Generating Efficient, Terminating Logic Programs

Jonathan C. Martin[1] and Andy King[2]

[1] Department of Electronics and Computer Science, University of Southampton, Southampton, SO9 5NH, UK. jcm93r@ecs.soton.ac.uk
[2] Computing Laboratory, University of Kent at Canterbury, Canterbury, CT2 7NF, UK. a.m.king@ukc.ac.uk

Abstract. The objective of control generation in logic programming is to automatically derive a computation rule for a program that is efficient and yet does not compromise program correctness. Progress in solving this important problem has been slow and, to date, only partial solutions have been proposed where the generated programs are either incorrect or inefficient. We show how the control generation problem can be tackled with a simple automatic transformation that relies on information about the depths of SLD-trees. To prove correctness of our transform we introduce the notion of a semi delay recurrent program which generalises previous ideas in the termination literature for reasoning about logic programs with dynamic selection rules.

1 Introduction

A logic program can be considered as consisting of a logic component and a control component [8]. Although the meaning of the program is largely defined by its logical specification, choosing the right control mechanism is crucial in obtaining a correct and efficient program. In recent years, one of the most popular ways of defining control is via suspension mechanisms which delay the selection of an atom in a goal until some condition is satisfied. Such mechanisms include the block declarations of SICStus Prolog [7] and the DELAY declarations of Gödel [6]. These mechanisms are used to define dynamic selection rules with the two main aims of enhancing performance through coroutining and ensuring termination. In practise, however, these two aims are not complementary and it is often the case that termination, and hence program correctness, is sacrificed for efficiency.

Consider, for instance, the Append program given below (in Gödel style syntax) with its standard DELAY declaration which delays the selection of an Append/3 atom until either the first or third argument is instantiated to a non-variable term.

```
Append([], x, x).
Append([u|x], y, [u|z]) ← Append(x, y, z).

DELAY Append(x, _, z) UNTIL Nonvar(x) ∨ Nonvar(z).
```

Interestingly, although it is intended to assist termination the delay declaration is not sufficient to ensure that *all* Append/3 goals terminate. The goal

$\leftarrow$ Append([x|xs], ys, xs), for example, satisfies the condition in the declaration and yet is non-terminating [14].

Termination can only be guaranteed by strengthening the condition in the delay declaration. This is where the trade off between efficiency, termination and completeness takes place. The stronger the condition, the more goals suspend. Although termination may eventually be assured, it may be at the expense of not resolving goals which have finite derivations. Also the stronger the delay condition, the more time consuming it usually is to check. Thus one of the main problems in generating control of this form is finding suitable conditions which are inexpensive to check and guarantee termination and completeness. We will refer to this as the local termination issue, to contrast it with another global aspect of the termination problem which we will examine shortly.

1.1 Local Termination

There have been several attempts at solving the local termination problem. We will examine each of these in the context of the Append program above, though each technique has wider applicability.

Linearity In the case of single literal goals, one additional condition sufficient for termination is that the goal is linear, that is, no variable occurs more than once in the goal [10]. Although this restriction would prevent the looping Append/3 call above from proceeding, it would also unfortunately delay many other goals with finite derivations such as $\leftarrow$ Append([x, x], ys, zs). In addition, the time complexity for checking linearity is quadratic in the number of variables in the goal.

Rigidity An alternative approach by Marchiori and Teusink [11] and Mesnard [13] proposes delaying Append/3 goals until the first or third argument is a list of determinate length. Termination is obtained for a large class of goals, but at a price. Checking such a condition requires the complete traversal of the list and the condition must be checked on *every* call to the predicate[3]. Naish argues that this approach can be "... expensive to implement and ... can delay the detection of failure in a sequential system and restrict parallelism in a stream and-parallel system" [14].

Modes Naish goes on to solve the problem with the use of modes. Termination can be guaranteed with the above DELAY declaration if the modes of the Append/3 calls are *acyclic*, or more generally *cycle bounded* [14]. This restriction essentially stops the output feeding back into the input. Although modes form a good basis for solving the local termination problem, they have not been shown to be satisfactory for reasoning about another termination problem, that of speculative output bindings.

[3] In [13] the check is, in fact, only performed on the initial call, but there is no justification for this optimisation given in the paper. For non-structurally recursive predicates, e.g. Quicksort/2 of Sect. 1.2, such an optimisation is usually not possible.

1.2 Global Termination

Even when finite derivations exist, delay conditions alone are not, in general, sufficient to ensure termination. Infinite computations may arise as a result of speculative output bindings [14], which can occur due to the dynamic selection of atoms. There are several problems associated with speculative output bindings (see [14] for a discussion of these). Here we are only interested in the affect that they have on termination, and this is what we call the global termination issue. To illustrate the problem caused by speculative output bindings consider the Quicksort program shown below. This is an example of a well known program whose termination behaviour can be unsatisfactory. With the given delay declarations, the program can be shown to terminate in forward mode, that is for queries of the form ← Quicksort(x, y) where x is bound and y is uninstantiated. In reverse mode, however, where y is bound and x is uninstantiated, the program does not always terminate. More precisely, a query such as ← Quicksort(x, [1,2,3]) will terminate *existentially*, i.e. produce a solution, but not *universally*, i.e. produce all solutions. In fact, experimentation with the Gödel and SICStus implementations indicates that when the list of elements is not strictly increasing, e.g. in ← Quicksort(x, [1,1]) and ← Quicksort(x, [2,1]), the program does not even existentially terminate! This is illustrative of the subtle problems that dynamic selection rules pose in reasoning about termination, and which suggest that control should ideally be automated to avoid them.

```
Quicksort([], []).
Quicksort([x|xs], ys) ←  Partition(xs, x, l, b) ∧ Quicksort(l, ls) ∧
                         Quicksort(b, bs) ∧ Append(ls, [x|bs], ys).

DELAY Quicksort(x, y) UNTIL Nonvar(x) ∨ Nonvar(y).

Partition([], _, [], []).
Partition([x|xs], y, [x|ls], bs) ← x ≤ y ∧ Partition(xs, y, ls, bs).
Partition([x|xs], y, ls, [x|bs]) ← x > y ∧ Partition(xs, y, ls, bs).

DELAY Partition(x, _, y, z) UNTIL Nonvar(x) ∨ (Nonvar(y) ∧ Nonvar(z)).
```

To improve matters, the delay conditions can be strengthened in the manner prescribed by Naish or Marchiori and Teusink. In general, however, no matter how strong the delay conditions are, they are not always sufficient to ensure termination, even though a terminating computation exists. To see why, consider augmenting the Quicksort program with the clause

```
Append(x, [_|x], x) ← False.
```

The declarative semantics of the program are completely unchanged by the addition of this clause and one would hope that the new program would produce exactly the same set of answers as the original. This will not be the case, however, if this clause is selected before all other Append/3 clauses. Consider the query ← Quicksort(x, [1,2,3]). Following resolution with the second clause of Quicksort/2, the only atom which can be selected is Append(ls, [x|bs], [1,2,3]). When this unifies with the above clause, both ls and bs are immediately bound to the term [1,2,3]. As a result of these speculative output bindings the previously suspended calls Quicksort(l, ls) and Quicksort(b, bs) will be woken *before* the computation reaches the call to False. The net result is an infinite computation due to recurring goals of the form $\leftarrow$ Quicksort(x, [1,2,3]).

The problem here is that the output bindings are made before it is known that the goal will fail and no matter how stringent the conditions are on the Quicksort/2 goals, loops of this kind cannot generally be avoided. The reason for this is that a delay condition only measures a local property of a goal without regard for the computation as a whole. The conditions can ensure that goals are bounded, but are unable to ensure that the bounds are decreasing.

Local Computation Rule To remedy this, Marchiori and Teusink [11] propose the use of a *local computation rule*. Such a rule only selects atoms from those that are most recently introduced in a derivation. This ensures that any atom selected from a goal, is completely resolved before any other atom in the goal is selected. The effect in the above example is that the call to False would be selected and the Append/3 goal fully resolved before the calls to Quicksort/2 are woken. This prevents an infinite loop. The main disadvantage of local computation rules is that they do not allow any form of coroutining. This is clearly a very severe restriction.

Delayed Output Unification A similar solution proposed by Naish [14] is that of delaying output unification. In the example above, assuming a left-to-right computation rule, the extra Append/3 clause would be rewritten as

$$\mathsf{Append(x, y, z)} \leftarrow \mathsf{False} \wedge \mathsf{y} = [\_|\mathsf{x}] \wedge \mathsf{z} = \mathsf{x}.$$

The intended effect of such a transformation is that no output bindings should be made until the computation is known to succeed. This has parallels with the local computation rule and also restricts coroutining.

Constraints Mesnard uses interargument relationships compiled as constraints to guarantee that the bounds on goals decrease [13]. For example, solving the constraint $|\mathsf{ys}|_{\mathsf{length}} = |\mathsf{ls}|_{\mathsf{length}} + 1 + |\mathsf{bs}|_{\mathsf{length}}$ before selecting the atom Append(ls, [x|bs], ys) ensures that bs and ls are only bound to lists with lengths less than that of ys. This is enough to guarantee termination, but is expensive to check as it requires calculating the lengths of all three arguments of Append/3.

1.3 Our Contribution

In summary, we see that the most promising approaches to control generation, while guaranteeing termination and completeness, produce programs which are inefficient, either directly due to expensive checks which must be performed at run-time or indirectly by restricting coroutining.

In this paper we present an elegant solution to the above problems. To solve the local termination problem, we use delay declarations in the spirit of [11] combined with a novel program transformation which overcomes the inefficiencies of their approach. Simultaneously, the transformation inexpensively solves the global termination problem *without* grossly restricting coroutining. The transformation is simple and is easy to automate. Transformed programs are guaranteed to terminate and are also efficient. Hence, the technique forms a sound basis for automatically generating efficient, terminating logic programs from logical specifications.

The technique is based on the following idea. If the maximum depth of the SLD-tree needed to solve a given query can be determined, then by only searching to that depth the query will be completely solved, i.e. *all* answers (if any) will be obtained, in a finite number of steps. We first present the technique through an example. Then we formalise the transformation and prove termination for the transformed programs.

2 Example

We demonstrate our program transformation on the Quicksort program from the introduction. The transformed program is shown below. Termination is guaranteed for all queries ← Quicksort(x, y). Furthermore when x or y is a ground list of integers, the computation does not flounder and if it succeeds then the set of answers produced is complete with respect to the declarative semantics.

```
Quicksort(x, y) ← SetDepth_Q(x, y, d) ∧ Quicksort_1(x, y, d).

DELAY Quicksort_1(_, _, d) UNTIL Ground(d).

Quicksort_1(x, y, d) ← Quicksort_2(x, y, d).

Quicksort_2([], [], d) ← d ≥ 0.
Quicksort_2([x|xs], ys, d + 1) ←  d ≥ 0 ∧ Partition(xs, x, l, b) ∧ Quicksort_2(l, ls, d) ∧
                                  Quicksort_2(b, bs, d) ∧ Append(ls, [x|bs], ys).

Partition(xs, x, l, b) ← SetDepth_P(xs, l, b, d) ∧ Partition_1(xs, x, l, b, d).

DELAY Partition_1(_, _, _, _, d) UNTIL Ground(d).

Partition_1(xs, x, l, b, d) ← Partition_2(xs, x, l, b, d).

Partition_2([], _, [], [], d) ← d ≥ 0.
Partition_2([x|xs], y, [x|ls], bs, d + 1) ← d ≥ 0 ∧ x ≤ y ∧ Partition_2(xs, y, ls, bs, d).
Partition_2([x|xs], y, ls, [x|bs], d + 1) ← d ≥ 0 ∧ x > y ∧ Partition_2(xs, y, ls, bs, d).

Append(x, y, z) ← SetDepth_A(x, z, d) ∧ Append_1(x, y, z, d).

DELAY Append_1(_, _, _, d) UNTIL Ground(d).

Append_1(x, y, z, d) ← Append_2(x, y, z, d).

Append_2([], x, x, d) ← d ≥ 0.
Append_2([u|x], y, [u|z], d + 1) ← d ≥ 0 ∧ Append_2(x, y, z, d).
```

The predicate SetDepth_Q(x, y, d) calculates the lengths of the lists x and y, delaying until one of the lists is found to be of determinate length, at which point the variable d is instantiated to this length. Only then can the call to Quicksort_1/3 proceed. The purpose of this last argument is to ensure finiteness of the subsequent computation. More precisely, d is an upper bound on the number of calls to the recursive clause of Quicksort_2/3 *in any successful derivation*. Thus by failing any derivation where the number of such calls has exceeded this bound (using the constraint $d \geq 0$), termination is guaranteed without losing completeness. The predicates SetDepth_P/4 and SetDepth_A/3 are defined in a similar way.

2.1 Local and Global Control

The local control problem is solved in the first instance with a rigidity check in the style of [11]. This ensures that the initial goal is bounded. Boundedness of subsequent goals, however, is enforced by the depth parameter and further rigidity checks on these depth bounded goals are redundant. This allows, for example, the call Quicksort_2(l, ls, d) to proceed, without fear of an infinite computation, even if both l and ls are uninstantiated, providing d is ground. A huge improvement in performance is possible by eliminating these checks. The global control problem is also neatly solved. By restricting the search space to be finite, even though speculative output bindings may still occur, they cannot lead to infinite derivations.

2.2 A Simple Optimisation

Even though many of the rigidity checks have now been removed, the efficiency of the program is still unsatisfactory. This is due to the rigidity checks which are performed on each call to Append/3 and Partition/4. It is easy to show that the depths of these subcomputations are bounded by the same depth parameter occurring in Quicksort_2/3. Hence, we can replace the atoms Partition(xs, x, l, b) and Append(ls, [x|bs], ys) in the body of Quicksort_2/3 with the atoms Partition_2(xs, x, l, b, d) and Append_2(ls, [x|bs], ys, d).

This optimised version of the program is quite efficient. The only rigidity checks that are performed are those on the initial input, exactly at the point where they are needed to guarantee termination. Following the initial call to Quicksort_2/3 the program runs completely without delays and the only other overhead is the decrementation of the depth parameter and some trivial boundedness checks. The net result is that, with the Bristol Gödel implementation, the program actually runs faster on average than the original program with the Nonvar delay declarations!

2.3 Coroutining

Notice in particular how the global termination problem is overcome without reducing the potential for coroutining. Simply knowing the maximum depth of any potentially successful branch of the SLD-tree allows us to force any derivations along this branch which extend beyond this depth to fail without losing completeness. These forced failures keep the computation tree finite but do not restrict the way in which the tree is searched. The addition of the failing Append/3 clause from the introduction (which would appear here as an Append_2/3 clause) cannot effect the termination of the algorithm, even if the same coroutining behaviour of the original program is used. Of course, we need to constrain the computation rule such that

1. the test $\mathsf{d} \geq 0$ is always selected before any other atoms in the body of the clause with a subterm d, and
2. the depth parameter is ground on each recursive call (or for any atom with a subterm d in the optimised version)

but this is not nearly as restrictive as using the local computation rule. Indeed, using the standard left-to-right selection rule these conditions will clearly be satisfied in the above program.

2.4 Termination and Efficiency

With termination guaranteed, the programmer is now free to concentrate on the program's performance. Notice for the program above that the order of the goals in the body of Quicksort_2 is critical to the efficiency of the algorithm. For the best performance, they must be arranged so that the computation is data driven. In fact, by defining SetDepth_Q/3 by

```
SetDepth_Q(x, y, d) ←
    Length(x, d) ∧
    Length(y, d).
```

the computation will be data driven in both forward and reverse modes with the ordering of the goals as above. This dependence on the ordering can be reduced by introducing the typical delay declarations used for this program. These declarations do *not* effect the terminating nature of the algorithm, in that they will not cause the algorithm to loop, though they may possibly reduce previously successful or failing derivations to floundering ones. They are inserted solely to improve the performance through coroutining. Alternatively, one may seek to optimise the performance for different modes through multiple specialisation, for example. The important point is that with this approach the trade-off between termination and performance is significantly reduced. In seeking an efficient algorithm, correctness does not have to be compromised.

3 Preliminaries

Terms, atoms and formulae are defined in the usual way [9]. A program P is a set of clauses of the form $\forall(a \leftarrow w)$ where a is an atom and w is either absent or a conjunction of atoms. We denote by $body(a \leftarrow w)$ the set of atoms appearing in w. Given a program P, then Σ_P denotes the alphabet of predicate symbols in P. We denote by $var(o)$ the set of variables in a syntactic object o. A grounding substitution for a syntactic object o is a substitution in which each variable in o is bound to a ground term. We denote by $rel(A)$ the predicate symbol of the atom A. We denote a tuple of elements $\langle d_1, \ldots, d_n \rangle$ by $\overline{d}$ and write $d_i \in \overline{d}$ if d_i is the ith element of the tuple $\overline{d}$. If the atom $p(t_1, \ldots, t_n)$ is denoted by $p(\overline{t})$, then the atom $p(t_1, \ldots, t_n, d)$ is denoted by $p(\overline{t}, d)$. Finally, we denote the minimal model of a program P by $M(P)$.

4 The Transformation

Our aim is to develop a program transformation which is able to derive correct and efficient programs from logical specifications. We divide the development into three stages where we consider termination, completeness and efficiency respectively.

4.1 Termination

To prove termination of the transformed programs we will need to introduce a new program class which subsumes that of delay recurrent programs introduced in [11]. Its introduction is motivated by an overly restrictive condition imposed in the definition of delay recurrency. By removing this unnecessary condition we obtain the new class of programs which we call semi delay recurrent. Our transformed programs will lie within this class. The following notions, due to Bezem [5], will be needed.

Definition 1 level mapping [5]. Let P be a program. A *level mapping* for P is a function $|.| : B_P \mapsto \mathbb{N}$ from the Herbrand base to the natural numbers. $\square$

A level mapping is only defined for ground atoms. The next definition lifts the mapping to non-ground atoms and goals.

Definition 2 bounded atom and goal [5]. An atom A is *bounded* wrt a level mapping $|.|$ if $|.|$ is bounded on the set $[A]$ of variable free instances of A. If A is bounded then $|[A]|$ denotes the maximum that $|.|$ takes on A. A goal $G = \leftarrow A_1, \ldots, A_n$ is *bounded* if every A_i is bounded. If G is bounded then $|[G]|$ denotes the (finite) multiset consisting of the natural numbers $|[A_1]|, \ldots, |[A_n]|$. $\square$

Level mappings are used to prove termination in the following way. Let $G = G_0, G_1, G_2, \ldots$ be the goals in a refutation of G and $|.|$ a level mapping. Given that G is bounded wrt $|.|$ and $|[G_i]| > |[G_{i+1}]|$ for all i, we can deduce that the sequence $G = G_0, G_1, G_2, \ldots$ is finite by the well-foundedness of the natural numbers. To prove the goal ordering property, that $|[G_i]| > |[G_{i+1}]|$ for all i and for all possible refutations of G, one must examine the clauses and the computation rule used. Various classes of program have been identified, where this property is satisfied for a given computation rule [1, 2, 5, 11]. Bezem, for example, introduced the class of recurrent programs [5], where the goal ordering property is always satisfied, regardless of the computation rule.

Definition 3 recurrency [5]. Let P be a definite logic program and $|.|$ a level mapping for P. A clause $H \leftarrow B_1, \ldots, B_n$ is recurrent (wrt $|.|$) if for every grounding substitution θ, $|H\theta| > |B_i\theta|$ for all $i \in [1, n]$. P is recurrent (wrt $|.|$) if every clause in P is recurrent (wrt $|.|$). □

One problem with recurrency, as noted in [3], is that it does not intuitively relate to the principal cause of non-termination in a logic program – recursion. The definition requires that level mappings decrease from clause heads to clause bodies irrespective of the recursive relation between the two. This relation is formalised in the following definition.

Definition 4 predicate dependency. Given Σ_p defined by a program P, we say that $p \in \Sigma_p$ *directly depends on* $q \in \Sigma_p$ if there is a statement in P with head $p(\overline{t})$ and a body atom $q(\overline{t'})$. The *depends on* relation is defined as the reflexive, transitive closure of the directly depends on relation. p and q are *mutually dependent*, written $p \simeq q$, if p depends on q and q depends on p. □

Notice that there is a well-founded ordering among the predicates of a program induced by the depends on relation. We write $p \sqsupset q$ whenever p depends on q but q does not depend on p, i.e. p calls q as a subprogram. By abuse of terminology we will say that two atoms are mutually dependent (with each other) if they have mutually dependent predicate symbols.

Apt and Pedreschi [3] observed that while it is necessary for the level mapping to decrease between the head $p(\overline{t})$ of a clause and each body atom $q(\overline{t'})$ with $p \simeq q$, a strict decrease is not required for the other atoms in the body. They introduced the notion of semi-recurrent program which exploited this observation. Their definition still insisted, however, that the level of the head was at least greater or equal to the level of all body atoms, whereas in fact it does not matter if the level of non-mutually dependent atoms is greater than in the head provided that these atoms are bounded whenever they are selected.

Marchiori and Teusink [11] noticed that boundedness of atoms could be enforced by using delay declarations but did not fully exploit this fact combined with the above observation in defining delay recurrency, a version of recurrency for programs using dynamic selection rules. Their definition required a decrease in the level mapping from the head to the non-mutually dependent atoms when in fact boundedness was already guaranteed by the delay declarations.

We generalise their definition here by removing this restriction. The new definition will prove useful for defining a large class of terminating programs which permit coroutining. We first need the following two definitions from [11].

Definition 5 direct cover [11]. Let $|.|$ be a level mapping and $c : H \leftarrow B$ a clause. Let $A \in body(c)$ and $C \subset body(c)$ such that $A \notin C$. Then C is a *direct cover* for A wrt $|.|$ in c, if there exists a substitution θ such that $A\theta$ is bounded wrt $|.|$ and $dom(\theta) \subseteq var(H, C)$. A direct cover C for A is minimal if no proper subset of C is a direct cover for A. □

Definition 6 cover [11]. Let $|.|$ be a level mapping and $c : H \leftarrow B$ a clause. Let $A \in body(c)$ and $C \subset body(c)$. Then C is a *cover* for A wrt $|.|$ in c, if (A, C) is an element of the least set S such that

1. $(A, \emptyset) \in S$ whenever the empty set is the minimal direct cover for A wrt $|.|$ in c, and
2. $(A, C) \in S$ whenever $A \notin C$, and C is of the form

$$\{A_1, \ldots, A_k\} \cup D_1 \cup \ldots \cup D_k$$

   such that $\{A_1, \ldots, A_k\}$ is a minimal direct cover of A in c, and for $i \in [1, k], (A_i, D_i) \in S$. □

Intuitively, a cover of an atom A in a clause is a subset of the body atoms which must be (partially) resolved in order for A to become bounded wrt some level mapping. Where possible, we will assume in the following that the level mapping is fixed for a given program. The following definition generalises that of a delay recurrent program in [11].

Definition 7 semi delay recurrency. Let $|.|$ be a level mapping and I an interpretation for a program P. A clause $c : H \leftarrow B_1, \ldots, B_n$. is *semi delay recurrent* wrt $|.|$ and I if

1. I is a model for c and
2. if $rel(H) \simeq rel(B_i)$, then for every cover C for B_i and for every grounding substitution θ for c such that $I \models C\theta$, we have that $|H\theta| > |B_i\theta|$.

A program P is semi delay recurrent wrt $|.|$ and I if every clause is semi delay recurrent wrt $|.|$ and I. □

Note that delay recurrency is *not* equivalent to semi delay recurrency. Every delay recurrent program is semi delay recurrent, but the converse is not true.

Example 1. The following program is semi delay recurrent, but not delay recurrent.

$\mathsf{P([x|y]) \leftarrow Append(\_, \_, \_) \wedge P(y)}$. □

Due to the possibility of speculative output bindings, in order to be sure that the condition $I \models C\theta$ holds, each atom in C must be completely resolved. In [11] local selection rules are used to ensure this property. A local selection rule only selects the most recently introduced atoms in a derivation and thus completely resolves sub-computations before proceeding with the main computation.

Notice, however, that for semi delay recurrency, it is only necessary for the covers of those atoms which are mutually dependent with the head of the clause to be resolved completely. This means that following the resolution of these covers, an arbitrary amount of coroutining may take place amongst the remaining atoms of the clause. To formalise a selection rule based on this idea we introduce the notion of covers and covered atoms in a goal.

Definition 8 covers and covered atoms in a goal. Let $G = \leftarrow A_1, \ldots, A_n$ be a goal and suppose that the atom A_i is resolved with the semi delay recurrent clause $c : H \leftarrow B$ giving $\theta \in mgu(H, A_i)$. If $A \in body(B)$ and $rel(A) \simeq rel(H)$, then $A\theta$ is a covered atom in G' and $C\theta$ is a cover of $A\theta$ in G' where C is a cover of A in c and G' is the resolvent of G. □

Definition 9 semi local selection rule. A semi local selection rule only selects a covered atom in a goal if one of its covers in a previous goal has been completely resolved. □

A semi local selection rule ensures that before selecting a covered atom A, we first fully resolve a cover of A. Before giving the main result of our construction, we need the following definition taken from [11].

Definition 10 safe delay declaration [11]. A delay declaration for a predicate p is safe wrt $|.|$ if for every atom A with predicate symbol p, if A satisfies its delay declaration, then A is bounded wrt $|.|$. □

Theorem 11. Let P be a program with a delay declaration for each predicate in P. Let $|.|$ be a level mapping and I an interpretation. Suppose that

1. P is semi delay recurrent wrt $|.|$ and I
2. The delay declarations for P are safe wrt $|.|$

Then every SLD-derivation for a query Q, using a semi-local selection rule is finite. □

We are now able to develop a program transformation based on the above result. We begin by transforming a given program into one which is semi delay recurrent, but with equivalent declarative semantics. Then by adding safe delay declarations we can obtain a program which terminates for all queries using a semi-local selection rule.

Definition 12 semi delay recurrent transform sdr**.** The transform sdr is defined as follows.

$$
\begin{array}{ll}
p \in \Sigma_P & \Rightarrow p \in \Sigma_{\mathsf{sdr}(P)} \wedge p^{\mathsf{sdr}} \in \Sigma_{\mathsf{sdr}(P)} \text{ where } p^{\mathsf{sdr}} \notin \Sigma_P \\
\forall(p(\overline{t}) \leftarrow) \in P & \Rightarrow \forall(p^{\mathsf{sdr}}(\overline{t}, \_) \leftarrow) \in \mathsf{sdr}(P) \\
c = \forall(p(\overline{t}) \leftarrow w) \in P & \Rightarrow \forall(p^{\mathsf{sdr}}(\overline{t}, d) \leftarrow d = \nu_c(\overline{d}) \wedge w') \in \mathsf{sdr}(P)
\end{array}
$$

where w' is obtained from w by replacing each atom in w of the form $q^i(\overline{s})$ with $q^i_{\mathsf{sdr}}(\overline{s}, d_i)$ if $p \simeq q^i$, $\overline{d}$ is a tuple such that $d_i \in \overline{d}$ if $p \simeq q^i$ and ν_c is a function with the property that $\nu_c(\overline{d}) > d_i \; \forall d_i \in \overline{d}$. The variables d and $d_i, \forall i$ are domain variables over $\mathbb{N}$. Finally for each $p \in \Sigma_P$ we introduce the auxiliary clause

$$\forall(p(\overline{t}) \leftarrow p^{\mathsf{depth}}(\overline{t}, d) \wedge p^{\mathsf{sdr}}(\overline{t}, d)) \in \mathsf{sdr}(P)$$

where $\overline{t}$ is a tuple of variables. □

Lemma 13 semi delay recurrency. If for each $p \in \Sigma_P$, the clauses defining p^{depth} are semi delay recurrent wrt $M(\mathsf{sdr}(P))$ and $||.||$, then the program sdr(P) is semi delay recurrent wrt $M(\mathsf{sdr}(P))$ and the level mapping $|.|$ defined by

$$\begin{aligned} |p^{\mathsf{sdr}}(\bar{t}, d)| &= d \\ |p(\bar{t})| &= 0 \\ |p^{\mathsf{depth}}(\bar{t})| &= ||p^{\mathsf{depth}}(\bar{t})|| \end{aligned}$$

for all $p \in \Sigma_P$. □

By Theorem 11 and Lemma 13 we can obtain a program which terminates for all queries under a semi-local computation rule by adding for each predicate, a delay declaration which is safe wrt the level mapping defined in Lemma 13. Note also that $d = \nu(\overline{d})$ is the only atom in the body of each non-auxiliary clause which will be a covering atom in a goal. This means that after its resolution, an arbitrary amount of coroutining may take place between the atoms in w'.

Example 2. The program of Section 2 is obtained by applying the above transform, with $\nu(d) = d + 1$, to the Quicksort program of Section 1 and adding safe delay declarations. Notice that the number of suspension checks performed has been minimised by introducing an auxiliary clause $p_1(\bar{t}) \leftarrow p_2(\bar{t})$ for each predicate p. □

4.2 Completeness

Having obtained a terminating program, we need to prove that the declarative semantics of the transformed program coincide with those of the original program. In this way, under the assumption that the transformed program is deadlock free [12], we can guarantee that all computed answers of this program are complete wrt the declarative semantics of the original program. We have the following result.

Lemma 14 equivalence. If $M(P) \models p(\bar{t})$ and $d \in \{d \mid M(\mathsf{sdr}(P)) \models p(\bar{t}, d)\}$ implies $M(\mathsf{sdr}(P)) \models p^{\mathsf{depth}}(\bar{t}, d)$ then for all $p \in \Sigma_P$

$$p(\bar{t}) \in M(P) \Leftrightarrow p(\bar{t}) \in M(\mathsf{sdr}(P))$$

The problem then is to define p^{depth} for each $p \in \Sigma_P$ such that the above equivalence result holds. Our novel solution to this problem uses information about the success set of the program. Suppose we can deduce, for example, that for a given goal G, all computed answers for G can be found in an SLD-tree of fixed depth, then we can compute the SLD-tree to that depth and no more, and be sure that we have found all answers for G. In reality, the granularity is finer, relying not on the depth of the SLD-tree as a whole but rather on the lengths of individual branches. More precisely, for each predicate p we find an upper bound on the number of calls to p. It will often be the case that this bound relates to the input arguments of the predicate. We thus use interargument relationships to capture this relation. Essentially, we define p^{depth} as the interargument relationship of the predicate p^{adr}.

Definition 15 interargument relationship. Given $p \in \Sigma_P$, a norm $|.|$ and a model M for p/n, an *interargument relationship* for p/n wrt S is a relation $I \subseteq \mathbb{N}^n$, such that if $M \models p(\bar{t})$ then $p(|\bar{t}|) \in I$. □

Interargument relationships can be automatically deduced using, for example, the analysis described in [4].

Example 3. The analysis in [4] can be used to deduce the argument size relations $I_{\mathsf{Quicksort_{abs}/3}} = \{\langle x, y, d\rangle \mid x = y, d = x\}$, $I_{\mathsf{Append_{abs}/4}} = \{\langle x, y, z, d\rangle \mid z = x + y, d = x\}$ and $I_{\mathsf{Partition_{abs}/5}} = \{\langle w, x, y, z, d\rangle \mid w = y + z, d = w\}$. These relations can be used to derive the definitions of SetDepth_Q/3, SetDepth_A/3 and SetDepth_P/4 for the program sdr(Quicksort) in Section 2. □

Example 4. Given the following predicate Split from the program Mergesort

```
Split([], [], []).
Split([x|xs], [x|o], e) ← Split(xs, e, o).
```

the argument size relation $I_{\mathsf{Split_{abs}/3}} = \{\langle x, y, z, d\rangle \mid d = x, d \leq 2y, d \leq 2z + 1\}$ can be derived. From this we can derive a program which terminates for all queries ← Split(x, y, z) where either x, y or z is a list of determinate length and the remaining two arguments are (optionally) unbound. We know of no other technique in the literature which can prove termination of these queries. The majority of approaches can only reason about the decrease in the level mapping of successive goals in a derivation. For the level mappings $|\mathsf{Split}(t_1, t_2, t_3)|_1 = |t_1|$ and $|\mathsf{Split}(t_1, t_2, t_3)|_2 = |t_2|$ the decrease only occurs on every second goal. A similar problem which our approach can also deal with occurs in [13].

4.3 Efficiency

We now give a brief appraisal of our approach from a performance perspective.

In theory, the rigidity checks should not incur much more overhead than the original delay declarations. For example, checking rigidity of the first argument of the query ← Append([1,2,3], y, z) requires three Nonvar tests - exactly the same number that would be required if the query were executed using the conventional delay declarations. There are additional costs due to unification and the calculation of the depth bound, but these costs could be minimised through careful implementation. We have naively implemented and tested some sample programs and some of the preliminary results are given below. The experiments have been carried out in SICStus Prolog [7] on a Sparc 4.

Program P	Goal G	Length of list L	Time(s) for $P \cup \{G\}$ one solution	all solutions	Time(s) for sdr(P) $\cup \{G\}$ one solution	all solutions
8-queens	qn(_)	-	0.4	6.8	0.3	5.3
permsort	ps(L, _)	10	6.8	∞	0.7	0.7
permsort	ps(_, L)	8	1.7	10.5	2.6	10.8
quicksort	qs(L, _)	4000	3.7	4.5	4.8	6.0
quicksort	qs(_, L)	8	$12ms$	∞	$6ms$	83.0

The main overhead is due to the rigidity checks and our implementation in this respect is rather naive and could be improved. Even in our experimental implementation this overhead only reaches a maximum factor of about three for the simplest programs, e.g. Append. The power of our approach, however, lies in its scalability and it is here where we believe the most impressive performance gains are to be made. Preliminary tests indicate that the most benefit is obtained from larger programs where only one rigidity test is performed at the beginning of the program and the rest of the computation is bounded by the depth bounds. Then our programs can outperform the original ones with the delay declarations, particularly as the amount of backtracking or coroutining increases.

5 Conclusion

The aim of control generation is to automatically derive a computation rule for a program that is efficient but does not compromise program correctness. In our approach to this problem we have transformed a program into a semantically equivalent one, introduced delay declarations and defined a flexible computation rule which ensures that all queries for the transformed program terminate. Furthermore, we have shown that the answers computed by the transformed program are complete with respect to the declarative semantics. This is significant.

Beyond the theoretical aspects of the work, we have demonstrated its practicality. In particular, we have shown how transformed programs can be easily implemented in a standard logic programming language and how such a program can be optimised to reduce the number of costly rigidity checks needed to ensure termination, dramatically improving its performance. Furthermore, we have seen how the termination problems caused by speculative output bindings can be eliminated without the use of a local computation rule or other costly overhead. The coroutining behaviour which is then possible contributes significantly to the efficiency of the generated code.

In terms of correctness, we have only considered termination and completeness in this work, though other correctness issues also need investigating. We believe the connection between acyclic modes and rigid terms may provide a solution in our approach to the occur check problem, since the check is never needed for acyclic moded goals. Also, the example of Section 2.2 illustrates how the problem of deadlock freedom may be handled.

The efficiency issues also require further investigation. We have separated to some extent the issues of termination and performance and it is not now clear what role extra delay declarations might play in improving the performance of the transformed programs, or even whether other techniques such as multiple specialisation would be more appropriate.

Acknowledgements

The authors would like to thank Elena Marchiori for providing useful literature and clarifying their understanding of delay recurrency.

References

1. K.R. Apt and M. Bezem. Acyclic programs. In David H. D. Warren and Péter Szeredi, editors, *Proceedings of the Seventh International Conference on Logic Programming*, pages 617–633, Jerusalem, 1990. The MIT Press.
2. K.R. Apt and D. Pedreschi. Proving termination of general Prolog programs. In T. Ito and A. R. Meyer, editors, *Theoretical Aspects of Computer Software*, volume 526 of *Lecture Notes in Computer Science*, pages 265–289, Sendai, Japan, September 1991. Springer-Verlag.
3. K.R. Apt and D. Pedreschi. Modular termination proofs for logic and pure Prolog programs. In G. Levi, editor, *Proceedings of the Fourth International School for Computer Science Researchers*. Oxford University Press, 1994.
4. F. Benoy and A. King. Inferring argument size relations with CLP($\mathcal{R}$). In *LOPSTR'96*. Springer-Verlag, 1996.
5. M. Bezem. Characterizing termination of logic programs with level mappings. In *NACLP'89*, pages 69–80, Cleveland, Ohio, USA, 1989. MIT Press.
6. P.M. Hill and J.W. Lloyd. *The Gödel Programming Language*. MIT Press, 1994.
7. Intelligent Systems Laboratory, SICS, PO Box 1263, S-164 28 Kista, Sweden. *SICStus Prolog User's Manual*, 1995.
8. R. Kowalski. Algorithm = Logic + Control. *Communications of the ACM*, 22(7):424–436, July 1979.
9. J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, second edition, 1987.
10. S. Lüttringhaus-Kappel. Control Generation for Logic Programs. In *ICLP'93*, pages 478–495. MIT Press, 1993.
11. E. Marchiori and F. Teusink. Proving termination of logic programs with delay declarations. In *ILPS'95*, pages 447–461. MIT Press, 1995.
12. E. Marchiori and F. Teusink. Proving deadlock freedom of logic programs with dynamic scheduling. In F.de Boer and M.Gabbrielli, editors, *JICSLP'96 Post-Conference Workshop W2 on Verification and Analysis of Logic Programs*, Bonn, 1996. TR-96-31, University of Pisa, Italy.
13. F. Mesnard. Towards Automatic Control for CLP($\mathcal{X}$) Programs. In *LOPSTR'95*. Springer-Verlag, 1995.
14. L. Naish. Coroutining and the construction of terminating logic programs. In *Australian Computer Science Conference*, Brisbane, February 1993.