Computer Science at Kent

Control Generation by Program Transformation

Jonathan C. Martin and Andy King

Technical Report No. 5-04
March 2004

Abstract: The objective of control generation in logic programming is to derive a computation rule for a program that is efficient and yet does not compromise program correctness. Progress in solving this fundamental problem in logic programming has been slow and, to date, only partial solutions have been proposed. Previously proposed schemes are either inefficient, incomplete (incorrect) or difficult to apply for programs consisting of many components (the scheme is not modular). This paper shows how the control generation problem can be tackled by program transformation. The transform relies on information about the depths of derivations to derive delay declarations which orchestrate the control. To prove correctness of the transform, the notion of semi-delay recurrency is introduced, which generalises previous ideas in the termination literature for reasoning about logic programs with delay declarations. In contrast to previous work, semi-delay recurrency does not require an atom to be completely resolved before another is selected for reduction. This enhancement permits the transformation to introduce control which is both flexible and efficient.

Published by the Computing Laboratory,
University of Kent, Canterbury, Kent, CT2 7NF, UK

1 Introduction

A logic program can be considered as consisting of a logic component and a control component [24]. 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 [7, 36]. Delays have been applied to, among other things, handle negation [12], delay non-linear constraints [22], support coroutining [42], improve search by rescheduling tests [10], implement concurrency protocols [41] and induce termination [37]. In general, delay mechanisms are used to define dynamic selection rules with the two main aims of enhancing performance through coroutining or expressiveness whilst ensuring termination. However, reasoning about logic programs with delay is notoriously difficult and often completeness, and hence program correctness, is inadvertently sacrificed by the programmer in the quest for improved efficiency or generality.

Consider, for example, the Prolog program that is listed below:

```
inorder(nil,[]).
inorder(tree(L,V,R), I) :- inorder(L, LI), inorder(R, RI), append(LI, [V|RI], I).

append([], X, X).
append([X|Xs], Ys, [X|Zs]) :- append(Xs, Ys, Zs).
```

Declaratively, the program defines the relation that the second argument (a list) is an in-order traversal of the first argument (a tree). With leftmost selection – the default computational rule of Prolog – the goal inorder(tree(nil, a, tree(nil, b, nil)), L) will terminate and bind L to [a, b]. The goal inorder(T, [a,b]) will also return an answer, namely, T = tree(nil, a, tree(nil, b, nil)). One subtlety, however, is that this goal has another answer, namely, T = tree(tree(nil, a, nil), b, nil) yet a request to compute this answer results in a diverging computation. Thus inorder existentially terminates (it finds one answer) but it does not universally terminate (it does not enumerate all the answers) [44]. The program can be enhanced by adding the delay declaration :- block append(-, ?, -) which asserts that an append atom cannot be selected until either its first or third argument is instantiated to a non-variable term. (Note that since the syntax and semantics of delay declarations have not been standardised [16], for purposes of exposition the paper adopts the block declarations of SICStus [40] since these can be simulated with, say, the DELAY declarations of Gödel [19] and the wait declarations of MU-Prolog [36].) This block declaration, however, is not by itself sufficient for universal termination in both modes. In addition, the body atoms of inorder have to be reordered to obtain:

```
inorder(nil,[]).
inorder(tree(L,V,R), I) :- append(LI, [V|RI], I), inorder(L, LI), inorder(R, RI).

:- block append(-, ?, -).
append([], X, X).
append([X|Xs], Ys, [X|Zs]) :- append(Xs, Ys, Zs).
```

Then both goals inorder(tree(nil, a, tree(nil, b, nil)), L) and inorder(T, [a, b]) universally terminate (at least with a computation rule that invokes a suspended goal as soon as its delay condition is satisfied [40]). This program illustrates an unsatisfactory feature of logic programming: although the declarative behaviour of inorder is transparent, its operational behaviour is opaque because of the control. The program also illustrates that delay declarations *can* be applied to obtain completeness and correctness. The problem is how to add delays systematically; the challenge is how to add delays automatically.

The seminal paper on logic and control [24] has been interpreted by many as setting an agenda for logic programming: control generation [20, 26, 37, 38, 45]. The objective of control generation in logic programming is to derive a computation rule for a program that is efficient and yet does not compromise completeness, hence universal termination. Progress in solving this fundamental problem has been slow, despite recent advances on termination checking in the context of sophisticated control [6, 39, 32]. However, crucial insight into this problem is given by Naish [36] within a discussion on the desirable properties of computation rules. Naish writes,

> "There is a slightly more subtle rule [computation rule tactic] which applies more to goals which have solutions ... Although the success branches of a tree are fixed, the number and length of the other branches are not. The rule, therefore, is to avoid the creation of unnecessary failure (and infinite) branches".

Martin and King [30] introduce a program transform which applies this tactic to generate a control component from the logic component that prunes out infinite branches. Specifically, if the maximum depth of the SLD-tree needed to solve a given goal can be determined, then by only searching to that depth, the goal will be completely solved. Hence all the answers will be obtained, in a finite number of steps, thereby enforcing termination without compromising completeness.

Pedreschi and Ruggieri [38] apply the same tactic with a similar transformation. This latter transform adds a counter that is decremented on each resolution step. Once the counter reaches zero, the derivation is failed. The success of the transformation is predicated on being able to find an upper bound on the depth of all the successful, finite SLD-trees. By way of contrast, the transformation advocated by [30] introduces not one but possibly many counters; one counter is used for each recursive loop in the program to control how many times it is traversed. One advantage of this approach is that it gives finer control on the depth to which the search space is probed, thereby improving efficiency. Moreover, it is easier to compute an upper bound on how many times a loop is traversed than find an upper bound on the total depth of the SLD-tree. The issue here is one of modularity [2, 5]. Ideally a termination argument or dually, a construction that enforces termination, should be modular in the sense that it should be derived in a bottom-up fashion by considering, in turn, each of the strongly connected components (SCCs) in the predicate dependency graph. The problem that complicates the derivation of the level mappings [31] which underpin termination analysis is the same problem that complicates the derivation of a depth bound in control generation. In both cases, it is necessary to consider not only the recursive behaviour of those predicates in the top-level SCC, but those predicates (sub-computations) that are invoked from within it. The modularity problem in termination analysis is to synthesis a level mapping which ensures that each call to a sub-computation is bounded. The modularity problem in control generation is to infer a depth bound such that each sub-computation falls within the bound. The approach to control generation proposed in this paper alleviates much of this problem. The disadvantage of the transform proposed by [30], as Pedreschi and Ruggieri [38] correctly point out, is that it incurs an additional run-time overhead for computing depth bounds on-the-fly for each sub-computation. However, as this paper shows, the transformation results in code that is surprisingly efficient.

Since this paper draws together a number of threads in the termination and control generation literature, as well as extending preliminary work on this topic [30], we summarise its contributions:

- The paper explains how the control generation problem can be tackled by program transformation. It also shows that the performance of program developed with control generation can compare favourably with that of a logically equivalent program equipped with *ad hoc* delay declarations.

- The paper explains how control generation can be tackled in a modular fashion.

- The paper formally argues the correctness of the transformation. Since it introduces delays to orchestrate the control, the correctness proof is non-trivial. In fact, the argument is formulated in terms of a new class of logic program – the class of semi-delay recurrent program (which was originally sketched in [30]) – which refines the concept of delay recurrency [28, 29]. Delay recurrency was proposed for proving termination of logic programs with delay. Delay recurrency is sufficient for termination provided the program is executed under local selection whilst satisfying the delay conditions. However, under local selection, the selected atom is completely resolved, that is, those atoms it directly and indirectly introduces are also resolved, before any other atom is selected [46]. As [29] concede, this bars coroutining. By way of contrast, semi-delay recurrent programs permit coroutining and thereby provide a foundation for more flexible and efficient control generation. Moreover, semi-delay recurrent programs can be directly implemented in systems such as SICStus Prolog [40] which implement the scheduling tactic that suspended goals should be invoked as soon as their delay conditions as satisfied [10].

- Finally, to make the ideas more accessible to a general program transformation audience, the paper reviews the problems that arise in control generation. It surveys some of the proposed solutions as well as their short-comings.

The review constitutes section 2. Section 3 illustrates the transformation tactic with a familiar example program. Section 4 summaries the key ideas in delay recurrency [29], before moving on to refine these concepts to introduce the class of semi-delay recurrent logic. Section 5 presents a formal development of the proposed transformation, including the correctness results. The transformed programs are by construction semi-delay recurrent and hence termination is guaranteed. Section 6 discusses automation and appraises of the performance and practicality of this transformational approach to control generation. Finally, section 7 presents the concluding discussion.

2 Issues of dynamism in control generation

The presence of delayed goals in a computation significantly complicates the termination behaviour of a logic program. This section reviews the kind of subtleties which can arise in using delays to induce the termination of logic programs, explaining why there is still room for improving the solutions that have been previously proposed.

2.1 The issue of local boundedness

Consider the append program introduced previously, complete with a block declaration which delays the selection of an append atom until either the first or third arguments are non-variable:

```
:- block append(-, ?, -).
append([], X, X).
append([X|Xs], Ys, [X|Zs]) :- append(Xs, Ys, Zs).
```

Although the block declaration is intended to assist termination, it is not sufficient to ensure that *all* append goals terminate. As Naish [37] points out, the goal append([X|Xs], Ys, Xs) satisfies the condition in the declaration and therefore does not suspend, yet its derivation is an infinite one. The derivation is infinite since unifying the goal with the head of the second clause will instantiate Xs to [X|Zs] so the sub-goal append(Xs, Ys, Zs) is then a variant of the initial goal.

Termination can only be guaranteed for all goals by strengthening the condition in the delay declaration. This is where the trade-off between efficiency, termination and non-suspension takes place. Ideally, the delay declarations would be as weak as possible since strengthening a delay declaration amounts to inspecting more sub-terms of a goal which in turn decreases efficiency. On the other hand, the delay condition should be strong enough to bar the resolution of goals which have infinite derivations. Yet, if the delay condition is too strong, goals may suspend which actually possess finite derivations. Thus one of the main problems in generating control is that of finding a suitable delay condition for a predicate that suspends those goals that have an infinite derivation, yet does not suspend goals which possess finite derivations. Moreover, the delay condition for a predicate should ideally be inexpensive to check. This trade-off primarily relates atomic goals – a call to a predicate and those it invokes – rather than the interaction between the sub-goals of a goal which shape how a search tree is traversed. The trade-off thus relates to *local* effects rather than global interactions. In particular, the trade-off relates to the degree of instantiation that arguments of an atomic goal require, or adopting terminology more in tune with the termination literature [13], whether the arguments of the goal are *bounded* with respect to some level mapping. This simply means that every instance of the goal is of fixed size where size is defined by a level mapping that maps goals to natural numbers. Thus, henceforth, this trade-off in control generation will be referred to as the *local boundedness* issue. There have been several attempts at addressing this problem, each of which will be discussed below with reference to the append program.

2.1.1 On using linearity

Lüttringhaus-Kappel [26] observed that for atomic goals, linearity is a property that is often sufficient for termination. Linearity relates to the number of times variables occur within a goal: a goal is non-linear if at least one variable occurs multiply within the goal; otherwise it is linear. Thus the goal append([X|Xs], Ys, Xs) is non-linear because the first and third arguments both contain the variable Xs. Delaying the goal append([X|Xs], Ys, Xs) until it is linear is equivalent to delaying the goal until Xs is instantiated to a term that contains no variables. This would indeed prevent looping. However, checking for linearity requires all the sub-terms within the arguments of a goal to be completely traversed and is therefore potentially expensive. Moreover, there is little prospect of applying abstract interpretation techniques [23] to detecting linearity at compile-time since these analysis techniques rely on the control being predetermined. More significantly still, delaying a goal until it is linear would suspend goals with finite derivations such as append([X, X], Ys, Zs). Thus linearity would be potentially expensive to enforce and would additionally suspend goals that have finite derivations.

2.1.2 On using rigidity

Mesnard [33] proposes generating control by delaying goals until their arguments are bound to rigid terms. A term is rigid with respect to a norm – a function that assigns a size to the term – if its size cannot change even when the variables within the term are further instantiated. For example, a term is rigid with respect to the list-length norm if the term is a list of determinate length. Mesnard [33] infers that append will terminate if called when its first or third arguments are rigid with respect to the list-length norm. Hence, calls to append are blocked until this rigidity property holds. This promising tactic successfully enforces termination but at the price of traversing (at least) one list on *each* call to the predicate. In fact, the check is only performed on the initial call [33], but no justification for this refinement is given. Thus the issues with this approach are primarily those of efficiency and correctness.

2.1.3 On using modes

Naish [37] proposes solving the problem of the goal append([X|Xs], Ys, Xs) with the use of modes which flag the fact that output should not feed back into the input. The mode system that Naish advocates is unusual in that modes are usually specifications on single procedures rather than conjunctions (a notable exception being the mode system of IC-Prolog which applied a form of mode to specify eager consumers and lazy producers [10]). However, although modes may form a good basis for solving this problem, they have not been shown to be satisfactory for reasoning about another termination problem, that of speculative output bindings. This is discussed below.

2.2 The issue of global boundedness

Even when finite derivations exist, delay conditions alone are not, in general, sufficient to ensure termination. Infinite computations may arise as a result of so-called speculative output bindings [37], which can arise from *global* interactions between the sub-goals of a goal due to dynamic selection. The effect that they have on termination is the focus of interest here and henceforth this issue will be referred to as the *global boundedness* issue. To illustrate the problem caused by speculative output bindings consider the familiar qsort program listed in figure 1 augmented with the clause:

```
append(X, [_|X], X) :- fail.
```

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. But this is not the case as the goal qsort(X, [1,2,3]) illustrates. Following resolution with the second clause of qsort, the only goal that can be selected is append(Ls,

```
:- block qsort(-, -).
qsort([], []).
qsort([X|Xs], Ys) :-
    part(Xs, X, L, B), qsort(L, Ls), qsort(B, Bs), append(Ls, [X|Bs], Ys).

:- block part(-, ?, -, ?), part(-, ?, ?, -).
part([], _, [], []).
part([X|Xs], Y, [X|Ls], Bs) :- leq(X, Y), part(Xs, Y, Ls, Bs).
part([X|Xs], Y, Ls, [X|Bs]) :- gt(X, Y), part(Xs, Y, Ls, Bs).

:- block append(-, ?, -).
append([], X, X).
append([X|Xs], Ys, [X|Zs]) :- append(Xs, Ys, Zs).

:- block leq(-, ?), leq(?, -).
leq(X, Y) :- X ≤ Y.

:- block gt(-, ?), gt(?, -).
gt(X, Y) :- X > Y.
```

Figure 1: The quicksort algorithm expressed in Prolog

[X|Bs], [1,2,3]). When this is unified with the above clause, both Ls and Bs are bound to the list [1,2,3]. These bindings are speculative because if the sub-goal fail were to be selected, then they would be retracted. As a result of these speculative output bindings the previously suspended sub-goals qsort(L, Ls) and qsort(B, Bs) can resume *before* the fail is selected. The net result is an infinite derivation due to recurring sub-goals of the form qsort(X, [1,2,3]). The problem is that bindings are made before failure is detected and no matter how stringent the delay conditions, loops of this kind cannot generally be avoided without regard for the computation as a whole.

2.2.1 On using local selection rules

To remedy this problem, Marchiori and Teusink [28, 29] propose the use of a local selection rule. Such a rule only selects atoms from those that are most recently introduced in a derivation [46]. 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 append sub-goals would be fully resolved, hence fail would be selected before the qsort sub-goals are reawakened. This would prevent an infinite loop, but at the expense of not allowing any form of coroutining. This is a severe restriction since coroutining is a useful programming tactic within logic programming.

Local selection is also applied in the control generation work of Hoarau and Mesnard [20]. This work builds on the termination inference methods developed by Mesnard and his colleagues [34, 35] and infers initial modes for a query that, if satisfied, ensure that a logic program left-terminates. The chief advance described in [20] is that the paper additionally infers how goals can be statically reordered so to obtain termination with leftmost selection. Of course, leftmost selection, prohibits coroutining.

2.2.2 On delaying output unification

Naish [37] proposes finessing the problem of speculative output bindings by postponing output unification. In the case of the above example, the clause could be rewritten to:

```
append(X, Y, Z) :- fail, Y = [_|X], Z = X.
```

This transformation presupposes left-to-right scheduling as the default. The intended effect of the transformation is that no output bindings should be made until the computation is known to succeed. This tactic parallels local selection in that it restricts coroutining.

3 Worked example

This paper shows how program transformation can be applied to solve both the local boundedness problem and the global boundedness problem. The transform introduces delay declarations that realise rigidity checks. These solve the local boundedness problem in the first instance by ensuring that the initial goal is bounded; sufficiently instantiated to assure termination. Boundedness of subsequent goals, however, is ensured by depth counters, which are introduced by the transform so as to eliminate subsequent rigidity checks. The global boundedness problem is also neatly solved. The depth counters ensure that the search space is finite, so even though speculative output bindings may still occur, they cannot lead to infinite derivations.

```
:- block msort(-, -).
msort([], []).
msort([X], [X]).
msort([X, Y|Xs], S) :-
    split([X, Y|Xs], L1, L2), msort(L1, S1), msort(L2, S2), merge(S1, S2, S).

:- block split(-, -, -).
split([], [], []).
split([X|L], [X|L1], L2) :- split(L, L2, L1).

:- block merge(-, ?, -), merge(?, -, -).
merge([], Ys, Ys).
merge(Xs, [], Xs).
merge([X|Xs], [Y|Ys], [X|Zs]) :- leq(X, Y), merge(Xs, [Y|Ys], Zs).
merge([X|Xs], [Y|Ys], [Y|Zs]) :- gt(X, Y), merge([X|Xs], Ys, Zs).
```

Figure 2: The mergesort algorithm expressed in Prolog

Moreover, the global boundedness problem is overcome without grossly impeding coroutining.

To illustrate the approach, consider the mergesort program listed in figure 2, assuming the predicates append, leq and gt are defined as in figure 1. The delay declarations are an attempt to enhance the program so that it can be additionally used in reverse; the query msort(L, [1,2,3]) should in principle terminate and systematically enumerate the solutions L = [1,2,3], L = [1,3,2], L = [2,1,3], L = [2,3,1], L = [3,1,2] and L = [3,2,1] though not necessarily in that order. Such a reversal would genuinely be useful because permutation generation frequently arises in solving constraint satisfaction problems. For example, figure 3 illustrates how a magic square program might be constructed in terms of msort. The cells of the square are constrained to be unique digits. The delay declaration on sum asserts that it cannot be invoked until its first, second and third arguments are all instantiated to non-variable terms.

The attempt at reversing mergesort given in figure 2 is not completely naïve. For instance, split will terminate if any of its arguments are bound to a rigid list (a list of predetermined length whose elements are not necessarily fixed). This is reflected in the delay declaration which asserts that split should block until any of its arguments are bound to a non-variable term.

```
magic(Square) :-
    Square = [A, B, C, D, E, F, G, H, I],
    sum(A, B, C, Sum), sum(D, E, F, Sum), sum(G, H, I, Sum),
    sum(A, D, G, Sum), sum(B, E, H, Sum), sum(C, F, I, Sum),
    sum(A, E, I, Sum), sum(G, E, C, Sum),
    msort([_ | Square], [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]).

:- block sum(-, ?, ?, ?), sum(?, -, ?, ?), sum(?, ?, -, ?).
sum(A, B, C, Sum) :- Sum is A + B + C.
```

Figure 3: Magic square generation using mergesort

In fact a query such as msort(L, [1,2,3]) *does* enumerate all its solutions; the problem is that it loops after doing so. This problem stems from the fact that split([X,Y|Xs], L1, L2) will instantiate L1 and L2 to lists, thereby enabling msort(L1, S1) and msort(L2, S2) to be invoked. This can instantiate L1 and L2 to 2 element lists, which in turn enables split([X,Y|Xs], L1, L2) to be reactivated to instantiate Xs to a list of 2 elements, so that L is of length 4. This process can continue without bound.

The mergesort program can be transformed into a version where termination is guaranteed for all goals. In particular, for a goal of the form msort(L1, L2), where L1 or L2 are lists of numbers, then computation will not flounder, that is, enter a state of indefinite suspension. Moreover, if it succeeds, then the set of answers produced is complete with respect to the declarative semantics. The transformed program is shown in figure 4. Each recursive predicate in the original program, namely msort, split and merge, is augmented by another predicate, namely msort_depth, split_depth and merge_depth, which computes a depth bound for that predicate. In addition, each predicate is replaced with an instrumented version, distinguished with the sdr suffix, which keeps track of the depth of the recursive calls. The auxiliary predicates that support the transform are given separately in figure 5.

More precisely, the predicate msort_depth(L1, L2, D) calculates the lengths of the lists L1 and L2 with the predicate list_length. The list_length predicate, which is not listed, merely traverses the spline of a list, using an accumulator to calculate the length, blocking whenever a variable is encountered. Once L1 or L2 are found to be rigid, the auxiliary predicate msort_aux calculates an upper bound D on the number of times that msort_sdr can call

```
msort(L, S) :- msort_depth(L, S, D), msort_sdr(L, S, D).

:- block msort_sdr(?, ?, -).
msort_sdr([], [], _).
msort_sdr([X], [X], _).
msort_sdr([X, Y | Xs], S, D) :- 1 ≤ D, D1 is D - 1,
    split([X, Y | Xs], L1, L2),
    msort_sdr(L1, S1, D1), msort_sdr(L2, S2, D1), merge(S1, S2, S).

split(L, L1, L2) :- split_depth(L, L1, L2, D), split_sdr(L, L1, L2, D).

:- block split_sdr(?, ?, ?, -).
split_sdr([], [], [], _).
split_sdr([X | L], [X | L1], L2, D) :- 1 ≤ D, D1 is D - 1,
    split_sdr(L, L2, L1, D1).

merge(L1, L2, L) :- merge_depth(L1, L2, L, D), merge_sdr(L1, L2, L, D).

:- block merge_sdr(?, ?, ?, -).
merge_sdr([], Ys, Ys, _).
merge_sdr(Xs, [], Xs, _).
merge_sdr([X|Xs], [Y|Ys], [X|Zs], D) :- 1 ≤ D, D1 is D - 1,
    leq(X, Y), merge_sdr(Xs, [Y|Ys], Zs, D1).
merge_sdr([X|Xs], [Y|Ys], [Y|Zs], D) :- 1 ≤ D, D1 is D - 1,
    gt(X, Y), merge_sdr([X|Xs], Ys, Zs, D1).
```

Figure 4: Control generation applied to the mergesort program

itself. Because of its divide-and-conquer nature, msort can call itself precisely $\lceil \log_2(l) \rceil$ times where l is the length of L1, or equivalently, L2. Once D is instantiated to this value, the call to msort_sdr(L, S, D) can proceed. The purpose of the last argument is to ensure finiteness of the subsequence computation. The key point is that D is an upper bound on the number of recursive calls to msort *in any successful derivation*. Thus, by failing any derivation that exceeds this bound, termination is guaranteed without losing completeness. Observe that the depth bound for msort is only required to bound the depth of the predicates within the SCC for msort. Indeed, $\lceil \log_2(l) \rceil$ does not bound the depth of the sub-computations for split and merge. Thus depth bounds can be calculated in an entirely modular fashion by considering each SCC in isolation.

For split, a depth bound on the number of recursive calls can be calculated as soon as either the first, second or third argument is rigid. This is because the depth bound is only required to be an *upper* bound. For instance, observe that if split is called with the third argument of length 2, and the call succeeds, then the first argument must be of length 4 or 5. Hence the number of recursive calls is either 4 or 5. Either way, $2l + 1$ is an upper bound on the depth where l is the length of the third argument. Similarly, $2l$ is an upper bound on the depth where l is the length of the second argument. The calculation of an exact depth bound would require the rigidity of both the second and third arguments, which would unnecessarily impede the execution of split. A less precise upper bound could be applied in msort_aux to save distinguishing two cases by modifying the second clause to D is integer(ceiling(log(D1 + 1) / log(2))), which is sufficient for avoiding a problematic zero logarithm. Alternatively putting D = D1 would avoid a logarithm altogether. In the case of merge, a depth bound can be calculated as soon as either the first two arguments are rigid, or the third argument is rigid, but not earlier.

To ensure termination, the computational rule is required to select each test $1 \leq$ D before any other body atom. Any system implementing left-to-right selection rule (with delay) will satisfy this requirement. Thus termination is enforced with delays without resorting to a local computation rule [29]. Observe too that list_length predicate has 3 rather than 2 arguments. This is to save traversing lists unnecessarily. In the predicate msort_depth, for example, if the call list_length(L, D1, D) instantiates D1, then msort_aux(D1, D2, D) will bind D so that the length of S is inconsequential. Hence list_length terminates immediately if its third argument is instantiated.

Interestingly, the transformation yields a completely reversible merge-sort program that possesses unexpected, though useful, computational properties. For instance, the query msort(L, [1,2,2]) produces the 3 solutions: L = [1,2,2], L = [2,1,2] and L = [2,2,1] without producing any solution multiply. The reversal of msort is thus a combination generator rather than a permutation generator. A combination generator would be difficult to program directly without some notion of state to filter out replicated solutions. Incidentally, with the version msort listed in figure 4, magic(Square) finds two solutions, namely Square = [6,1,8,7,5,3,2,9,4] and Square = [3,8,1,2,4,6,7,0,5] (the other solutions are reflections). In the first case the rows, columns and diagonals all sum to 15 and in the other case they all sum to 12.

```
msort_depth(L, S, D) :-
    list_length(L, D1, D), list_length(S, D2, D),
    msort_aux(D1, D2, D).

split_depth(L, L1, L2, D) :-
    list_length(L, S, D), list_length(L1, S1, D), list_length(L2, S2, D),
    split_aux(S, S1, S2, D).

merge_depth(L1, L2, L, F) :-
    list_length(L1, S1, D), list_length(L2, S2, D), list_length(L, S, D),
    merge_aux(S1, S2, S, D).

:- block msort_aux(-, -, ?).
msort_aux(D1, D2, D) :- D1 = D2, D1 = 0, !, D = 0.
msort_aux(D1, D2, D) :- D1 = D2, D is integer(ceiling(log(D1) / log(2))).

:- block split_aux(-, -, -, ?).
split_aux(S, _, _, D) :- nonvar(S), !, D = S.
split_aux(_, S1, _, D) :- nonvar(S1), !, D is 2 * S1.
split_aux(_, _, S2, D) :- D is 2 * S2 + 1.

:- block merge_aux(-, ?, -, ?), merge_aux(?, -, -, ?).
merge_aux(S1, S2, _, D) :- nonvar(S1), nonvar(S2), !, D is S1 + S2 - 1.
merge_aux(_, _, S, D) :- D is S - 1.
```

Figure 5: Auxiliary predicates supporting the transformation

4 Theoretical foundations

To provide a sound theoretical basis for termination of delay logic programs, it is natural to build on the preceding theoretical foundations established for logic programs.

4.1 Level mappings, norms and boundedness

The fundamental idea underlying all termination proofs is to define an order on the goals that can occur within a derivation. Given a program P and goal G_0, the finiteness of derivation $G_0, G_1, G_2, \ldots$ is in principle straightforward to demonstrate: it is sufficient to construct a well-founded order $<$ such that $G_{i+1} < G_i$ for all $i \geq 0$. The problem is to find such an order. To simplify the problem, it is convenient to define the order on abstractions of goals rather than on the goals themselves. Thus the order $<$ is defined such that $G' < G$ holds iff $\mathcal{A}(G') < \mathcal{A}(G)$ holds where $\mathcal{A}$ is an abstraction function. For example, $\mathcal{A}$ might be defined to map each goal G to a multiset of natural numbers, where each atom in G maps to a single number in the multiset. The approach leads to the concept of a level mapping [9].

Definition 4.1 Let P be a program. A level mapping for P is a function $|.| : B_P \mapsto \mathbb{N}$ from the Herbrand base of P to the set of natural numbers $\mathbb{N}$. For an atom $A \in B_P$, $|A|$ denotes the level of A.

The reader is referred to Lloyd [25] for the standard definitions of the Herbrand universe, Herbrand base, Herbrand interpretation, Herbrand model, clauses, substitutions, unifiers, resolvents, derivations, *etc.* Since a level mapping is defined over the Herbrand base, it is not defined for non-ground atoms. However, the concept can be straightforwardly lifted [4].

Definition 4.2 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 $[A]$.

The importance of boundedness cannot be over stressed. Since goals which are ground cannot be used to compute values, they are the exception rather than the norm in logic programming. Thus practical termination proofs must deal with non-ground goals and boundedness provides the basis for this.

Example 4.1 Let P be the program:

```
p(a, X) :- p(b, X).
p(b, a).
p(b, b).
```

The function $|.| : \{\mathsf{p(a,\ a)}, \mathsf{p(a,\ b)}, \mathsf{p(b,\ a)}, \mathsf{p(b,\ b)}\} \mapsto \mathbb{N}$ defined by $|\mathsf{p(a,\ a)}| = 34$, $|\mathsf{p(a,\ b)}| = 12$, $|\mathsf{p(b,\ a)}| = 0$ and $|\mathsf{p(b,\ b)}| = 27$ is a level mapping for P. The atom $\mathsf{p(a,\ X)}$ is bounded wrt $|.|$ since $[\mathsf{p(a,\ X)}] = \{\mathsf{p(a,\ a)}, \mathsf{p(a,\ b)}\}$ and $\max(\{|\mathsf{p(a,\ a)}|, |\mathsf{p(a,\ b)}|\}) = \max(\{34, 12\}) = 34$.

Level mappings are usually defined in terms of norms. A norm is a mapping from terms to natural numbers which provides some measure of the size of a term.

Example 4.2 The list-length norm $|.|_{list\text{-}length} : U_P \mapsto \mathbb{N}$ and the term-size norm $|.|_{term\text{-}size} : U_P \mapsto \mathbb{N}$ from the Herbrand universe to the natural numbers can be defined by

$$|t|_{list\text{-}length} = \begin{cases} 1 + |t_2|_{list\text{-}length} & \text{if } t = [t_1|t_2] \\ 0 & \text{otherwise} \end{cases}$$

$$|f(t_1, \ldots, t_n)|_{term\text{-}size} = n + \sum_{i=1}^{n} |t_i|_{term\text{-}size}$$

Then, for example, $|[\mathsf{X}, \mathsf{Y}, \mathsf{Z}]|_{list\text{-}length} = 3$ and $|\mathsf{f}(\mathsf{a}, \mathsf{g}(\mathsf{b}))|_{term\text{-}size} = 2+1 = 3$.

4.2 Atom selection

In the classic level mapping based approaches to termination [1, 4], a fundamental requirement is that only bounded atoms are selected. The reason is that, in general, when unbounded atoms are selected for resolution, it becomes more difficult to reason about the termination of the subsequent computation. This approach can still be applied when considering flexible computation rules. Moreover, delay declarations provide a mechanism to control this directly by delaying atoms until they become bounded. This idea is formally captured in the following definition which is due to Marchiori and Teusink [28, 29].

Definition 4.3 A delay declaration for a predicate p is safe wrt a level mapping $|.|$ if for every atom A with predicate symbol p, if A satisfies its delay declaration, then A is bounded wrt $|.|$.

4.3 Covers

To determine whether or not an atom is bounded when it is selected, requires a consideration of the atoms that have been (partially) resolved before the selection of the atom. The following definitions of cover, originally proposed by Marchiori and Teusink [28, 29], capture this idea. Note that in the following definition, $vars(o)$ denotes the set of variables within the syntactic object o; $body(c)$ is the set of body atoms occurring in a clause c; and $dom(\theta)$ is the set of variables which constitute the domain of a substitution θ.

Definition 4.4 Let $c = H{:-}B_1, \ldots, B_n$ be a clause and $|.|$ a level mapping. Let $A \in body(c)$ and $D \subseteq body(c)$ such that $A \notin D$. Then D is a direct cover for A wrt $|.|$ in c, if there exists a substitution θ such that:

- $\theta(A)$ is bounded wrt $|.|$ and
- $dom(\theta) \subseteq vars(H) \cup vars(D)$.

A direct cover D for A is minimal if no proper subset of D is a direct cover for A. The set of minimal direct covers of A wrt $|.|$ in c is denoted by $mdcovers_{|.|,c}(A)$.

Intuitively, a direct cover of an atom A in a clause c is a subset D of the body atoms of c such that for some instantiation θ of the variables in D, $\theta(A)$ is bounded. Note that a body atom may have zero, one or more (minimal) direct covers. In particular, an atom A will have no direct cover when, in order for A to become bounded, it is necessary to instantiate a variable of A which does not occur elsewhere in the clause. On the other hand, the atom A will have the empty set as its only minimal cover if A is bounded whenever the head of the clause is ground.

Example 4.3 Consider the quicksort program listed in figure 1 and the level mapping $|.|$ defined by:

$$|\mathsf{qsort}(x,y)| = y' + 1 \qquad |\mathsf{part}(w,x,y,z)| = y' + z' \qquad |\mathsf{append}(x,y,z)| = z'$$

where $y' = |y|_{\textit{list-length}}$ and $z' = |z|_{\textit{list-length}}$. Then

$$\begin{aligned} mdcovers_{|.|,c}(\mathsf{part}(\mathsf{Xs},\mathsf{X},\mathsf{L},\mathsf{B})) &= \{\{\mathsf{qsort}(\mathsf{L},\mathsf{Ls}), \mathsf{qsort}(\mathsf{B},\mathsf{Bs})\}\} \\ mdcovers_{|.|,c}(\mathsf{qsort}(\mathsf{L},\mathsf{Ls})) &= \{\{\mathsf{append}(\mathsf{Ls},[\mathsf{X}|\mathsf{Bs}],\mathsf{Ys})\}\} \\ mdcovers_{|.|,c}(\mathsf{qsort}(\mathsf{B},\mathsf{Bs})) &= \{\{\mathsf{append}(\mathsf{Ls},[\mathsf{X}|\mathsf{Bs}],\mathsf{Ys})\}\} \\ mdcovers_{|.|,c}(\mathsf{append}(\mathsf{Ls},[\mathsf{X}|\mathsf{Bs}],\mathsf{Ys})) &= \{\emptyset\} \end{aligned}$$

where c is the second clause of qsort. Note that $mdcovers_{|.|,c}(\mathsf{append}(\mathsf{Ls},[\mathsf{X}|\mathsf{Bs}],\mathsf{Ys})) = \{\emptyset\}$ because $\mathsf{Ys} \in vars(\mathsf{qsort}([\mathsf{X}|\mathsf{Xs}],\mathsf{Ys}))$. In this example each atom has exactly one minimal direct cover.

Marchiori and Teusink [29] explain the idea of cover in terms of "potential activators". Suppose that each predicate of the quicksort program is equipped with a delay declaration that is safe wrt the level mapping $|.|$ of example 4.3. Then the goal $\mathsf{qsort}(\mathsf{L},\mathsf{Ls})$ can only be selected when $\mathsf{qsort}(\mathsf{L},\mathsf{Ls})$ is bounded wrt $|.|$, that is, when Ls is instantiated to a rigid list. Resolution of $\mathsf{append}(\mathsf{Ls},[\mathsf{X}|\mathsf{Bs}],\mathsf{Ys})$ can potentially activate this goal by instantiating Ls thereby making $\mathsf{qsort}(\mathsf{L},\mathsf{Ls})$ selectable. The (minimal) direct cover codifies this potential producer-consumer relationship among the body atoms of a clause. It can also express multiple producer-single consumer relationships. The goal $\mathsf{part}(\mathsf{Xs}, \mathsf{X}, \mathsf{L}, \mathsf{B})$ is only selectable if it is bounded wrt $|.|$, or equivalently, when L and B are instantiated to lists of fixed length. The goals $\mathsf{qsort}(\mathsf{L}, \mathsf{Ls})$ and $\mathsf{qsort}(\mathsf{B}, \mathsf{Bs})$ are potential activators for $\mathsf{part}(\mathsf{Xs}, \mathsf{X}, \mathsf{L}, \mathsf{B})$ since, when combined, they can potentially instantiate L and B to rigid lists. Note, however, that although (minimal) direct cover encapsulates the idea of potential activation, it misses the idea that activators themselves need to be activated. This iterative application of potential activation leads to the concept of cover.

Definition 4.5 Let $c = H{:-}B_1, \ldots, B_n$ be a clause and $|.|$ a level mapping. Let $A \in \mathit{body}(c)$ and $C \subseteq \mathit{body}(c)$ such that $A \notin C$. Then C is a *cover* for A wrt $|.|$ in c, if $\langle A, C \rangle$ is an element of the least set S such that:

- $\langle A, \emptyset \rangle \in S$ if $\emptyset \in \mathit{mdcovers}_{|.|,c}(A)$ or
- $\langle A, C \rangle \in S$ if $A \notin C$, $\{A_1, \ldots, A_k\} \in \mathit{mdcovers}_{|.|,c}(A)$, $\langle A_i, C_i \rangle \in S$ for all $i \in [1, k]$ and $C = \cup_{i=1}^{k} \{A_i\} \cup C_i$.

The set of covers of A wrt $|.|$ in c is denoted by $\mathit{covers}_{|.|,c}(A)$.

Interestingly, although the cover relation is a kind of closure of the direct cover relation, it is not a transitive one; a direct cover of an atom is not necessarily a cover of that atom. This is illustrated by $\mathsf{part}(\mathsf{Xs}, \mathsf{X}, \mathsf{L}, \mathsf{B})$. The minimum direct cover specifies that this goal can only be activated by $\mathsf{qsort}(\mathsf{L}, \mathsf{Ls})$ and $\mathsf{qsort}(\mathsf{B}, \mathsf{Bs})$. However, these can only be activated by $\mathsf{append}(\mathsf{Ls}, [\mathsf{X}|\mathsf{Bs}], \mathsf{Ys})$. Hence the two $\mathsf{qsort}(\mathsf{L}, \mathsf{Ls})$ and $\mathsf{qsort}(\mathsf{B}, \mathsf{Bs})$ do not qualify as cover for $\mathsf{part}(\mathsf{Xs}, \mathsf{X}, \mathsf{L}, \mathsf{B})$; $\mathsf{part}(\mathsf{Xs}, \mathsf{X}, \mathsf{L}, \mathsf{B})$ can only be activated once $\mathsf{append}(\mathsf{Ls}, [\mathsf{X}|\mathsf{Bs}], \mathsf{Ys})$ has activated both $\mathsf{qsort}(\mathsf{L}, \mathsf{Ls})$ and $\mathsf{qsort}(\mathsf{B}, \mathsf{Bs})$.

Example 4.4 Consider the program qsort, the level mapping $|.|$ and clause c of example 4.3, and the set S as defined in definition 4.5. Then

- Since $\emptyset \in \mathit{mdcovers}_{|.|,c}(\mathsf{append}(\mathsf{Ls}, [\mathsf{X}|\mathsf{Bs}], \mathsf{Ys}))$ it follows $\langle \mathsf{append}(\mathsf{Ls}, [\mathsf{X}|\mathsf{Bs}], \mathsf{Ys}), \emptyset \rangle \in S$;
- Since $\{\mathsf{append}(\mathsf{Ls}, [\mathsf{X}|\mathsf{Bs}], \mathsf{Ys})\} \in \mathit{mdcovers}_{|.|,c}(\mathsf{qsort}(\mathsf{L}, \mathsf{Ls}))$ and $\langle \mathsf{append}(\mathsf{Ls}, [\mathsf{X}|\mathsf{Bs}], \mathsf{Ys}), \emptyset \rangle \in S$ then $\langle \mathsf{qsort}(\mathsf{L}, \mathsf{Ls}), C \rangle \in S$ where $C = \{\mathsf{append}(\mathsf{Ls}, [\mathsf{X}|\mathsf{Bs}], \mathsf{Ys})\} \cup \emptyset$;
- It follows similarly that $\langle \mathsf{qsort}(\mathsf{B}, \mathsf{Bs}), \{\mathsf{append}(\mathsf{Ls}, [\mathsf{X}|\mathsf{Bs}], \mathsf{Ys})\} \rangle \in S$;

- Finally $\langle \mathsf{part}(\mathsf{Xs},\mathsf{X},\mathsf{L},\mathsf{B}), C\rangle \in S$ where $C = (\{\mathsf{qsort}(\mathsf{L},\mathsf{Ls})\} \cup \{\mathsf{append}(\mathsf{Ls},[\mathsf{X}|\mathsf{Bs}],\mathsf{Ys})\}) \cup (\{\mathsf{qsort}(\mathsf{B},\mathsf{Bs})\} \cup \{\mathsf{append}(\mathsf{Ls},[\mathsf{X}|\mathsf{Bs}],\mathsf{Ys})\})$ because

  - $\{\mathsf{qsort}(\mathsf{L},\mathsf{Ls}), \mathsf{qsort}(\mathsf{B},\mathsf{Bs})\} \in \mathit{mdcovers}_{|.|,c}(\mathsf{part}(\mathsf{Xs},\mathsf{X},\mathsf{L},\mathsf{B}))$ and
  - $\langle \mathsf{qsort}(\mathsf{L},\mathsf{Ls}), \{\mathsf{append}(\mathsf{Ls},[\mathsf{X}|\mathsf{Bs}],\mathsf{Ys})\}\rangle \in S$ and
  - $\langle \mathsf{qsort}(\mathsf{B},\mathsf{Bs}), \{\mathsf{append}(\mathsf{Ls},[\mathsf{X}|\mathsf{Bs}],\mathsf{Ys})\}\rangle \in S$.

Hence
$$\begin{aligned}
\mathit{covers}_{|.|,c}(\mathsf{part}(\mathsf{Xs},\mathsf{X},\mathsf{L},\mathsf{B})) &= \{\{\mathsf{qsort}(\mathsf{L},\mathsf{Ls}), \mathsf{qsort}(\mathsf{B},\mathsf{Bs}),\\
&\qquad \mathsf{append}(\mathsf{Ls},[\mathsf{X}|\mathsf{Bs}],\mathsf{Ys})\}\}\\
\mathit{covers}_{|.|,c}(\mathsf{qsort}(\mathsf{L},\mathsf{Ls})) &= \{\{\mathsf{append}(\mathsf{Ls},[\mathsf{X}|\mathsf{Bs}],\mathsf{Ys})\}\}\\
\mathit{covers}_{|.|,c}(\mathsf{qsort}(\mathsf{B},\mathsf{Bs})) &= \{\{\mathsf{append}(\mathsf{Ls},[\mathsf{X}|\mathsf{Bs}],\mathsf{Ys})\}\}\\
\mathit{covers}_{|.|,c}(\mathsf{append}(\mathsf{Ls},[\mathsf{X}|\mathsf{Bs}],\mathsf{Ys})) &= \{\emptyset\}
\end{aligned}$$

4.4 Delay recurrency

Using the notion of cover, Marchiori and Teusink [28, 29] introduced the class of delay recurrent programs. It was intended that programs lying within this class would be terminating under a dynamic selection rule.

Definition 4.6 Let P be a program, $|.|$ a level mapping and I an interpretation for P. A clause $c = H{:-}B_1, \ldots, B_n$ is delay recurrent wrt $|.|$ and I iff

- I is a model for c and
- for all $i \in [1, n]$, for every cover $C \in \mathit{covers}_{|.|,c}(B_i)$ and for every grounding substitution θ for c such that $I \models \theta(C)$, it follows that $|\theta(H)| > |\theta(B_i)|$.

A program P is delay recurrent wrt $|.|$ and I iff every clause of P is delay recurrent wrt $|.|$ and I.

To be precise, definition 4.6 differs from that in [28, 29] since the original contains some slight redundancy/ambiguity. The above definition, however, reflects the intentions of Marchiori and Teusink [27].

Example 4.5 Let $|.|$ be the level mapping of example 4.3 and I the interpretation

$$\begin{array}{rcl}
\{\mathsf{qsort}(x,y) & | & |x|_{\mathit{list\text{-}length}} = |y|_{\mathit{list\text{-}length}}\} \cup\\
\{\mathsf{part}(x,w,y,z) & | & |x|_{\mathit{list\text{-}length}} = |y|_{\mathit{list\text{-}length}} + |z|_{\mathit{list\text{-}length}}\} \cup\\
\{\mathsf{append}(x,y,z) & | & |z|_{\mathit{list\text{-}length}} = |x|_{\mathit{list\text{-}length}} + |y|_{\mathit{list\text{-}length}}\}
\end{array}$$

and consider the second clause of qsort dubbed c. To check that I is a model for c let $\theta = \{\mathsf{X} \mapsto t_1,\ \mathsf{Xs} \mapsto t_2,\ \mathsf{Ys} \mapsto t_3,\ \mathsf{L} \mapsto t_4,\ \mathsf{B} \mapsto t_5,\ \mathsf{Ls} \mapsto t_6,\ \mathsf{Bs} \mapsto t_7\}$ be a grounding substitution for c such that $I \models \{\mathsf{part}(t_2, t_1, t_4, t_5), \mathsf{qsort}(t_4, t_6), \mathsf{qsort}(t_5, t_7), \mathsf{append}(t_6, [t_1|t_7], t_3)\}$. I is a model for c because

$$
\begin{aligned}
|[t_1|t_2]|_{\mathit{list\text{-}length}} &= 1 + |t_2|_{\mathit{list\text{-}length}} \\
&= 1 + (|t_4|_{\mathit{list\text{-}length}} + |t_5|_{\mathit{list\text{-}length}}) \\
&= 1 + (|t_6|_{\mathit{list\text{-}length}} + |t_7|_{\mathit{list\text{-}length}}) \\
&= |t_6|_{\mathit{list\text{-}length}} + |[t_1|t_7]|_{\mathit{list\text{-}length}} \\
&= |t_3|_{\mathit{list\text{-}length}}
\end{aligned}
$$

Now consider the body atom $A = \mathsf{part}(\mathsf{Xs}, \mathsf{X}, \mathsf{L}, \mathsf{B})$ and let $C \in \mathit{covers}_{|.|,c}(A)$. It follows that $C = \{\mathsf{qsort}(\mathsf{L}, \mathsf{Ls}), \mathsf{qsort}(\mathsf{B}, \mathsf{Bs}), \mathsf{append}(\mathsf{Ls}, [\mathsf{X}|\mathsf{Bs}], \mathsf{Ys})\}$. Consider again θ as it is a grounding substitution for c and suppose $I \models \{\mathsf{qsort}(t_4, t_6), \mathsf{qsort}(t_5, t_7), \mathsf{append}(t_6, [t_1|t_7], t_3)\}$. Then

$$
\begin{aligned}
|\mathsf{qsort}([t_1|t_2], t_3)| &= |t_3|_{\mathit{list\text{-}length}} + 1 \\
&= (|t_6|_{\mathit{list\text{-}length}} + |t_7|_{\mathit{list\text{-}length}} + 1) + 1 \\
&= (|t_4|_{\mathit{list\text{-}length}} + |t_5|_{\mathit{list\text{-}length}} + 1) + 1 \\
&> |t_4|_{\mathit{list\text{-}length}} + |t_5|_{\mathit{list\text{-}length}} \\
&= |\mathsf{part}(t_2, t_1, t_4, t_5)|
\end{aligned}
$$

It is straightforward to check that condition 2 of definition 4.6 holds for every other body atom of c. Hence c is delay recurrent wrt $|.|$ and I.

The intention behind the definition of delay recurrency is that a delay recurrent program P, when augmented with a set of safe delay declarations for the predicates of P, only admits finite derivations. The delay declarations handle the local boundedness issue, but there is still the global boundedness issue to consider. Suppose C is a cover for an atom B in a delay recurrent clause, and θ is a partial answer substitution for C such that $\theta(B)$ is bounded (note that θ may not necessarily be a *correct* answer substitution since the atoms in C have not yet been fully resolved). At this point θ speculatively binds the variables of B since it is not yet known whether or not there exists some substitution σ such that $I \models \sigma(\theta(C))$. If $\theta(B)$ is selected at this point an infinite computation may arise since there is no guarantee that the level of the head is greater than the level of $\theta(B)$. Instead, by fully resolving each atom in C such that a correct answer substitution θ is obtained, $\theta(B)$ can be selected safely since $I \models \sigma(\theta(C))$ for all σ, whence by condition 2 of delay recurrency, the level of $\theta(B)$ is less than the level of the head. Full resolution of C can be achieved by using a local selection rule. To reiterate, 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 [46]. The notion is formally defined below.

Definition 4.7 For a goal $G = A_1, \ldots, A_m$ the ith atom of G is A_i. Let $G_0, G_1, G_2, \ldots, G_k, \ldots$ be a derivation. The age of the ith atom in G_k, denoted $age_{G_k}(i)$, is defined as follows:

- If $G_0 = A_1, \ldots, A_m$, then $age_{G_0}(i) = 0$ for all $i \in [1, m]$;
- If $G_{k+1} = \theta(A_1, \ldots, A_{s-1}, B_1, \ldots, B_n, A_{s+1}, \ldots, A_m)$ and $G_k = A_1, \ldots, A_m$, then

$$age_{G_{k+1}}(i) = \begin{cases} age_{G_k}(i) + 1 & \text{for all } i \in [1, s-1] \\ 0 & \text{for all } i \in [s, s+n-1] \\ age_{G_k}(i - n + 1) + 1 & \text{for all } i \in [s+n, n+m-1] \end{cases}$$

For a goal $G = A_1, \ldots, A_m$, an atom A_i is introduced in G if $age_G(A) = 0$. An atom A_i is most recently introduced in G iff $age_G(i) \leq age_G(j)$ for all $j \in [1, m]$.

Definition 4.8 A derivation $G_0, G_1, G_2, \ldots, G_k, \ldots$ applies local selection iff whenever an atom A_i is selected for resolution from G_k then A_i is most recently introduced in G_k.

Definition 4.9 A derivation $G_0, G_1, G_2, \ldots, G_k, \ldots$ satisfies the delay declarations of P iff whenever an atom A_i is selected for resolution from G_k then A_i satisfies the delay declarations of P.

The main result regarding delay recurrent programs [28, 29] can now be stated:

Theorem 4.1 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

- P is delay recurrent wrt $|.|$ and I, and
- the delay declarations for P are safe wrt $|.|$

Then every derivation that satisfies the delay declarations of P, whilst applying local selection, is finite.

4.5 Semi-delay recurrency

Marchiori and Teusink [28, 29] noticed that boundedness of atoms could be enforced by using delay declarations but did not fully exploit this fact. Their definition requires a decrease in the level mapping from the head to the non-recursive body atoms, when in fact the boundedness of selected atoms is already guaranteed by safe delay declarations. To relax this restriction, notion of predicate dependency is introduced to distinguish between recursive and non-recursive body atoms.

Definition 4.10 Let $p, q \in \Pi$ where Π is the set of predicate symbols in a logic program P. Then p directly-depends-on q iff $p(t_1, \ldots, t_k){:-}B_1, \ldots, B_n \in P$ and $B_i = q(s_1, \ldots, s_l)$ for some $i \in [1, n]$. The depends-on relation, denoted $\sqsupseteq$, is defined as the reflexive, transitive closure of the directly-depends-on relation. If $p \sqsupseteq q$ and $q \sqsupseteq p$ then p and q are mutually dependent and this is denoted by $p \simeq q$.

Example 4.6 *Returning to the quicksort program listed in figure 1,* qsort $\sqsupseteq$ part*,* qsort $\sqsupseteq$ append*,* part $\sqsupseteq$ leq*,* part $\sqsupseteq$ gt*,* qsort $\sqsupseteq$ leq *and* qsort $\sqsupseteq$ gt*. For this program, $p \simeq q$ iff $p = q$.*

The following new definition will prove useful for defining a large class of terminating programs which permit coroutining. In what follows $rel(A)$ denotes the predicate symbol of the atom A.

Definition 4.11 Let $|.|$ be a level mapping and I an interpretation for a program P. A clause $c = H{:-}B_1, \ldots, B_n$ is semi-delay recurrent wrt $|.|$ and I iff

- I is a model for c and
- for all $i \in [1, n]$ such that $rel(H) \simeq rel(B_i)$, for every cover $C \in covers_{|.|,c}(B_i)$ and for every grounding substitution θ for c such that $I \models \theta(C)$, it follows that $|\theta(H)| > |\theta(B_i)|$.

A program P is semi-delay recurrent wrt $|.|$ and I iff every clause of P is semi-delay recurrent wrt $|.|$ and I.

Observe in this definition that there is no restriction placed on the relation between the level of the head of the clause and the level of the non-recursive body atoms.

Example 4.7 Let I be the interpretation and $|.|$ the level mapping of example 4.5. As before, I is a model for the second clause c of the qsort predicate. Now $\mathit{rel}(\mathsf{qsort}([\mathsf{X}|\mathsf{Xs}], \mathsf{Ys})) \simeq \mathit{rel}(\mathsf{qsort}(\mathsf{L}, \mathsf{Ls}))$ and $\mathit{rel}(\mathsf{qsort}([\mathsf{X}|\mathsf{Xs}], \mathsf{Ys})) \simeq \mathit{rel}(\mathsf{qsort}(\mathsf{B}, \mathsf{Bs}))$. Then

$$\begin{array}{rcl} \mathit{covers}_{|.|,c}(\mathsf{qsort}(\mathsf{L}, \mathsf{Ls})) & = & \{\{\mathsf{append}(\mathsf{Ls}, [\mathsf{X}|\mathsf{Bs}], \mathsf{Ys})\}\} \\ \mathit{covers}_{|.|,c}(\mathsf{qsort}(\mathsf{B}, \mathsf{Bs})) & = & \{\{\mathsf{append}(\mathsf{Ls}, [\mathsf{X}|\mathsf{Bs}], \mathsf{Ys})\}\} \end{array}$$

Let $\theta = \{\mathsf{X} \mapsto t_1,\ \mathsf{Xs} \mapsto t_2,\ \mathsf{Ys} \mapsto t_3,\ \mathsf{L} \mapsto t_4,\ \mathsf{B} \mapsto t_5,\ \mathsf{Ls} \mapsto t_6,\ \mathsf{Bs} \mapsto t_7\}$ be a grounding substitution for c such that $I \models \{\mathsf{append}(t_6, [t_1|t_7], t_3)\}$. Then

$$\begin{aligned} |\mathsf{qsort}([t_1|t_2], t_3)| &= |t_3|_{\mathit{list\text{-}length}} + 1 \\ &= (|t_6|_{\mathit{list\text{-}length}} + |t_7|_{\mathit{list\text{-}length}} + 1) + 1 \\ &> |t_6|_{\mathit{list\text{-}length}} + 1 \\ &= |\mathsf{qsort}(t_4, t_6)| \end{aligned}$$

$$\begin{aligned} |\mathsf{qsort}([t_1|t_2], t_3)| &= |t_3|_{\mathit{list\text{-}length}} + 1 \\ &> |t_7|_{\mathit{list\text{-}length}} + 1 \\ &= |\mathsf{qsort}(t_5, t_7)| \end{aligned}$$

Hence c is semi-delay recurrent wrt $|.|$ and I.

It would be straightforward to prove that theorem 4.1 still holds if the program is replaced by one which is semi-delay recurrent, but a more significant result can be obtained. Observe that a local selection rule is used to ensure that a cover of an atom is completely resolved before the atom itself is selected. Notice, however, that for semi-delay recurrency, it is only necessary for the covers of the mutually recursive atoms 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, the notion of covering is lifted from the clause level to the goal level. A covering of a recursive atom A in a goal G, is the set of atoms in G which have yet to be resolved before A can be safely selected. An atom A may have more than one covering, though it is only necessary to fully resolve the atoms of one of them before the selection of A. Coverings of atoms will change during the course of a derivation as new atoms are introduced and others are fully resolved.

Definition 4.12 Let $G_0, G_1, G_2, \ldots$ be a derivation and $|.|$ a level mapping. A covering C for an atom A in a goal G_l wrt $|.|$, denoted $C \in \mathit{covers}_{|.|,G_l}(A)$, is defined as follows:

- Suppose $G_0 = A_1, \ldots, A_m$. Then $\emptyset \in \mathit{covers}_{|.|,G_0}(A_i)$ for all $i \in [1, m]$.
- Suppose $G_{l+1} = \theta(A_1, \ldots, A_{s-1}, B_1, \ldots, B_n, A_{s+1}, \ldots, A_m)$ is the resolvent derived from $G_l = A_1, \ldots, A_s, \ldots, A_m$ and $c = H{:-}B_1, \ldots, B_n$, where A_s is the selected atom in G_l and $\theta \in \mathit{mgu}(H, A_s)$. Then
  - for all $i \in [1, n]$,
    - if $\mathit{rel}(H) \simeq \mathit{rel}(B_i)$ and $C \in \mathit{covers}_{|.|,c}(B_i)$ then $\theta(C) \in \mathit{covers}_{|.|,G_{l+1}}(\theta(B_i))$;
    - if $\mathit{rel}(H) \not\simeq \mathit{rel}(B_i)$ then $\emptyset \in \mathit{covers}_{|.|,G_{l+1}}(\theta(B_i))$.
  - for all $i \in [1, m]$, if $i \neq s$, $C \subseteq \{A_1, \ldots, A_{i-1}, A_{i+1}, \ldots, A_m\}$ and $C \in \mathit{covers}_{|.|,G_l}(A_i)$ then
    - if $A_s \in C$ then $\theta(C) \backslash \{\theta(A_s)\} \cup \theta(\{B_1, \ldots, B_n\}) \in \mathit{covers}_{|.|,G_{l+1}}(\theta(A_i))$;
    - if $A_s \notin C$ then $\theta(C) \in \mathit{covers}_{|.|,G_{l+1}}(\theta(A_i))$.

Definition 4.13 Let $|.|$ be a level mapping. A derivation $G_0, G_1, G_2, \ldots, G_k, \ldots$ applies semi-local selection wrt $|.|$ iff whenever an atom A_i is selected for resolution from G_k then $\emptyset \in \mathit{covers}_{|.|,G_k}(A_i)$.

Observe that, if an atom is selectable under a local selection rule, then it is selectable under a semi-local selection rule. The main result can now be stated:

Theorem 4.2 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

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

Then every derivation that satisfies the delay declarations of P, whilst applying semi-local selection wrt $|.|$, is finite.

The termination argument is constructed in terms of a multiset ordering that is specific to the program. The multiset ordering $\prec_{mul}$ is itself formulated in terms of an ordering $\prec$ on $\Pi \times \mathbb{N}$. The binary relation $\prec$ is defined $\langle p, m\rangle \prec \langle q, n\rangle$ iff ($p \simeq q$ and $m < n$) or ($p \not\simeq q$ and $q \sqsupset p$). The multiset ordering is defined by $m_1 \prec_{mul} m_2$ iff there exists $\langle p_1, m_1\rangle, \ldots, \langle p_k, m_k\rangle \in m_1$ and $\langle p, m\rangle \in m_2$ such that $m_1 = (m_2 / \{\!\{\langle p, m\rangle\}\!\}) \cup \{\!\{\langle p_1, m_1\rangle \ldots, \langle p_k, m_k\rangle\}\!\}$

and $\langle p_i, m_i \rangle \prec \langle p, m \rangle$ for all $i \in [1, k]$. Since Π is finite and $\mathbb{N}$ is well-ordered, it follows that $\prec$ is well-ordered, hence $\prec_{mul}$ is well-ordered [43]. It then remains to specify a mapping from each goal of a derivation to multiset such that a decrease can be observed between multisets of consecutive goals. Such a mapping is given below:

Definition 4.14 Let $G_0, G_1, G_2, \ldots$ be a derivation, $|.|$ be a level mapping, I an interpretation and $G_k = A_1, \ldots, A_n$ be a goal. For all $i \in [1, n]$ define

$$|[G_k]_I^i| = \left\{ |\theta(A_i)| + 1 \;\middle|\; \begin{array}{l} \theta \text{ is a grounding substitution for } G_k \\ C \in \mathit{covers}_{|.|,G_k}(A_i) \\ I \models \theta(C) \end{array} \right\}$$

If $|[G]_I^i|$ is finite for each $i \in [1, n]$ then the multiset $|[G]_I|$ is defined

$$|[G]_I| = \{\!\{ \langle \mathit{rel}(A_1), \max |[G]_I^1| \rangle, \ldots, \langle \mathit{rel}(A_n), \max |[G]_I^n| \rangle \}\!\}$$

where $\max \emptyset = 0$ and $\max\{n_1, \ldots, n_k\} = n_i$ where $n_i \geq n_j$ for all $j \in [1, k]$

Lemma 4.1 Let $|.|$ be a level mapping and let $G_0, G_1, G_2, \ldots, G_k, G_{k+1}$ be a (partial) derivation that satisfies the delay declarations of P, whilst applying semi-local selection wrt $|.|$. Let I be an interpretation for program P and suppose P is semi-delay recurrent wrt $|.|$ and I. Then $|[G_{k+1}]_I| \prec_{mul} |[G_k]_I|$.

Proof 4.1 For brevity, let $G = G_k$ and $G' = G_{k+1}$. Then $G' = \theta(A_1, \ldots, A_{s-1}, B_1, \ldots, B_n, A_{s+1}, \ldots, A_m)$ is the resolvent derived from $G = A_1, \ldots, A_s, \ldots, A_m$ and $c = H{:-}B_1, \ldots, B_n$ where A_s is the selected atom in G and $\theta \in \mathit{mgu}(A_s, H)$.

- To show $\langle \mathit{rel}(\theta(B_i)), \max |[G']_I^{s+i-1}| \rangle \prec \langle \mathit{rel}(A_s), \max |[G]_I^s| \rangle$ for all $i \in [1, n]$. Let $i \in [1, n]$. If $\mathit{rel}(B_i) \not\simeq \mathit{rel}(H) \simeq \mathit{rel}(A_s)$ then $\mathit{rel}(B_i) \sqsupseteq \mathit{rel}(A_s)$ and the result follows. Now suppose $\mathit{rel}(B_i) \simeq \mathit{rel}(H)$. By definition 4.13, $\emptyset \in \mathit{covers}_{|.|,G}(A_s)$. By definition 4.14 it follows that $\max |[G]_I^s| = \max\{|\sigma(A_s)| + 1 \mid \sigma \text{ is a grounding substitution for } G\}$. Since the derivation satisfies the delay declarations of P, A_s is bounded wrt $|.|$, hence $\max |[G]_I^s| = |[A_s]| + 1$, whence $|[A_s]| \geq |[\theta(A_s)]| = |[\theta(H)]|$. Thus $\max |[G]_I^s| > |[\theta(H)]|$. By definition 4.14 it follows that

$$\begin{aligned} |[G']_I^{s+i-1}| &= |[\theta(A_1, \ldots, A_{s-1}, B_1, \ldots, B_n, A_{s+1}, \ldots, A_m)]_I^{s+i-1}| \\ &= \left\{ |\sigma(\theta(B_i))| + 1 \;\middle|\; \begin{array}{l} \sigma \text{ is a grounding substitution for } G' \\ D \in \mathit{covers}_{|.|,G'}(\theta(B_i)) \\ I \models \sigma(D) \end{array} \right\} \end{aligned}$$

- Suppose $|[G']_I^{s+i-1}| \neq \emptyset$. Then there exists (1) a grounding substitution σ for G' and (2) a cover $D \in covers_{|.|,G'}(\theta(B_i))$ such that (3) $I \models \sigma(D)$ and (4) $|\sigma(\theta(B_i))| + 1 = \max |[G']_I^{s+i-1}|$. From (2) it follows that there exists a cover $C \in covers_{|.|,c}(B_i)$ such that $D = \theta(C)$, hence by (3), $I \models \sigma(\theta(C))$. By (1), $\sigma \circ \theta$ is a grounding substitution for $B_1, \ldots, B_n$. Let κ be a grounding substitution for $\sigma(\theta(H))$. Then $\kappa \circ \sigma \circ \theta$ is a grounding substitution for c such that $I \models \kappa(\sigma(\theta(C)))$ since $\kappa(\sigma(\theta(C))) = \sigma(\theta(C))$. Hence, by the semi-delay recurrency of c, $|\kappa(\sigma(\theta(H))| > |\kappa(\sigma(\theta(B_i))| = |\sigma(\theta(B_i))|$. Thus $\max |[G']_I^{s+i-1}| \leq |\kappa(\sigma(\theta(H)))| = |[\kappa(\sigma(\theta(H)))]| \leq |[\theta(H)]|$. It follows $\max |[G]_I^s| > \max |G'|_I^{s+i-1}$.
- Suppose $|[G']_I^{s+i-1}| = \emptyset$. Then $\max |[G']_I^{s+i-1}| = 0$. But $\max |[G]_I^s| > 0$ so $\max |[G]_I^s| > \max |[G']_I^{s+i-1}|$.

- To show $\langle rel(\theta(A_i)), \max |[G']_I^i|\rangle \preceq \langle rel(A_i), \max |[G]_I^i|\rangle$ for all $i \in [1, s-1]$. Let $i \in [1, s-1]$.
  - Suppose $|[G']_I^i| \neq \emptyset$. Then there exists (5) a grounding substitution σ for G' and (6) a covering $D \in covers_{|.|,G'}(\theta(A_i))$ such that (7) $I \models \sigma(D)$ and (8) $|\sigma(\theta(A_i))| + 1 = \max |[G']_I^i|$. From (6) it follows that there exists a $C \subseteq \{A_1, \ldots, A_{i-1}, A_{i+1}, \ldots, A_m\}$ such that $C \in covers_{|.|,G}(A_i)$ and either:
    * $A_s \in C$ and $D = \theta(C) \setminus \{\theta(A_s)\} \cup \theta(\{B_1, \ldots, B_n\})$ or
    * $A_s \notin C$ and $D = \theta(C)$.

    In the first case, by (7) $I \models \sigma(\theta(C) \setminus \{\theta(A_s)\} \cup \theta(\{B_1, \ldots, B_n\})$ and consequently, since I is a model for c, $I \models \sigma(\theta(C) \setminus \{\theta(A_s)\} \cup \{\theta(H)\})$. But then $I \models \sigma(\theta(C))$ because $\theta(H) = \theta(A_s)$. In the second case, by (7), $I \models \sigma(\theta(C))$.

    By (1) $\sigma \circ \theta$ is a grounding substitution for $A_1, \ldots, A_{s-1}, A_{s+1}, \ldots, A_m$. Let κ be a grounding substitution for $\sigma(\theta(A_s))$. Then $\kappa \circ \sigma \circ \theta$ is a grounding substitution for G such that $I \models \kappa(\sigma(\theta(C)))$ since $\kappa(\sigma(\theta(C))) = \sigma(\theta(C))$. Hence $|\kappa(\sigma(\theta(A_i)))| + 1 \in |[G]_I^i|$ and as a result $\max |[G']_I^i| \leq \max |[G]_I^i|$.
  - Suppose $|[G']_I^i| = \emptyset$. Then $\max |[G']_I^i| = 0 \leq max|[G]_I^i|$.

- To show $\langle rel(\theta(A_i)), \max |[G']_I^{i+n-1}|\rangle \preceq \langle rel(A_i), \max |[G]_I^i|\rangle$ for all $i \in [s+1, m]$. This is analogous to the previous case.

The proof of theorem 4.2 then follows as immediately consequence of lemma 4.1 since $\prec_{mul}$ is well-ordered. This completes the termination argument for semi-delay recurrent programs.

5 The transformation: termination, soundness and completeness

The force of theorem 4.2 is that it provides a pathway for developing a program transformation, which is able to derive a correct and an efficient program from a logical specification. The idea is to transform a given program into one which is semi-delay recurrent, but with equivalent declarative semantics. Then, by adding safe delay declarations, a program is obtained which terminates for all goals using semi-local selection.

The transform is defined at two levels: a transform that separately acts on each clause in the original program, to yield a clause that is semi-delay recurrent; and a transform that is applied to each predicate in the original program, to produce an additional new clause. The clause-level transform replaces each clause defining a predicate $p \in \Pi$, with another defining a predicate $p^{sdr} \in \Pi^{sdr}$. The predicate-level transform, when acting on a predicate p, introduces a clause that defines p in terms of p^{sdr} and an auxiliary predicate $p^{depth} \in \Pi^{depth}$. The symbol sets Π, Π^{sdr} and Π^{depth} are required to be disjoint, that is, $\Pi \cap \Pi^{sdr} = \Pi \cap \Pi^{depth} = \Pi^{sdr} \cap \Pi^{depth} = \emptyset$. In order to define the transform, it is convenient to represent depth counters (numbers) as successor terms, that is, encode 0, 1, 2 and arbitrary k by the terms $\mathsf{0}$, $\mathsf{succ}(\mathsf{0})$, $\mathsf{succ}(\mathsf{succ}(\mathsf{0}))$ and $\mathsf{succ}^k(0)$. Using this representation, the clause-level semi-delay recurrent (SDR) transform is defined as follows:

Definition 5.1 If $c = H \text{ :- } B_1, \ldots, B_n \in P$ then $sdr(c) = p^{sdr}(\vec{t}, d) \text{ :- } B'$, $B'_1, \ldots, B'_n$ where

$$B' = \begin{cases} \mathsf{true} \text{ if } \wedge_{i=1}^{n} H \not\simeq B_i \\ d = \mathsf{succ}(d') \text{ otherwise} \end{cases} \qquad B'_i = \begin{cases} B_i \text{ if } H \not\simeq B_i \\ p_i^{sdr}(\vec{t_i}, d') \text{ otherwise} \end{cases}$$

$H = p(\vec{t})$, $B_i = p_i(\vec{t_i})$ and the variables d, d' are fresh, that is, $\{d, d'\} \cap vars(c) = \emptyset$.

The predicate true always succeeds; it is a no-operation that is used to simplify the definition of the transform. The unification $\mathsf{d} = \mathsf{succ}(\mathsf{d}')$ decrements d to obtain d'. It succeeds only if d equals or exceeds $\mathsf{succ}(\mathsf{0})$. For purposes of efficiency, this operation is actually realised as two separate operations: a test and a decrement operating on conventional integers. The predicate-level transform is defined by:

Definition 5.2 If $p \in \Pi$ then

$$sdr(p) = p(x_1, \ldots, x_k) \text{ :- } p^{depth}(x_1, \ldots, x_k, d), p^{sdr}(x_1, \ldots, x_k, d)$$

where the variables $x_1, \ldots, x_k, d$ are distinct and k is the arity of p.

Together, the sdr transforms define the set of clauses $P^{sdr} = \{sdr(c) \mid c \in P\} \cup \{sdr(p) \mid p \in \Pi\}$. The transformation is completed by augmenting P^{sdr} with a set of clauses P^{depth} that specify the predicates within Π^{depth}. The intuition is that each predicate $p^{depth}(x_1, \ldots, x_k, d)$ calculates a bound d, from the arguments $x_1, \ldots, x_k$, that controls the depth to which the predicate p^{sdr} is searched. For the program $P^{sdr} \cup P^{depth}$ to yield finite derivations under semi-local selection, the clauses of Π^{depth} are required to be semi-delay recurrent. The termination issue is consider further in section 5.1. For the program $P^{sdr} \cup P^{depth}$ to be correct relative to P, it must preserve each answer that P can produce (it must be complete relative to P) whilst not introducing additional answers (it must be sound relative to P). These correctness issues are addressed in section 5.2; in particular the section specifies criteria on the predicates of P^{depth} that ensure completeness (soundness follows almost immediately). The remainder of this section illustrates the transform.

Example 5.1 Consider again the quicksort program of figure 1, denoting the i^{th} clauses of predicates qsort and append by qsort_i and append_i respectively. Then

$$
\begin{aligned}
sdr(\mathsf{qsort}_1) &= \mathsf{qsort}^{sdr}([], [], \mathsf{D}) \;\text{:-}\; \mathsf{true}. \\
sdr(\mathsf{qsort}_2) &= \mathsf{qsort}^{sdr}([\mathsf{X}|\mathsf{Xs}], \mathsf{Ys}, \mathsf{D}) \;\text{:-}\; \mathsf{D} = \mathsf{succ}(\mathsf{D1}), \\
&\qquad \mathsf{part}(\mathsf{Xs}, \mathsf{X}, \mathsf{L}, \mathsf{B}), \mathsf{qsort}^{sdr}(\mathsf{L}, \mathsf{Ls}, \mathsf{D1}), \mathsf{qsort}^{sdr}(\mathsf{B}, \mathsf{Bs}, \mathsf{D1}), \\
&\qquad \mathsf{append}(\mathsf{Ls}, [\mathsf{X}|\mathsf{Bs}], \mathsf{Ys}) \\
sdr(\mathsf{append}_1) &= \mathsf{append}^{sdr}([], \mathsf{X}, \mathsf{X}, \mathsf{D}) \;\text{:-}\; \mathsf{true}. \\
sdr(\mathsf{append}_2) &= \mathsf{append}^{sdr}([\mathsf{X}|\mathsf{Xs}], \mathsf{Ys}, [\mathsf{X}|\mathsf{Zs}], \mathsf{D}) \;\text{:-}\; \mathsf{D} = \mathsf{succ}(\mathsf{D1}), \\
&\qquad \mathsf{append}^{sdr}(\mathsf{Xs}, \mathsf{Ys}, \mathsf{Zs}, \mathsf{D1}).
\end{aligned}
$$

$$
\begin{aligned}
sdr(\mathsf{qsort}) &= \mathsf{qsort}(\mathsf{X}, \mathsf{Y}) \;\text{:-}\; \mathsf{qsort}^{depth}(\mathsf{X}, \mathsf{Y}, \mathsf{D}), \mathsf{qsort}^{sdr}(\mathsf{X}, \mathsf{Y}, \mathsf{D}) \\
sdr(\mathsf{append}) &= \mathsf{append}(\mathsf{X}, \mathsf{Y}, \mathsf{Z}) \;\text{:-}\; \mathsf{append}^{depth}(\mathsf{X}, \mathsf{Y}, \mathsf{Z}, \mathsf{D}), \mathsf{append}^{sdr}(\mathsf{X}, \mathsf{Y}, \mathsf{Z}, \mathsf{D})
\end{aligned}
$$

5.1 The termination issue

The following theorem addresses the termination issue, by showing that the semi-delay recurrency of P^{depth} is sufficient to ensure that $P^{sdr} \cup P^{depth}$ is also semi-delay recurrent. The value of this result is that theorem 4.2 guarantees that the program $P^{sdr} \cup P^{depth}$ will yield finite derivations under semi-local selection, providing that it is equipped with delay declarations that are safe.

Theorem 5.1 Let I be a model of $P^{sdr} \cup P^{depth}$ and suppose P^{depth} is semi-delay recurrent wrt a level mapping $|.|_1$ and I. Then $P^{sdr} \cup P^{depth}$ is semi-delay recurrent wrt $|.|_2$ and I where

$$|p^{sdr}(\vec{t}, t')|_2 = |t'|_{term\text{-}size} \qquad |p(\vec{t})|_2 = 0 \qquad |p^{depth}(\vec{t}, t')|_2 = |p^{depth}(\vec{t}, t')|_1$$

Thus the program $P^{sdr} \cup P^{depth}$ will be terminating for all goals under semi-local selection if, for each predicate, a delay declaration is added which is safe wrt $|.|_2$. In fact, since the goals $p(\vec{t})$ are always bounded wrt $|.|_2$, delay declarations need only be supplied for the p^{sdr} and p^{depth} predicates.

Proof 5.1 Let $\simeq$ be the mutual dependence relation induced from $P^{sdr} \cup P^{depth}$. Since P^{depth} is semi-delay recurrent wrt $|.|_1$ and I, it remains to consider each $c \in P^{sdr}$:

- Suppose $c = p(\vec{x}) \text{ :- } p^{depth}(\vec{x}, d), p^{sdr}(\vec{x}, d)$. Then c is semi-delay recurrent wrt $|.|_2$ and I since $p \not\simeq p^{depth}$ and $p \not\simeq p^{sdr}$.

- Suppose $c = p^{sdr}(\vec{t}, d) \text{ :- } \mathsf{true}, B_1, \ldots, B_n$ and $p \not\simeq \mathit{rel}(B_1), \ldots, p \not\simeq \mathit{rel}(B_n)$. Then again c is trivially semi-delay recurrent wrt $|.|_2$ and I.

- Suppose $c = p^{sdr}(\vec{t}, d) \text{ :- } d = \mathsf{succ}(d'), B'_1, \ldots, B'_n$. Let $i \in [1, n]$ such that $B'_i = p_i^{sdr}(\vec{t}, d')$, that is, $p \simeq p_i$. Then $\mathit{mdcovers}_{|.|_2,c}(B'_i) = \{d = \mathsf{succ}(d')\}$ and $\mathit{mdcovers}_{|.|_2,c}(d = \mathsf{succ}(d')) = \{\emptyset\}$, hence $\mathit{covers}_{|.|_2,c}(B'_i) = \{d = \mathsf{succ}(d')\}$. Let θ be a grounding substitution for c such that $I \models \theta(d = \mathsf{succ}(d'))$. Hence $\theta(d) = \theta(\mathsf{succ}(d'))$, thus $|\theta(p^{sdr}(\vec{t}, d))|_2 = |\theta(d)|_{term\text{-}size} = |\theta(\mathsf{succ}(d'))|_{term\text{-}size} = 1 + |\theta(d')|_{term\text{-}size} > |\theta(d')|_{term\text{-}size} = |\theta(p_i^{sdr}(\vec{t}, d'))|_2$.

The proof of theorem 5.1 exploits the property that $d = \mathsf{succ}(d')$ is the only cover of each recursive body atom $B'_i = p_i^{sdr}(\vec{t}, d')$ occurring within the body of a clause defining p^{sdr}. Thus, once $d = \mathsf{succ}(d')$ is resolved, then the remaining goals $B'_1, \ldots, B'_n$ can be scheduled in any order. Thus coroutining can be liberally applied.

The main consequence of theorem 5.1 is that any program can be transformed into one which is semi-delay recurrent and whose semantics are equivalent in a sense which will be examined shortly.

5.2 The issues of soundness and completeness

The transform is completed by providing clauses P^{depth} for the predicates Π^{depth}. These clauses compute depth bounds on the subsequent computation, for example, the rôle of the qsort^{depth} predicate is to establish a bound on the depth of the recursive calls to qsort^{sdr}. The bound needs to be large enough to enable the search space to be probed to sufficient depth to find all the answers to a query. For example, the goal $\mathsf{qsort}([1,2,3],\mathsf{L})$ will fail if $\mathsf{qsort}^{depth}([1,2,3],\mathsf{L},\mathsf{D})$ instantiates D to $\mathsf{succ}(0)$, since more than one recursive call to the predicate qsort^{sdr} is required to find the unique answer $\mathsf{L} = [1,2,3]$. On the other hand, the goal $\mathsf{qsort}([1,2,3],\mathsf{L})$ will not produce an incorrect answer. Thus the depth bound effects the completeness of the transform rather than soundness. Nevertheless, to ensure that completeness is preserved, the definition of qsort^{depth} needs to be suitably constrained. The constraint manifests itself as a condition for completeness in the equivalence theorem that is stated below.

Theorem 5.2 Let I and J be minimal models for P and $P^{sdr} \cup P^{depth}$.

- Then if $J \models p(\vec{t})$ and $p \in \Pi$ it follows that $I \models p(\vec{t})$ (soundness of the transform).
- Suppose that whenever $J \models p^{sdr}(\vec{t}, d)$ there exists $k \geq 0$ such that $J \models p^{depth}(\vec{t}, \mathsf{succ}^k(d))$. Then if $I \models p(\vec{t})$ it follows that $J \models p(\vec{t})$ (completeness of the transform).

The criteria for completeness is a statement about the relative sizes of the depth parameters of p^{sdr} and p^{depth}. The theorem does not require these depth parameters to coincide; instead it is sufficient for the depth parameter of p^{depth} to match or exceed that of p^{sdr}. This is a useful result from a practical point of view since since upper bounds are easier to compute than exact bounds.

In order to reason about the equivalence of the minimal models I and J, and in particular prove theorem 5.2, the standard T_P [25] is introduced since its iterative formulation provides a way of constructing an inductive equivalence argument.

Definition 5.3

$$T_P(I) = \left\{ \theta(H) \;\middle|\; \begin{array}{ll} c = H{:-}B_1, \ldots, B_n \in P & \wedge \\ \theta \textit{ is a grounding substitution for } c & \wedge \\ \theta(B_1), \ldots, \theta(B_n) \in I & \end{array} \right\}$$

It turns out that the soundness argument is more involved than the completeness argument. Since $P^{sdr} \cup P^{depth}$ defines intermediate predicates Π^{sdr} that do not appear in P, if $T_P^k(\emptyset) \models p(\vec{t})$ it does not necessarily follow that $T^k_{P^{sdr} \cup P^{depth}}(\emptyset) \models p(\vec{t})$; in general, a larger value $k' > k$ is required for

$T^{k'}_{P^{sdr} \cup P^{depth}}(\emptyset) \models p(\vec{t})$. To construct an inductive hypothesis for soundness, and specifically define k' in terms of k, it is necessary to reason about the relative positions (or *level*) of predicates with the call graph hierarchy of P. This requirement leads to the following definition.

Definition 5.4 Suppose that $\simeq$ and $\sqsupseteq$ are the relations over $\Pi = \{p_1, \ldots, p_n\}$ induced from P. Then define $level(p_i) = \min\{l_i \mid \langle l_1, \ldots, l_n \rangle \in level(\Pi)\}$ where

$$level(\Pi) = \left\{ \langle l_1, \ldots, l_n \rangle \in \mathbb{N}^n \;\middle|\; \begin{array}{ll} p_j \simeq p_k \rightarrow l_j = l_k & \wedge \\ p_j \not\simeq p_k \wedge p_j \sqsupseteq p_k \rightarrow l_j > l_k & \end{array} \right\}$$

Example 5.2 Returning to the quicksort program of figure 1 and example 4.6, $\Pi = \{\mathsf{qsort}, \mathsf{part}, \mathsf{append}, \mathsf{leq}, \mathsf{gt}\}$ and using the notation in definition 5.4,

$$level(\Pi) = \left\{ \langle l_1, \ldots, l_5 \rangle \in \mathbb{N}^5 \;\middle|\; \begin{array}{l} l_1 > l_2 \wedge l_1 > l_3 \wedge l_1 > l_4 \wedge l_1 > l_5 \wedge \\ l_2 > l_4 \wedge l_2 > l_5 \end{array} \right\}$$

hence $level(\mathsf{leq}) = 1$, $level(\mathsf{gt}) = 1$, $level(\mathsf{append}) = 1$, $level(\mathsf{part}) = 2$ and $level(\mathsf{qsort}) = 3$.

The proof for theorem 5.2 (which follows immediately after that of lemma 5.2) is constructed in three stages. Firstly, lemma 5.1 states a monotonicity condition on p^{sdr} (its proof is straightforward and thus is not included). Secondly, lemma 5.1 is then used with the *level* machinery to argue a form of equivalence between T_P and $T_{P^{sdr} \cup P^{depth}}$. This gives lemma 5.2. Thirdly, the equivalence result of lemma 5.2 is then used to establish theorem 5.2 in conjunction with the property that the minimal model of a program P coincides with the limit of T_P. Interestingly, this final stage of proof requires that the clauses defined in P^{delay} to not invoke any predicates defined within P^{sdr}, which is a reasonable requirement. This construction completes the argument that the declarative semantics of the original and transformed programs (restricted to the predicates defined in the original program) coincide.

Lemma 5.1 If $T^k_{P^{sdr}}(\emptyset) \models p^{sdr}(\vec{t}, d)$ then $T^k_{P^{sdr}}(\emptyset) \models p^{sdr}(\vec{t}, \mathsf{succ}(d))$ for all $p \in \Pi$ and for all $k \geq 0$.

Lemma 5.2 Let J and M be minimal models for $P^{sdr} \cup P^{depth}$ and P^{depth}.

- Then if $T^k_P(\emptyset) \models p(\vec{t})$ it follows that

- $T^{k+level(p)}_{P^{sdr}}(M) \models p^{sdr}(\vec{t}, \mathsf{succ}^l(0))$ for some $l \geq 0$;
- $T^{k+level(p)+1}_{P^{sdr}}(M) \models p(\vec{t})$

- Suppose that whenever $J \models p^{sdr}(\vec{t}, d)$ there exists $l \geq 0$ such that $J \models p^{depth}(\vec{t}, \mathsf{succ}^l(d))$. Then if $T^k_{P^{sdr}}(M) \models p^{sdr}(\vec{t}, \mathsf{succ}^l(0)), p(\vec{t})$ for some $l \geq 0$ it follows that $T^k_P(\emptyset) \models p(\vec{t})$.

Proof 5.2 Consider the first case, as the second is straightforward. Proof by induction. Suppose $T^k_P(\emptyset) \models p(\vec{t})$.

- To show that $T^{k+level(p)}_{P^{sdr}}(M) \models p^{sdr}(\vec{t}, \mathsf{succ}^l(0))$ for some $l \geq 0$. Since $T^k_P(\emptyset) = \emptyset$ the result vacuously holds for $k = 0$. Thus consider $k \geq 1$. Suppose $T^k_P(\emptyset) \models p(\vec{t})$. Then a grounding substitution θ exists for a clause $c = H :- B_1, \ldots, B_n \in P$ such that $\theta(H) = p(\vec{t})$ and $\theta(B_1), \ldots, \theta(B_n) \in T^{k-1}_P(\emptyset)$. Suppose $H = p(\vec{s})$ and $B_i = p_i(\vec{s}_i)$.
  - Suppose $sdr(c) = p^{sdr}(\vec{s}, d) {:-} d = \mathsf{succ}(d'), B'_1, \ldots, B'_n$. Let $I = \{i \in [1, n] \,|\, p_i \simeq p\}$.
    * Consider $i \in I$. By induction it follows that $T^{(k-1)+level(p)}_{P^{sdr}}(M) \models p^{sdr}_i(\theta(\vec{s}_i), d_i)$ for some $d_i = \mathsf{succ}^{l_i}(0)$ since $level(p_i) = level(p)$. Since $I \neq \emptyset$, put $l = \max(\{l_i \mid i \in J\})$ and define $\sigma = \theta \cup \{d = \mathsf{succ}(\mathsf{succ}^l(0)), d' \mapsto \mathsf{succ}^l(0)\}$. By lemma 5.1 it follows that $T^{(k-1)+level(p)}_{P^{sdr}}(M) \models \sigma(p^{sdr}_i(\vec{s}_i, d'))$.
    * Consider $i \in [1, n] \setminus I$. By induction $T^{(k-1)+level(p_i)+1}_{P^{sdr}}(M) \models p_i(\theta(\vec{s}_i))$. But $level(p_i) < level(p)$ hence $T^{(k-1)+level(p)}_{P^{sdr}}(M) \models p_i(\sigma(\vec{s}_i))$.

    Because θ is a grounding substitution for c, it follows that σ is a grounding substitution for $sdr(c)$. Hence $T^{k+level(p)}_{P^{sdr}}(M) \models \sigma(p^{sdr}(\vec{s}, d)) = p^{sdr}(\vec{t}, \mathsf{succ}^{l+1}(0))$ as required.
  - Suppose $sdr(c) = p^{sdr}(\vec{s}, d) {:-} \mathsf{true}, B'_1, \ldots, B'_n$. The argument is a simplified version of the previous case.

- To show $T^{k+level(p)+1}_{P^{sdr}}(M) \models p(\vec{t})$. Recall $c = p(\vec{x}) {:-} p^{depth}(\vec{x}, d), p^{sdr}(\vec{x}, d) \in P^{sdr}$. By the preceding argument there exists $l \geq 0$ with $T^{k+level(p)}_{P^{sdr}}(M) \models p^{sdr}(\vec{t}, \mathsf{succ}^l(0))$. Also there exists $k \geq 0$ such that $M \models p^{sdr}(\vec{t}, \mathsf{succ}^{k+l}(0)))$. By lemma 5.1, $T^{k+level(p)}_{P^{sdr}}(M) \models p^{sdr}(\vec{t}, \mathsf{succ}^{k+l}(0)))$. A grounding substitution θ for c exists such that $\theta(\vec{x}) = \vec{t}$ and $\theta(d) = \mathsf{succ}^{k+l}(0)$. Hence $T^{k+level(p)+1}_{P^{sdr}}(M) \models p(\vec{t})$ as required.

Proof 5.3 Let I, J and M be minimal models for P, $P^{sdr} \cup P^{depth}$ and P^{depth}.

- Suppose $J \models p(\vec{t})$. There exists $k \geq 0$ such that $T^k_{P^{sdr} \bigcup P^{delay}}(\emptyset) \models p(\vec{t})$, thus $T^k_{P^{sdr}}(M) \models p(\vec{t})$. Since $T^k_{P^{sdr}}(M) \models p(\vec{t})$ it follows $T^k_{P^{sdr}}(M) \models p^{sdr}(\vec{t}, \mathsf{succ}^l(0))$ for some $l \geq 0$. Whence $T^k_P(\emptyset) \models p(\vec{t})$ by lemma 5.2. But $I \supseteq T^k_{P^{sdr} \bigcup P^{delay}}(M)$ hence the result follows.

- Suppose $I \models p(\vec{t})$. There exists $k \geq 0$ such that $T_P(k) \models p(\vec{t})$, whence $T^{k+level(p)+1}_{P^{sdr}}(M) \models p(\vec{t})$ by lemma 5.2. But $J \supseteq T^{k+level(p)+1}_{P^{sdr} \bigcup P^{depth}}(\emptyset)$ and $T^{k+level(p)+1}_{P^{sdr} \bigcup P^{depth}}(\emptyset) = T^{k+level(p)+1}_{P^{sdr}}(M)$ since the predicates in P^{delay} are not defined in terms of those in P^{sdr}. The result then follows.

6 The transformation: automation and efficiency

Thus far, the development of the transform has been theoretical: the issues of termination, soundness and completeness have been addressed. This section is more discursive since it considers the prospects for automating the transform as well examining the efficiency of this approach to control generation.

6.1 Automation

The construction of P^{sdr} is straightforward to automate. Automatically synthesising definitions for P^{delay} is more challenging, but nevertheless this problem is also susceptible to automation. Recall that the rôle of the predicate $p^{depth}(x_1, \ldots, x_k, d)$ is to calculate an upper bound d in terms of the arguments $x_1, \ldots, x_k$. Recall too that the goal $p^{sdr}(x_1, \ldots, x_k, d)$, and those it recursively calls, constrain d in terms of $x_1, \ldots, x_k$ and *vice versa*. This is illustrated below.

Example 6.1 Consider again the quicksort program of figure 1, and in particular the predicate append^{sdr} introduced in example 5.1.

append^{sdr}([], X, X, D) :- true.
append^{sdr}([X|Xs], Ys, [X|Zs], D) :- D = succ(D1), append^{sdr}(Xs, Ys, Zs, D1).

Observe that the first and fourth arguments of the goal append^{sdr} constrain one another. Specifically, if I is the minimal model of this program and $I \models \mathsf{append}^{sdr}(t_1, t_2, t_3, d)$ then $|d|_{\textit{term-size}} \geq |t_1|_{\textit{list-length}}$.

This suggests a tactic for computing depth bounds: perform argument size analysis on P^{sdr}. Argument size analysis infers size relationships between the arguments of each predicate in a given program. This analysis is usually applied to reason about termination; inter-argument size relationships inferred for the body atoms of a clause can be combined to observe a decrease between the size of arguments in the head and the size of the arguments in the body [13]. Because of its wide-applicability in termination analysis, argument size analysis is a mature topic in abstract interpretation [3, 11, 14, 15], the established approach being to perform analysis over a constraint program (an abstract program) that is derived from the input program (the concrete program). An analysis of the abstract program then reveals argument size information for the concrete program [18]. Each syntactic equation in the concrete program, for example $\mathsf{T1} = [\mathsf{X}|\mathsf{Xs1}]$, is replaced with a linear equation in the abstract program, for instance $\mathsf{T1} = 1+\mathsf{Xs1}$. This constraint captures the property that T1 is bound to a list of fixed length iff Xs1 is bound to a list of fixed length, where the length of T1 exceeds Xs1 by exactly one.

Example 6.2 Continuing with example 6.1, the abstract process is simplified by putting the concrete program into a normal form that makes unification explicit. This leads to the following version of the concrete program where T1 and T2 are fresh variables:

append^{sdr}(T1, X, T2, D) :- T1 = [], X = T2, true.
append^{sdr}(T1, Ys, T2, D) :- T1 = [X|Xs], T2 = [X|Zs], D = succ(D1),
    append^{sdr}(Xs, Ys, Zs, D1).

The abstraction of syntactic constraint is actually performed wrt to a norm such as $|.|_{\textit{list-length}}$ or $|.|_{\textit{term-size}}$ (or even a combination of norms [17]). For the purposes of exposition, equations that involve the depth parameters are abstracted wrt $|.|_{\textit{term-size}}$ whereas all other equations are abstracted wrt $|.|_{\textit{list-length}}$. Thus the equation $\mathsf{D} = \mathsf{succ}(\mathsf{D1})$ is rendered as $\mathsf{D} = 1 + \mathsf{D1}$. The non-negativity constraints on the variables in each clause express the fact that the size of terms (as qualified by the norms $|.|_{\textit{term-size}}$ and $|.|_{\textit{list-length}}$) are always non-negative.

append^{sdr}(T1, X, T2, D) :-
    T1 = 0, X = T2,
    T1 $\geq$ 0, X $\geq$ 0, T2 $\geq$ 0, D $\geq$ 0.
append^{sdr}(T1, Ys, T2, D) :-

$\mathsf{T1} = 1 + \mathsf{Xs}$, $\mathsf{T2} = 1 + \mathsf{Zs}$, $\mathsf{D} = 1 + \mathsf{D1}$,
$\mathsf{T1} \geq 0$, $\mathsf{Ys} \geq 0$, $\mathsf{T2} \geq 0$, $\mathsf{D} \geq 0$, $\mathsf{X} \geq 0$, $\mathsf{Xs} \geq 0$, $\mathsf{Zs} \geq 0$, $\mathsf{D} \geq 0$, $\mathsf{D1} \geq 0$,
$\mathsf{append}^{sdr}(\mathsf{Xs}, \mathsf{Ys}, \mathsf{Zs}, \mathsf{D1})$.

The minimal model of this abstract program is

$$\{\mathsf{append}^{sdr}(t_1, t_2, t_3, d) \mid t_1 + t_2 = t_3 \wedge t_1 \leq d \wedge 0 \leq t_1 \wedge 0 \leq t_2\}$$

This minimal model reveals that if I is the minimal model for the concrete program and $I \models \mathsf{append}^{sdr}(t_1, t_2, t_3, d)$ then $|d|_{term\text{-}size} \geq |t_1|_{list\text{-}length}$. It also shows that $|t_3|_{list\text{-}length} \geq |t_1|_{list\text{-}length}$, hence if $|d|_{term\text{-}size} \geq |t_3|_{list\text{-}length}$ then $|d|_{term\text{-}size} \geq |t_1|_{list\text{-}length}$. Therefore the predicate $\mathsf{append}^{depth}(\mathsf{X}, \mathsf{Y}, \mathsf{Z}, \mathsf{D})$ should delay until X or Z are rigid, and then proceed to instantiate D to the length of the rigid argument.

The minimal model of the abstract program can be computed (albeit in an approximate way [11]) by iteratively applying an abstract version of the T_P operator [3, 11, 18]. It is worth noting that the modular nature of the transform – that one depth counter is associated with each SCC – considerably simplifies the application of argument size analysis. In fact, for purposes of efficiency, (goal-independent) argument size analysis infers argument size relations for each SCC in turn, in a bottom-up fashion, following the call graph hierarchy [3]. Calculating an upper bound on the depth of the whole computation, as required in the approach of [38], would be more difficult to automate.

Although argument size analysis traditionally expressed argument relationship with systems of linear equations [14] or linear inequations [3, 11] there is no reason why the technique cannot be refined. In fact, the use of difference equation techniques have already been explored in cost analysis [15] and there in no reason why the Groebner base and cylindrication algorithms that are routinely applied in computer algebra cannot be applied within constraint programming [21]. Such a constraint system would enable polynomial argument size relationship to be inferred. Thus, in principle, argument size analysis can be refined to deduce more expressive (non-linear) size constraints on the depth parameter.

6.2 Efficiency

The essential idea behind the transform is to ensure termination by delaying possibly non-terminating goals until certain arguments of those goals become rigid. This section discusses the efficiency of this approach to control generation. Relative to the original program, the transformation induces additional overheads in terms of synchronisation, depth checking and rigidity detection. We consider each of these overheads in turn. Firstly, suspending and resuming a goal controlled via a block declaration is a constant time operation in SICStus; indeed this operation is directly supported at the level of the abstract machine [8]. Secondly, the test against one and decrement operations both translate to single abstract machine instructions in SICStus 3 [8]. Thirdly, although the transformation introduces rigidity checks that are realised with block declarations, it should be noted that block declarations, or more expensive devices such as a synchronising meta-call [40], are likely to be used in any *ad hoc* attempt at specifying the control.

If efficiency is really a critical issue, then it should be noted that the use of depth bounds – which are merely required to be upper bounds – offers scope for performance tuning. Returning to the mergesort example, recall from section 3 that a depth of l where l is the length of the first and second arguments of msort is a convenient upper bound on the depth since it avoids the calculation of a transcendental function. However, recall also that split is depth bounded by the length of its first argument and merge is depth bounded by the sum of the length of its first and second arguments. It therefore follows that l also bounds the depth of split and merge as well as that of msort. This observation enables rigidity checks within split and merge to be eliminated and leads to the program given in figure 6. Furthermore, the block declarations for split_sdr and merge_sdr will always be satisfied and therefore can be removed (for clarity they have been commented out). The optimised version of the program only performs rigidity checking on the initial input; following the call to msort_aux the program runs without synchronisation checks and the only overhead is the decrementation and depth testing. The net result is that, with SICStus 3.8.5 on a 1GHz, 256MByte PC, the optimised program averages 75 msecs to sort 2048 random numbers. The original program given in figure 2 averages 128 msecs on the same data set, whereas the unoptimised transformed version of figure 4 takes 166 msecs.

Other sample programs are available from http://www.cs.kent.ac.uk/people/staff/amk/. The code qsort_naive represents an attempt at making the classic quicksort algorithm reversible. This program averages 618 msecs to sort 1024 random numbers but, alas, only produces one solution in reverse mode before looping. The code qsort_control generated by a naïve application of the transform averages 838 msecs on random data, but is reversible. In reverse mode, it enumerates combinations in a similar fashion to mergesort, when its second argument is instantiated to a list of ordered numbers.

The program nqueens_control was obtained by a straightforward applica-

```
msort(L, S) :- msort_depth(L, S, D), msort_sdr(L, S, D).

msort_depth(L, S, D) :- list_length(L, D1, D), list_length(S, D2, D),
    msort_aux(D1, D2, D).

:- block msort_aux(-, -, ?).
msort_aux(D1, D2, D) :- D1 = D2, D = D1.

:- block msort_sdr(?, ?, -).
msort_sdr([], [], _).
msort_sdr([X], [X], _).
msort_sdr([X, Y | Xs], S, D) :- 1 ≤ D, D1 is D - 1,
    split_sdr([X, Y | Xs], L1, L2, D),
    msort_sdr(L1, S1, D1), msort_sdr(L2, S2, D1), merge_sdr(S1, S2, S, D).

%:- block split_sdr(?, ?, ?, -).
split_sdr([], [], [], _).
split_sdr([X | L], [X | L1], L2, D) :- 1 ≤ D, D1 is D - 1,
    split_sdr(L, L2, L1, D1).

%:- block merge_sdr(?, ?, ?, -).
merge_sdr([], Ys, Ys, _).
merge_sdr(Xs, [], Xs, _).
merge_sdr([X|Xs], [Y|Ys], [X|Zs], D) :- 1 ≤ D, D1 is D - 1,
    leq(X, Y), merge_sdr(Xs, [Y|Ys], Zs, D1).
merge_sdr([X|Xs], [Y|Ys], [Y|Zs], D) :- 1 ≤ D, D1 is D - 1,
    gt(X, Y), merge_sdr([X|Xs], Ys, Zs, D1).
```

Figure 6: Optimised version of the control generation transformation for the mergesort program

tion of transform to the program nqueens_naive. The nqueens_naive program encodes the n-queens problem but is actually buggy because its perm predicate – which is intended to encode a permutation generator – does not universally terminate for some queries.

```
perm([], []).
perm([X | Xs], P) :- select(P, X, Rs), perm(Xs, Rs).
```

```
select([N | Ns], N, Ns).
select([N | Ns], S, [N | Rs]) :- select(Ns, S, Rs).
```

Specifically, perm([1,2,3], P) only produces one solution P = [1,2,3] before looping. However, the generated program nqueens_control terminates and takes 92 secs to count the number of solutions for the $n = 10$ board instance. The code is guaranteed to terminate no matter what the ordering of the original body goals, and in particular the perm(Range, Soln) and safe(Soln) goals in the following clause:

```
nqueens(N, Soln):-
    length(Range, N), length(Soln, N), range(Range, 1),
    perm(Range, Soln), safe(Soln).
```

can be reordered without compromising termination. The goal safe(Soln) checks that the board configuration represented by the list Soln is safe whereas range(Range, 1) instantiates elements of the list Range to consecutive numbers starting at 1. Since SICStus implements left-to-right scheduling by default, with the proviso that suspended goals are invoked as soon as possible, this reordering introduces coroutining between the goals safe(Soln) and perm(Range, Soln) in the transformed program. In effect, bindings to the elements of the list Soln are incrementally generated by perm(Range, Soln). As soon as Soln contains two elements, representing two queens that can take one another, then Soln is an unsafe board configurations; it is unsafe no matter how the elements of Soln are instantiated. By ordering safe(Soln) before perm(Range, Soln), the goal safe(Soln) will check the safety of a new queen, relative to the existing queens, as soon as the queen is added to the board. When the new queen is verified to be safe with respect to the existing queens, control switches back from safe(Soln) to perm(Range, Soln) to add the next queen. This form of coroutining is an important tactic for improving search since unsafe configurations can be detected without necessarily adding all the queens to the board. In fact, coroutining reduces the total running time from 83 secs to 681 msecs for $n = 10$. For the purposes of a fair comparison, block declarations were carefully added to nqueens_naive to mimic this coroutining behaviour. The termination problem was finessed by observing that perm(P, [1,2,3]) *does* correctly enumerate permutations, hence the arguments in the call perm(Range, Soln) were reversed to enforce termination. The resulting program, queens_test_and_generate, requires 691 msecs for $n = 10$, which is close to the speed of the transformed code.

Experimental work can never be conclusive, but these results do suggest that the transformation is not prohibitively expensive. Moreover, the resulting code is flexible. Since the transform is underpinned by semi-delay recurrency (rather than delay recurrency [29]) goals can be reordered to introduce coroutining into the transformed code and thereby improve efficiency. The flexibility offered by the use of upper bounds allows the performance to be tuned by choosing judicious upper bounds that eliminate rigidity checks. However, even without these refinements, the transform produces code that is surprisingly efficient considering its generality.

7 Conclusions

The aim of control generation is to automatically derive a computation rule for a program that is efficient but does not compromise program correctness. The problem has been effectively tackled by transforming a program into a semantically equivalent one, introducing safe delay declarations and defining a flexible computation rule, which ensures that all goals for the transformed program terminate. Furthermore, it has been 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, its practicality has been demonstrated. In particular, it has been shown how transformed programs can be straightforwardly implemented in a standard logic programming system, and how such a program can be optimised to reduce the number of rigidity checks. Furthermore, with the proposed transformation, the termination problems caused by speculative output bindings are eliminated without the use of a local computation rule or other costly overhead. Moreover, the coroutining behaviour which is then possible contributes significantly to the efficiency of the generated code.

Acknowledgements We would like to thank Elena Marchiori for clarifying details of her work. Thanks are also due to Samir Genaim for providing the inorder example.

References

[1] K. R. Apt and D. Pedreschi. Reasoning about Termination of Pure Prolog Programs. *Information and Computation*, 106(1):109–157, 1993.

[2] K. R. Apt and D. Pedreschi. Modular Termination Proofs for Logic and Pure Prolog programs. In G. Levi, editor, *Advances in Logic Pro-*

gramming Theory, pages 183–229. Oxford University Press, 1994. Also available as technical report CS-R9316 from Centrum voor Wiskunde en Informatica, CWI, Amesterdam.

[3] F. Benoy and A. King. Inferring Argument Size Relationships with CLP($\mathcal{R}$). In J. P. Gallagher, editor, *Logic Programming Synthesis and Transformation*, volume 1207 of *Lecture Notes in Computer Science*, pages 204–223. Springer-Verlag, 1996.

[4] M. Bezem. Strong Termination of Logic Programs. *The Journal of Logic Programming*, 15(1&2):79–97, 1993.

[5] A. Bossi, N. Cocco, S. Etalle, and S. Rossi. On Modular Termination Proofs of General Logic Programs. *Theory and Practice of Logic Programming*, 2(3):263–291, 2002.

[6] A. Bossi, S. Etalle, S. Rossi, and J.-G. Smaus. Semantics and Termination of Simply-Moded Logic Programs with Dynamic Scheduling. *ACM Transactions on Computational Logic*, 2004. To appear.

[7] M. Carlsson. Freeze, Indexing and Other Implementation Issues in the WAM. In J.-L. Lassez, editor, *International Conference on Logic Programming*, pages 40–58. MIT Press, 1987.

[8] M. Carlsson. Personal communications between Andy King and Mats Carlsson on the sicstus abstract machine, 2002–2004.

[9] L. Cavedon. Continuity, consistency, and completeness properties of logic programs. In G. Levi and M. Martelli, editors, *International Conference on Logic Programming*, pages 571–584. MIT Press, 1989.

[10] K. L. Clark, F. G. McCabe, and S. Gregory. IC-Prolog Language Features. In K. L. Clark and S.-Å. Tärnlund, editors, *Logic Programming*, pages 253–266. Academic Press, 1982.

[11] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *JLP*, 1992.

[12] V. Dahl. Two Solutions for the Negation Problem. In S.-Å. Tärnlund, editor, *Workshop on Logic Programming*, 1980.

[13] D. De Schreye and S. Decorte. Termination of Logic Programs: The Never-Ending Story. *The Journal of Logic Programming*, 19&20:199–260, 1994.

[14] D. De Schreye and K. Verschaetse. Deriving Linear Size Relations for Logic Programs by Abstract Interpretation. *New Generation Computing*, 13(2):117–154, 1995.

[15] S. K. Debray and N.-W. Lin. Cost Analysis of Logic Programs. *ACM Transactions on Programming Languages and Systems*, 15(5):826–875, 1993.

[16] P. Deransart, A. Ed-Dbali, and L. Cervoni. *Prolog: The Standard Reference Manual*. Springer-Verlag, 1996.

[17] S. Genaim, M. Codish, J. P. Gallagher, and V. Lagoon. Combining Norms to Prove Termination. In A. Cortesi, editor, *Verification, Model Checking and Abstract Interpretation*, number 2294 in Lecture Notes in Computer Science, pages 126–138. Springer-Verlag, 2002.

[18] R. Giacobazzi, S. K. Debray, and G. Levi. Generalized Semantics and Abstract Interpretation for Constraint Logic Programs. *The Journal of Logic Programming*, 3(25):191–248, 1995.

[19] P. M. Hill and J. W. Lloyd. *The Gödel Programming Language*. MIT Press, 1994.

[20] S. Hoarau and F. Mesnard. Inferring and Compiling Termination for Constraint Logic Programs. In P. Flener, editor, *Logic-based Program Synthesis and Transformation*, volume 1559 of *Lecture Notes in Computer Science*, pages 240–254. Springer-Verlag, 1998.

[21] H. Hong. Non-linear real constraints in constraint logic programming. In H. Kirchner and G. Levi, editors, *Algebraic and Logic Programming*, volume 632 of *Lecture Notes in Computer Science*, pages 201–212. Springer-Verlag, 1992.

[22] J. Jaffar, S. Michaylov, P. J. Stuckey, and R. H. C. Yap. The CLP($\mathcal{R}$) Language and System. *ACM Transactions on Programming Languages and Systems*, 14(3):339–395, 1992.

[23] A. King. Pair-sharing over Rational Trees. *The Journal of Logic Programming*, 46(1-2):139–155, 2000.

[24] R. A. Kowalski. Algorithm = Logic + Control. *Communications of the ACM*, 22(7):424–436, 1979.

[25] J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987.

[26] S. Lüttringhaus-Kappel. Control Generation for Logic Programs. In D. S. Warren, editor, *International Conference on Logic Programming*, pages 478–495. The MIT Press, 1993.

[27] E. Marchiori. Personal communication between Jonathan Martin and Elena Marchiori, 1996.

[28] E. Marchiori and F. Teusink. Termination of Logic Programs with Delay Declarations. In J. W. Lloyd, editor, *International Logic Programming Symposium*, pages 447–461. MIT Press, 1995.

[29] E. Marchiori and F. Teusink. Termination of Logic Programs with Delay Declarations. *The Journal of Logic Programming*, 39(1–3):95–124, 1999.

[30] J. C. Martin and A. King. Generating Efficient, Terminating Logic Programs. In M. Bidoit and M. Dauchet, editors, *Theory and Practice of Software Development*, volume 1214 of *Lecture Notes in Computer Science*, pages 273–284. Springer-Verlag, 1997. A longer 15 page version is available from http://www.cs.kent.ac.uk/pubs/1997/216/.

[31] J. C. Martin and A. King. On the Inference of Natural Level Mappings. In M. Bruynooghe and K.-K. Lau, editors, *Program Development in Computational Logic*, Lecture Notes in Computer Science. Springer-Verlag, 2004.

[32] J.-G. Smaus, P. M. Hill, and A. King. Verifying Termination and Error-Freedom of Logic Programs with block Declarations. *Theory and Practice of Logic Programming*, 1(4):447–486, 2001.

[33] F. Mesnard. Towards Automatic Control for CLP($\mathcal{X}$) Programs. In M. Proietti, editor, *Logic Programming Synthesis and Transformation*, volume 1048 of *Lecture Notes in Computer Science*, pages 106–119. Springer-Verlag, 1995.

[34] F. Mesnard. Inferring Left-terminating Classes of Queries for Constraint Logic Programs by means of Approximations. In *Joint International Conference and Symposium on Logic Programming*, pages 7–21. MIT Press, 1996.

[35] F. Mesnard and S. Ruggieri. On Proving Left Termination of Constraint Logic Programs. *ACM Transactions on Computational Logic*, 4(2):207–259, 2003.

[36] L. Naish. *Negation and Control in Logic Programs.* Springer-Verlag, 1986.

[37] L. Naish. Coroutining and the Construction of Terminating Logic Programs. *Australian Computer Science Communications*, 15(1):181–190, 1993.

[38] D. Pedreschi and S. Ruggieri. Bounded Nondeterminism of Logic Programs. In D. De Schreye, editor, *International Conference on Logic Programming*, pages 350–364. MIT Press, 1999.

[39] D. Pedreschi, S. Ruggieri, and J.-G. Smaus. Classes of Terminating Logic Programs. *Theory and Practice of Logic Programming*, 2(3):369–418, 2002.

[40] SICS. *SICStus Prolog User's Manual*, 2004. See http://www.sics.se/sicstus/.

[41] A. Takeuchi and K. Furukawa. Bounded Buffer Communication in Concurrent Prolog. *New Generation Computing*, 3(2):145–155, 1985.

[42] M. H. van Emden and G. J. de Lucena Filho. Predicate Logic as a Language for Parallel Programming. In K. L. Clark and S.-Å. Tärnlund, editors, *Logic Programming*, pages 189–198. Academic Press, 1982.

[43] J. Van Leeuwen, editor. *Handbook of Theoretical Computer Science: Volume B.* Elsevier, 1990.

[44] T. Vasak and J. Potter. Characterisation of Terminating Logic Programs. In *Symposium on Logic Programming*, pages 140–147. IEEE Press, 1986.

[45] K. Verschaetse, D. De Schreye, and M. Bruynooghe. Generation and Compilation of Efficient Computation Rules. In D. H. D. Warren and P. Szeredi, editors, *International Conference on Logic Programming*, pages 700–714. MIT Press, 1990.

[46] L. Vieille. Recursive Query Processing: The Power of Logic. *Theoretical Computer Science*, 69(1):1–53, 1989.