [33]. Reps [28] describes program slicing for functional programs as a backwards transformation.

Compared to all these techniques our algorithm is strikingly simple, very efficient, and easy to prove correct. The obvious drawback of our technique is that it is less precise. Nevertheless, the benchmarks show that our algorithm performs well on a range of mechanically generated programs, indicating a good trade-off between complexity and precision.

Acknowledgements. The work reported in this paper grew out of joint work with D. De Schreye, R. Glück, J. Jørgensen, B. Martens, and A. de Waal, to whom we are grateful for discussions. D. De Schreye, J. Jørgensen, B. Martens and anonymous referees provided valuable comments on a draft of the paper. We are also indebted to T. Reps for discussions on slicing, J. Hric and J. Gallagher for comments which led to the FAR algorithm, F. Henderson and Z. Somogyi for discussions on Mercury, as well as A. Pettorossi and M. Proietti for insightful discussions on the relation between this work and the unfold/fold approach.

References

1. A. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Design.* Addison-Wesley, 1986.
2. K. Benkerimi and P. M. Hill. Supporting transformations for the partial evaluation of logic programs. *Journal of Logic and Computation*, 3(5):469–486, October 1993.
3. W.-N. Chin. *Automatic Methods for Program Transformation.* PhD thesis, Imperial College, University of London, 1990.
4. B. Demoen. On the transformation of a Prolog program to a more efficient binary program. In K.-K. Lau and T.P. Clement, editors, *Logic Program Synthesis and Transformation.* Proceedings of LOPSTR'92, pages 242–252. Springer Verlag.
5. D.A. de Waal. Analysis and Transformation of Proof Procedures. PhD Thesis, Department of Computer Science, University of Bristol, 1992.

6. D.A. de Waal and J. Gallagher. The applicability of logic program analysis and transformation to Theorem Proving. In A. Bundy, editor, *Automated Deduction—CADE-12*, pages 207–221, 1994. Springer Verlag.
7. J. Gallagher. Tutorial on specialisation of logic programs. In *Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 88–98. ACM Press, 1993.
8. J. Gallagher and M. Bruynooghe. Some low-level transformations for logic programs. In M. Bruynooghe, editor, *Proceedings of Meta90 Workshop on Meta Programming in Logic*, pages 229–244, Leuven, Belgium, 1990.
9. J. Gallagher and D.A. de Waal. Deletion of redundant unary type predicates from logic programs. In K.-K. Lau and T.P. Clement, editors, *Logic Program Synthesis and Transformation.* Proceedings of LOPSTR'92, pages 151–167. Springer Verlag.
10. R. Glück and M.H. Sørensen. Partial deduction and driving are equivalent. In M. Hermenegildo and J. Penjam, editors, *Programming Language Implementation and Logic Programming. Proceedings*, *Lecture Notes in Computer Science* 844, pages 165–181, Madrid, Spain, 1994. Springer-Verlag.
11. R. Glück, J. Jørgensen, B. Martens, and M.H. Sørensen. Controlling conjunctive partial deduction of definite logic programs. In H. Kuchen and S.D.. Swierstra, editors, *Programming Language Implementation and Logic Programming. Proceedings*, *Lecture Notes in Computer Science* 1140, pages 152–166, Aachen, Germany, 1996. Springer-Verlag.
12. J. Hughes. Backwards analysis offunctional programs. In D. Bjørner, A.P. Ershov, and N.D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 187–208, Amsterdam, 1988. North-Holland.
13. J. Jørgensen, M. Leuschel and B. Martens. Conjunctive partial deduction in practice. In J. Gallagher, editor, *Pre-Proceedings of LOPSTR'96*, pages 46–62, Stockholm, Sweden, August 1996. Extended version as Technical Report CW 242, Katholieke Universiteit Leuven, 1996.
14. J. Komorowski. Partial evaluation as a means for inferencing data structures in an applicative language: A theory and implementation in the case of Prolog. In *9th ACM Symposium on Principles of Programming Languages*, pages 255–167. ACM Press, 1982.
15. M. Leuschel. Ecological partial deduction: Preserving characteristic trees without constraints. In M. Proietti, editor, *Proceedings of LOPSTR'95*, *Lecture Notes in Computer Science* 1048, pages 1–16, 1996. Springer-Verlag.
16. M. Leuschel and B. Martens. Global control for partial deduction through characteristic atoms and global trees. In O. Danvy, R. Glück, and P. Thiemann, editors, *Proceedings Dagstuhl Seminar on Partial Evaluation*, *Lecture Notes in Computer Science* 1110, pages 263–283, Schloss Dagstuhl, Germany, February 1996. Springer-Verlag.
17. 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, *Proceedings of the Joint International Conference and Symposium on Logic Programming, JICSLP'96*, pages 319–332, Bonn, Germany, September 1996. MIT Press.
18. M. Leuschel. The ECCE partial deduction system and the DPPD library of benchmarks. Accessible via `http://www.cs.kuleuven.ac.be/~lpai`.
19. J.W. Lloyd. *Foundations of Logic Programming*, North-Holland, New York, 1987.
20. J.W. Lloyd and J.C. Shepherdson. Partial evaluation in logic programming. *Journal of Logic Programming*, 11(3-4):217–242, 1991.

21. J.W. Lloyd, editor. *Logic Programming: Proceedings of the 1995 International Symposium*. MIT Press, 1995.
22. K. Marriott, L. Naish, and J.-L. Lassez. Most specific logic programs. In *Proceedings of the Joint International Conference and Symposium on Logic Programming*, Seattle, 1988. IEEE, MIT Press.
23. B. Martens. *On the Semantics of Meta-Programming and the Control of Partial Deduction in Logic programming*. PhD thesis, Katholieke Universiteit Leuven, 1994.
24. B. Martens and J. Gallagher. Ensuring global termination of partial deduction while allowing flexible polyvariance. In L. Sterling, editor, *Proceedings ICLP'95*, pages 597–611, Shonan Village Center, Kanagawa, Japan, June 1995. MIT Press.
25. A. Pettorossi and M. Proietti. Transformation of logic programs: Foundations and techniques. *Journal of Logic Programming*, 19 & 20:261–320, 1994.
26. M. Proietti and A. Pettorossi. Unfolding – definition – folding, in this order for avoiding unnecessary variables in logic programs. In *Programming Language Implementation and Logic Programming*, *Lecture Notes in Computer Science* 528, pages 347–358. Springer-Verlag, 1991.
27. M. Proietti and A. Pettorossi. The loop absorption and the generalization strategies for the development of logic programs and partial deduction. *Journal of Logic Programming*, 16:123–161, 1993.
28. T. Reps. Program specialization via program slicing. In O. Danvy, R. Glück, and P. Thiemann, editors, *Proceedings Dagstuhl Seminar on Partial Evaluation*, pages 409–429, Schloss Dagstuhl, Germany, 1996. Springer-Verlag.
29. S. Schoenig and M. Ducassé. A hybrid backward slicing algorithm producing executable slices for Prolog. In *Proceedings of the 7th Workshop on Logic Programming Environments*, pages 41–48, Portland, USA, December 1995.
30. H. Seidl. Parameter-reduction of higher-level grammars. *Theoretical Computer Science*, 55:47–85, 1987.
31. M.H. Sørensen and R. Glück. An algorithm of generalization in positive supercompilation. In [21], pages 465–479.
32. M.H. Sørensen, R. Glück, and N.D. Jones. Towards unifying deforestation, supercompilation, partial evaluation, and generalized partial computation. In D. Sannella, editor, *Programming Languages and Systems*, *Lecture Notes in Computer Science* 788, pages 485–500. Springer-Verlag, 1994.
33. F. Tip. A survey of program slicing techniques. *Journal of Programming Languages* 3:121–181, 1995.
34. V.F. Turchin. The algorithm of generalization in the supercompiler. In D. Bjørner, A.P. Ershov, and N.D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 531–549. North-Holland, 1988.
35. P.L. Wadler. Deforestation: Transforming programs to eliminate intermediate trees. *Theoretical Computer Science*, 73:231–248, 1990. Preliminary version in ESOP'88, LNCS vol. 300.

This article was processed using the LaTeX macro package with LLNCS style