Sonic Partial Deduction

Jonathan Martin and Michael Leuschel
mal@ecs.soton.ac.uk

Declarative Systems and Software Engineering Group
Technical Report DSSE-TR-99-3
February 12 1999

www.dsse.ecs.soton.ac.uk/techreports/

Department of Electronics and Computer Science
University of Southampton
Highfield, Southampton SO17 1BJ, United Kingdom

Sonic Partial Deduction

Jonathan Martin and Michael Leuschel

Department of Electronics and Computer Science
University of Southampton, Southampton SO17 1BJ, UK
e-mail: {jcm93r,mal}@ecs.soton.ac.uk
Fax: +44 1703 59 3045 Tel: +44 1703 59 3377

Abstract. The current state of the art for ensuring finite unfolding of logic programs consists of a number of online techniques where unfolding decisions are made at specialisation time. Introduction of a static termination analysis phase into a partial deduction algorithm permits unfolding decisions to be made offline, before the actual specialisation phase itself. This separation improves specialisation time and facilitates the automatic construction of compilers and compiler generators. The main contribution of this paper is how this separation may be achieved in the context of logic programming, while providing non-trivial support for partially static datastructures.
The paper establishes a solid link between the fields of static termination analysis and partial deduction enabling existing termination analyses to be used to ensure finiteness of the unfolding process. This is the first offline technique which allows arbitrarily partially instantiated goals to be sufficiently unfolded to achieve good specialisation results. Furthermore, it is demonstrated that an offline technique such as this one can be implemented very efficiently and, surprisingly, yield even better specialisation than a (pure) online technique. It is also, to our knowledge, the first offline approach which passes the KMP test (i.e., obtaining an efficient Knuth-Morris-Pratt pattern matcher by specialising a naive one).

Keywords: Partial evaluation, mixed computation, and abstract interpretation, Program transformation and specialisation, Logic programming, Partial deduction, Termination.

1 Introduction

Control of partial deduction—a technique for the partial evaluation of pure logic programs—is divided into two levels. The local level guides the construction of individual SLDNF-trees while the global level manages the forest, determining which, and how many trees should be constructed. Each tree gives rise to a specialised predicate definition in the final program so the global control ensures a finite number of definitions are generated and also controls the amount of polyvariance, i.e. the number of specialised versions produced for each individual source predicate. The local control on the other hand determines what each specialised definition will look like.

Recent work on global control of partial deduction has reached a level of maturity where fully automatic algorithms can be described which offer a near optimal control of polyvariance and guarantee termination of the overall partial deduction process [32]. Such algorithms are parameterised by the local control component: an unfolding rule which describes how an incomplete SLDNF-tree should be constructed for a given goal and program. It is a requirement of any terminating partial deduction system that such trees are necessarily finite. Techniques developed to ensure finite unfolding of logic programs [9, 36, 35] have been inspired by the various methods used to prove termination of rewrite systems [17, 16]. Whilst, by no means *ad hoc*, there is little direct relation between these techniques and those used for proving termination of logic programs (or even those of rewrite systems). This means that advances in the static termination analysis technology do not directly contribute to improving the control of partial deduction and the quality of specialised code produced by partial deduction systems. The work of this chapter aims to bridge this gap.

Moreover, the control described in [9, 36, 35] as well as the more recent [43, 29] are inherently online, meaning that they are much slower than offline approaches and that they are not based on a *global analysis* of the program's behaviour which enables control decisions to be taken before the actual specialisation phase itself.

Offline approaches to local control of partial deduction on the other hand [38, 22, 23, 10] have been very limited in other respects. Specifically, each atom in the body of a clause is marked as either *reducible* or *non-reducible*. Reducible atoms remain are *always* unfolded while non-reducible atoms on the other hand are *never* unfolded. Whilst this approach permits goals to be unfolded at normal execution speed, it can unduly restrict the amount of unfolding which takes place with a detrimental effect on the resulting specialised program. Another problem of [38, 23] is that it classifies *arguments* either as static (known at specialisation time) or dynamic (unknown at specialisation time). This division is too coarse, however, to allow refined unfolding of goals containing partially instantiated data where some parts of the structure are known and others unknown. Such goals are very common in logic programming, and the key issue which needs to be considered is termination. A partial solution to this problem has been presented in [10], but it still sticks with the limited unfolding mentioned above and can "only" handle a certain class of partially instantiated data (data bounded wrt some semi-linear norm).

A Sonic Approach This paper proposes a flexible solution to the local termination problem for offline partial deduction of logic programs, encompassing the best of

both worlds. Based on the cogen approach[1] for logic programs [23], the construction of a generating extension will be described which "compiles in" the local unfolding rule for a program and is capable of constructing maximally expanded SLDNF-trees of finite depth.

The technique builds directly on the work of [37] which describes a method for ensuring termination of logic programs with delay. The link here is that the residual goals of a deadlocked computation are the leaves of an incomplete SLD-tree. The basic idea is to use static analysis to derive relationships between the sizes of goals and the depths of derivations. This depth information is incorporated in a generating extension and is used to accurately control the unfolding process. At specialisation time the sizes of certain goals are computed and the maximum depth of subsequent derivations is fixed according to the relationships derived by the analysis. In this way, termination is ensured whilst allowing a flexible and generous amount of unfolding. Section 3 reviews the work of [37] and shows how it can be used directly to provide the basis of a generating extension which allows finite unfolding of bounded goals. A simple extension to the technique is described in Section 4 which also permits the safe unfolding of unbounded goals.

This is the *first* offline approach to partial deduction which is able to successfully unfold arbitrarily **partially instantiated** (i.e. unbounded) goalssuch as the one encountered in the example in Section 4. In fact, it is demonstrated that the method can, surprisingly, yield even better specialisation than (pure) online techniques. In particular, some problematic issues in unfolding, notably unfolding under a coroutining computation rule and the back propogation of instantiations [35], can be easily handled within the approach (Section 6). Furthermore, it is the *first* offline approach which passes the **KMP test** (i.e., obtaining an efficient Knuth-Morris-Pratt pattern matcher by specialising a naive one), as demonstrated by the extensive experiments in Section 7.

An analysis which measures the depths of derivations may be termed a *sounding analysis*. Section 5 describes how such an analysis can be based on existing static termination analyses which compute level mappings and describes how the necessary depths may be obtained from these level mappings. Unfolding based on a sounding analysis then, is the basis of *sonic partial deduction*.

2 Preliminaries

Familiarity with the basic concepts of logic programming and partial deduction is assumed [33, 34]. A *level mapping* (resp. *norm*) is a mapping from ground atoms (resp. ground terms) to natural numbers. For an atom A and level mapping $|.|$, $A_{|.|}$ denotes the set $\{|A\theta| \mid A\theta \text{ is ground}\}$. An atom A is *(un)bounded* wrt $|.|$ if $A_{|.|}$ is (in)finite [13]. For this paper, the notion of level mapping is extended to non-ground atoms by defining for any atom A, $|A| = min(A_{|.|})$; and similarly for norms. The norm $|t|_{len}$ returns the length of the list t. A list t is rigid iff $|t|_{len} = |t\theta|_{len}$ for all θ. A clause $c : H \leftarrow A_1, \ldots, A_n$ is *recurrent* if for every grounding substitution θ for c, $|H\theta| > |A_i\theta|$ for all $i \in [1, n]$.

[1] Instead of trying to achieve a compiler generator (cogen) by self-application [18] one writes the cogen directly [41].

3 Unfolding Bounded Atoms

A fundamental problem in adapting techniques from the termination literature for use in controlling partial deduction is that the various analyses that have been proposed (see [13] for a survey) are designed to prove *full* termination for a given goal and program, in other words guaranteeing finiteness of the complete SLDNF-tree constructed for the goal. For example, consider the goal ← Flatten([x, y, z], w) and the program Flatten consisting of the clauses app_1, app_2, $flat_1$ and $flat_2$.

$flat_1$ Flatten([], []).
$flat_2$ Flatten([e|x], r) ← Append(e, y, r) ∧ Flatten(x, y).
app_1 Append([], x, x).
app_2 Append([u|x], y, [u|z]) ← Append(x, y, z).

A typical static termination analysis would (correctly) fail to deduce termination for this program and goal. Most analyses can infer that a goal of the form ← Flatten(x, y) will terminate if x is a rigid list of rigid lists, or if x is a rigid list and y is a rigid list. In the context of partial deduction however, such a condition for termination will usually be too strong. The problem is that the information relating to the goal, by the very nature of partial deduction, is often incomplete. For example, the goal ← Flatten([x, y, z], w), will not terminate but the program can be partially evaluated to produce the following specialised definition of Flatten/2.

Flatten([x, y, z], r) ← Append(x, r1, r) ∧ Append(y, r2, r1) ∧ Append(z, [], r2).

The scheme described in [37] transforms programs into *efficient* and *terminating* programs. It will for instance transform the non-terminating program Flatten into the following efficient, terminating program, by adding an extra depth parameter.

$flat^*$ Flatten(x, y) ← SetDepth_F(x, d) ∧ Flatten(x, y, d).

DELAY Flatten(_, _, d) UNTIL Ground(d).
$flat_1^*$ Flatten([], [], d) ← d ≥ 0.
$flat_2^*$ Flatten([e|x], r, d) ← d ≥ 0 ∧ Append(e, y, r) ∧ Flatten(x, y, d − 1).

app^* Append(x, y, z) ← SetDepth_A(x, z, d) ∧ Append(x, y, z, d).

DELAY Append(_, _, _, d) UNTIL Ground(d).
app_1^* Append([], x, x, d) ← d ≥ 0.
app_2^* Append([u|x], y, [u|z], d) ← d ≥ 0 ∧ Append(x, y, z, d − 1).

For now, assume that the (meta-level) predicate SetDepth_F(x, d) is defined such that it always succeeds instantiating the variable d to the length of the list x if this is found to be rigid,(i.e., $|x|_{len} = |x\theta|_{len}$ for every substitution θ), and leaving d unbound otherwise. Note that a call to Flatten/3 will proceed only if its third argument has been instantiated as a result of the call to SetDepth_F(x, d). The purpose of this last argument is to ensure finiteness of the subsequent computation. More precisely, d is an upper bound on the number of calls to the recursive clause $flat_2^*$ *in any successful derivation.* Thus by failing any derivation where the number of such calls has exceeded this bound (using the test d ≥ 0), termination is guaranteed without losing completeness. The predicate SetDepth_A/3 is defined in a similar way, but instantiates d to the minimum of the lengths of the lists x and z, delaying if both x and z are unbounded.

The main result of [37] guarantees that the above program will terminate for every goal (in some cases the program will deadlock). Moreover, given a goal of the form ← Flatten(x, y) where x is a rigid list of rigid lists or where x is a rigid list and y is a rigid list, the program does not deadlock and produces all solutions to such a goal. In other words, both termination and completeness of the program are guaranteed.

Since the program is terminating for all goals, it can be viewed as a means of constructing a finite (possibly incomplete) SLD-tree for any goal. As mentioned above, it is indeed capable of complete evaluation but a partial evaluation for bounded goals may also be obtained. Quite simply, the deadlocking goals of the computation are seen to be the leaf nodes of an incomplete SLD-tree.

For example, the goal ← Flatten([x, y, z], r) leads to deadlock with the residual goal ← Append(x, r1, r, d1) ∧ Append(y, r2, r1, d2) ∧ Append(z, [], r2, d3). Removing the depth bounds, this residue can be used to construct a partial evaluation of the original goal resulting in the specialised definition of Flatten/2 above.

The approach, thus far, is limited in that it can only handle bounded goals. For unbounded goals the unfolding will deadlock immediately and it is not possible, for example, to specialise ← Flatten([[], [a] | z], r) in a non-trivial way. This strong limitation will be overcome in the following sections.

3.1 Relation to Previous Approaches

The method proposed in [8] (and further developed in [35]) ensures the construction of a finite SLD-tree through the use of a measure function which associates with each node (goal) in the tree a weight from a well-founded set (see also section 6.1). For example, the original measure function proposed by [8] maps individual atoms to natural numbers and then defines the weight of a goal to be the weight of its selected atom.

Finiteness is ensured by imposing the condition that the weight of any goal is strictly less than the weight of its *direct covering ancestor*. This last notion is introduced to prevent the comparison of unrelated goals which could precipitate the end of the unfolding process. Clearly, one should only compare the weights of goals whose selected atoms share the same predicate symbol. But this is not enough. Consider the atoms Append([1], y, r, 1) and Append([2], y1, y, 1) appearing in the LD-tree for Flatten([[1],[2]], r,2) (see Figure 1). Any sensible measure function would assign exactly the same weight to each atom. But, if these weights were compared, unfolding would be prematurely halted after four steps. Hence, this comparison must be avoided and this is justified by the fact that the atoms occur in separate "sub-derivations" of the main derivation. The direct covering ancestor of a goal G then is, loosely speaking, the "closest" ancestor G' occuring in the same sub-derivation where the selected atoms in G and G' share the same predicate symbol[2].

In the approach described here, the above notions are dealt with implicitly. Figure 1 depicts the SLD-tree for the goal ← Flatten([[1], [2]], r, 2) and the transformed version of Flatten. The depth argument of each atom may be seen as a weight as described above. Note that the weight of any atom in a sub-derivation (except the first) is implicitly derived from the weight of its direct covering ancestor by the process of resolution. This conceptual simplicity eliminates the need to explicitly trace direct covering ancestors, improving performance of the specialisation process and removing a potential source of programming errors.

[2] In fact, [8] states a slightly more general condition which is useful for unfolding meta-interpreters, but the details are not important here.

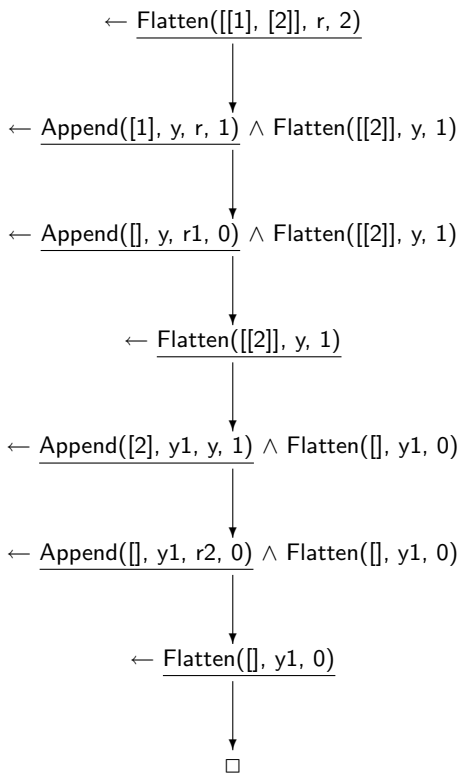

Fig. 1. Unfolding of ← Flatten([[1], [2]], r, 2)

4 Unfolding Unbounded Atoms

The main problem with the above transformation is that it only allows the unfolding of bounded goals. Often, as mentioned in the introduction, to achieve good specialisation it is necessary to unfold *unbounded* atoms also. This is *especially true in a logic programming setting*, where partially instantiated goals occur very naturally even at runtime. This capability may be incorporated into the above scheme as follows. Although an atom may be unbounded, it may well have a *minimum* size. For example the length of the list [1,2,3|x] must be at least three regardless of how x may be instantiated. In fact, this minimum size is an accurate measure of the size of the part of the term which is partially instantiated and this may be used to determine an estimate of the number of unfolding steps necessary for this part of the term to be consumed in the specialisation process. For example, consider the Append/3 predicate and the goal ← Append([1,2,3|x], y, z). Given that the minimum size of the first argument is three it may be estimated that at least three unfolding steps must be performed. Now suppose that the number of unfolding steps is fixed at one plus the minimum (this will usually give exactly the required amount of specialisation). The transformed Flatten program may now be used to control the unfolding by simply calling ← Append([1,2,3|x], y, z, 3). The problem here, of course, is that completeness is lost, since the goal fails if x does not become instantiated to []. To remedy this, an extra clause is introduced to capture the leaf nodes of the SLD-tree. The Append/3 predicate would therefore be transformed into the following.

app_1^* Append([], x, x, d) ← d ≥ 0.
app_2^* Append([u|x], y, [u|z], d) ← d ≥ 0 ∧ Append(x, y, z, d − 1).
app_3^* Append(x, y, z, d) ← d < 0 ∧ Append(x, y, z, _).

The call to Append/4 in the clause app_3^* immediately suspends since the depth argument is uninstantiated. The clause is only selected when the derivation length has exceeded the approximated length and the effect is that a leaf node (residual goal) is generated precisely at that point. For this reason, such a clause is termed a *leaf generator* in the sequel. Now for the goal ← Append([1,2,3|x], y, z, 3) the following resultants are obtained.

Append([1,2,3], y, [1,2,3|y], 3) ←
Append([1,2,3,u|x'], y, [1,2,3,u|z'], 3) ← Append(x', y, z')

Observe that the partial input data has been completely consumed in the unfolding process. In fact, in this example, one more unfolding step has been performed than is actually required to obtain an "optimal" specialisation, but this is due to the fact that the goal has been unfolded non-deterministically. In some cases, this non-deterministic unfolding may actually be desirable, but this is an orthogonal issue to termination (this issue will be re-examined in Section 7).

Furthermore, note that the SetDepth predicates must now be redefined to assign depths to unbounded atoms. Also a predicate such as SetDepth_A(x, z, d) must be defined such that d gets instantiated to the *maximum* of the minimum lengths of the lists x and z to ensure a maximal amount of unfolding. Note that this maximum will always be finite.

5 Deriving Depth Bounds from Level Mappings

The above transformations rely on a sounding analysis to determine the depths of derivations or unfoldings. Such an analysis may be based on exisiting termination

analyses which derive level mappings. To establish the link with the termination literature the *depth* argument in an atom during *unfolding* may simply be chosen to be the *level* of the atom with respect to some level mapping used in a termination proof. Whilst, in principle a depth bound for unfolding may be derived from any level mapping, in practice this can lead to excessive unfolding. The following example illustrates this.

Example 1. Consider again the Append program and the level mapping $|.|$ defined by $|\mathsf{Append}(x, y, z)| = 3 * |x|_{\text{list-length}}$. The program can be proven to be recurrent wrt $|.|$ and thus goals of the form Append(x, y, z) where x is a rigid list are guaranteed to terminate. If the upper bound on the number of derivation steps in a computation is defined to be equal to the level-mapping, a gross over-approximation is obtained. Given the goal $\leftarrow$ Append([1|x], y, z), the number of derivation steps will be estimated as three. Non-determinate unfolding wrt the clauses app_1^*, app_2^* and app_3^* then produces the following resultants (with the depth bounds removed)

Append([1], y, [1|y]) $\leftarrow$
Append([1,u], y, [1,u|y]) $\leftarrow$
Append([1,u,v], y, [1,u,v|y]) $\leftarrow$
Append([1,u,v,w|x], y, [1,u,v,w|z]) $\leftarrow$ Append(x, y, z)

This specialisation is clearly undesirable. The problem can be fixed here by using a determinate unfolding rule, but this may not always be the case. A more general solution is to consider the difference in the level mappings between the head and the (mutually recursive) body atoms. By subtracting the difference on each recursive call the number of unfolding steps may be accurately controlled.

Example 2. Consider clause app_2 of the Append program and the level mapping $|.|$ defined in example 1. Since

$$|\mathsf{Append}([\mathsf{u}|\mathsf{x}], \mathsf{y}, [\mathsf{u}|\mathsf{z}])| - |\mathsf{Append}(\mathsf{x}, \mathsf{y}, \mathsf{z})| = (3 \times |\mathsf{x}|_{\text{list-length}} + 3) - (3 \times |\mathsf{x}|_{\text{list-length}}) = 3$$

the clause app_2 may be transformed into the clause $app_2^\dagger$ below. Then an Append/3 atom whose size is measured wrt $|.|$ can be unfolded wrt app_1^*, $app_2^\dagger$ and app_3^* resulting in the desired specialisation.

$app_2^\dagger$ Append([u|x], y, [u|z], d) $\leftarrow$ d $\geq$ 0 $\wedge$ Append(x, y, z, d $-$ 3).

□

It is often the case that the head and recursive body atoms in a predicate contain distinct variables and thus the difference in their levels is an expression over these variables. Such expressions can often be reduced to a constant using interargument relationships.

Example 3. Consider the well known naive reverse predicate below.

rev_1 Reverse([], []).
rev_2 Reverse([x|xs], [y|ys]) $\leftarrow$
Delete(y, [x|xs], zs) $\wedge$
Reverse(zs, ys).

Given the interargument relationship $\{\mathsf{Delete}(x, y, z) \mid y = z + 1\}$ and the level mapping $|.|$ defined by $|\mathsf{Reverse}(x, y)| = |x|_{len} + 1$ and $|\mathsf{Delete}(x, y, z)| = |y|_{len}$ the difference in the levels between the head and the recursive call of the clause rev_2 is $(1 + |\mathsf{xs}|_{len} + 1) - (|\mathsf{zs}|_{len} + 1) = (1 + |\mathsf{zs}|_{len} + 1) - (|\mathsf{zs}|_{len} + 1) = 1$. Note that this may be automatically derived using constraint technology. □

One problem remains in this example. For any goal $\leftarrow$ Reverse(x, y), the level mapping |.| over-approximates the number of unfolding steps by 1 each time which may lead to sub-optimal specialisation. Careful examination of the termination literature reveals that level mappings involving additive constants such as |.| are needed in termination proofs where the recursive structure of the program is not fully exploited, such as in a proof of recurrency ([13]). For example, to prove that the Reverse predicate is recurrent wrt |.| the inequality

$$|\mathsf{Reverse([x|xs], [y|ys])}| > |\mathsf{Delete(y, [x|xs], zs)}|$$

must hold and hence |.| must be defined by $|\mathsf{Reverse}(x, y)| = |x|_{len} + 1$ rather than $|\mathsf{Reverse}(x, y)| = |x|_{len}$. In [37] the class of semi delay recurrent programs was introduced which allowed termination proofs to be based on the recursive structure of a program. Additive constants are seldom needed in such proofs. Indeed, for directly recursive programs, they are completely unnecessary and for mutually recursive programs they can be minimised. The Reverse predicate, for example, is semi delay recurrent wrt the level mapping defined as in example 3 but with $|\mathsf{Reverse}(x, y)| = |x|_{len}$. It is straightforward to adapt existing termination analyses to derive these simpler level mappings which can then be used to give an accurate measure of the number of unfolding steps required for a given goal.

It may happen that the difference in levels between head and body atoms is not a constant, but is bounded by a constant n. In this case, it is safe to take the bound n as the difference as this will alow a large (though not necessarily maximal) amount of unfolding.

Finally the most problematic case arises when the difference is (bounded by) a variable expression which cannot be reduced to a constant. Here it may be possible to track the sizes of the relevant variables. This involves only a few extra arithmetic operations and *not* the calculation of a large number of term sizes and so incurs only a small performance penalty.

Example 4. Consider the Match program

m_1 Match([], _, _, _).
m_2 Match([p|ps], [p|ts], p, t) $\leftarrow$
    Match(ps, ts, p, t).
m_3 Match([p|v], [q|w], p, [x|t]) $\leftarrow$
    p $\neq$ q $\wedge$
    Match(p, t, p, t).

and the level mapping |.| defined by $|\mathsf{Match}(w, x, y, z)| = |x|_{len} + (|z|_{len})^2$. The difference between the head of the clause m_3 and its recursive body atom is not a constant:

$$\begin{aligned} |\mathsf{Match([p|v], [q|w], p, [x|t])}| - |\mathsf{Match(p, t, p, t)}| &= (1 + w + 1 + 2t + t^2) - (t + t^2) \\ &= (1 + w) + (1 + t) \end{aligned}$$

where $w = |\mathsf{w}|_{len}$ and $t = |\mathsf{t}|_{len}$. Tight control of the unfolding process can still be achieved however by transforming Match into the following, where extra arguments are added to track the sizes of x and z which in turn can be used to calculate a more accurate depth for each recursive call.

m_1^* Match([], _, _, _, (size_2, size_4, d)) $\leftarrow$
    d $\geq$ 0 $\wedge$ $\mathsf{size}_2 \geq 0 \wedge \mathsf{size}_4 \geq 0$.

m_2^* Match([p|ps], [p|ts], p, t, (size_2, size_4, d)) ←
    $\mathsf{d} \geq 0 \wedge \mathsf{size}_2 \geq 0 \wedge \mathsf{size}_4 \geq 0 \wedge$
    Match(ps, ts, p, t, ($\mathsf{size}_2 - 1$, size_4, $\mathsf{d} - 1$)).
m_3^* Match([p|_], [q|_], p, [_|t], (size_2,size_4,d)) ←
    $\mathsf{d} \geq 0 \wedge \mathsf{size}_2 \geq 0 \wedge \mathsf{size}_4 \geq 0 \wedge$
    $\mathsf{p} \neq \mathsf{q} \wedge$
    Match(p, t, p, t, ($\mathsf{size}_4 - 1$, $\mathsf{size}_4 - 1$, $\mathsf{d} - \mathsf{size}_2 - \mathsf{size}_4$)).

In this program, the argument d keeps track of the level of each Match atom. The level of the recursive call of clause m_3^* is calculated from the level of the head by subtracting the sizes of the second and fourth head arguments. The necessary expression (i.e. $\mathsf{d} - \mathsf{size}_2 - \mathsf{size}_4$) can be obtained automatically, using, for example, constraint technology, from the difference in the levels of the head and the recursive call as calculated above (note that the size of the second argument in the head is $1 + w$ and the size of the fourth argument is $1 + t$). □

It is not clear when such a transformation would be generally applicable. It is important to note, however, that finiteness can always be guaranteed; the problems raised above relate only to the quality of the specialisation and, as mentioned earlier, this is also dependent on the control of determinacy. Although this has been touched upon in [19] this is still a relatively unexplored area in the context of partial deduction. Many of the problems above may disappear altogether with the right balance of bounded and determinate unfolding. Finally, note that the problem of deriving a tight upper bound on the number of derivation steps in a computation is also useful in the context of cost analysis [14].

6 Offline versus Online Unfolding

This section compares the power of sonic partial deduction with existing online techniques. The most interesting conclusion of this study is that the choice of an offline approach does not necessarily entail the sacrifice of unfolding potential. On the contrary, in some cases the unfolding behaviour is better with the proposed method than with the most recent online ones. This is demonstrated through some simple examples which illustrate known problematic unfolding issues [35].

6.1 Measure Functions and Level Mappings

Online unfolding methods, e.g. [35, 8, 36] use *measure functions* to assign *weights* to atoms and goals. Unfolding is controlled by ensuring that weights are strictly decreasing at each unfolding step. In the seminal online work [8], weights were assigned to individual atoms using *set based* measure functions of the form

$$|p(t_1, \ldots, t_n)|_{p,S} = |t_{a_1}| + \ldots + |t_{a_m}|$$

where $S = \{a_1, \ldots, a_m\} \subseteq \{1, \ldots, n\}$ and $|t|$ counts the number of (non 0-ary) functors in the term t. The subset S of argument positions for each predicate is determined dynamically during the unfolding process. Clearly, such a function corrseponds to a restricted form of level mapping and in principle, the level mapping could be derived *a priori* using static analysis. Much depends of course on the power of the anlaysis and also to what extent the decreasing weights of goals is dependent on the program input rather than the structure of the program itself. In many cases, however, current termination analysis techniques are able to derive exactly the same level mappings that are obtained through online unfolding.

6.2 Lexicographical Priorities

Set based measure functions can lead to overly restrictive unfolding as the following example from [35] illustrates.

Example 5. Consider the ProduceConsume program

pc_1 ProdCons([x|xs], []) ←
        ProdCons(xs, [x]).
pc_2 ProdCons(x, [y|ys]) ←
        ProdCons(x, ys).

and the goal ← ProdCons([1,2|xs], []). Figure 2 depicts a finite incomplete SLD-tree illustrating the desired unfolding for this goal (the additional third argument in each atom should be ignored at this point). As [35] points out, there is no subset S for which the set based measure function $|.|_{\mathsf{ProdCons},S}$ is decreasing for each successive atom in this tree (excluding the last node).

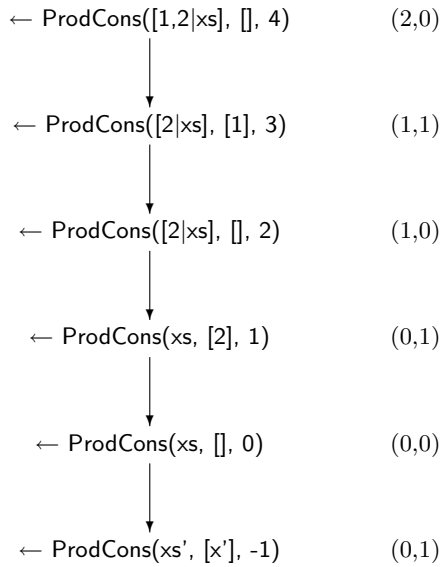

Fig. 2. Unfolding of ← ProdCons([1,2|xs], [], 4)

In order to obtain the desired unfolding, Martens and De Schreye refine their measure functions by introducing the notion of a *partition based* measure function. Such a function maps an atom to an ordered n-tuple where each element in the tuple is obtained by applying a set based measure function to the atom. By using the lexicographical ordering to compare n-tuples, this refinement effectively allows priorities to be assigned amongst the arguments of an atom. Figure 2 shows the 2-tuples assigned to the atoms in the SLD-tree by the function $|.|_{\mathsf{ProdCons},(\{1\},\{2\})}$ defined by

$$|\mathsf{ProdCons}(x,y)|_{\mathsf{ProdCons},(\{1\},\{2\})} = (|x|_{\mathsf{ProdCons},\{1\}}, |y|_{\mathsf{ProdCons},\{2\}})$$

The problem is easily handled within the framework proposed here by the choice of a level mapping which ensures termination. The above program is recurrent wrt the level mapping $|.|$ defined by $|\mathsf{ProdCons}(x, y)| = 2 * |x|_{len} + |y|_{len}$. Then

$$|\mathsf{ProdCons([x|xs], [])}| - |\mathsf{ProdCons(xs, [x])}| = 1$$

$$|\mathsf{ProdCons(x, [y|ys])}| - |\mathsf{ProdCons(x, ys)}| = 1$$

and the clauses pc_1 and pc_2 can be transformed into

pc_1^* ProdCons([x|xs], [], d) ←
    ProdCons(xs, [x], d − 1).
pc_2^* ProdCons(x, [y|ys], d) ←
    ProdCons(x, ys, d − 1).

Now $|\mathsf{ProdCons([1,2|xs], [])}| = 4$ and the goal ← ProdCons([1,2|xs], [], 4) can be unfolded wrt the clauses pc_1^*, pc_2^* and a leaf generator to produce the SLD-tree depicted in figure 2. Notice how the priority assigned to the first argument of ProdCons/2 by the lexicographical ordering is captured by the co-efficient 2 in the level mapping $|.|$. In fact, exactly the same result may be obtained using any other level mapping $|.|'$ defined by $|\mathsf{ProdCons}(x, y)|' = a * |x|_{len} + b * |y|_{len}$ where $a > b > 0$ are arbitrary integers. Of course, the generating extension and goal are different in each case. Also note that such a level mapping can be automatically derived using current termination analysis technology, e.g. [15].

6.3 Well-quasi Orders and Homeomorphic Embedding

It turns out that, for the online approach, well-founded orders as used in Sections 6.1 and 6.2 are sometimes too rigid or (conceptually) too complex. Recently, *well-quasi orders* have therefore gained popularity to ensure online termination of program manipulation techniques ([5, 42, 43, 21, 24, 31, 1, 26, 45, 32]).

Definition 1 (well-quasi order). An ordered set $S(\leq)$ is called *well-quasi ordered* iff for any infinite sequence $e_1, e_2, \ldots$ of elements of S, there exist elements e_i and e_j with $i < j$ such that $e_i \leq e_j$. □

The additional power of well-quasi orders stems from the fact that uncomparable elements are allowed within sequences (while approaches based upon well-founded orders have to impose strict decreases). On the formal side, [29] shows that a rather simple well-quasi approach—the *homeomorphic embedding* relation $\trianglelefteq$— is strictly more powerful than any online approach using monotonic well-founded orders or simplification orders. This covers the approaches of [9, 36] and [35] as described in Sections 6.1 and 6.2. Furthermore, there is no well-founded order—monotonic or not—which is strictly more powerful than $\trianglelefteq$. In practice this means that there will be cases where $\trianglelefteq$ is more powerful than our sonic approach based upon well-founded orders.

Nonetheless, well-quasi orders are more costly to implement (at every unfolding step, a comparison is required with *every* ancestor while well-founded orders only require a comparison with the covering ancestor due to transitivity). Moreover, the well-founded orders used by the sonic approach are *not restricted to be monotonic* and do not have to be simplification orders. They are thus incomparable in power to $\trianglelefteq$. For example, the list-length norm $|.|_{len}$ is neither monotonic nor a simplification order, and indeed, given $t_1 = [1, 2, 3]$ and $t_2 = [[1, 2, 3], 4]$ then $|t_1|_{len} = 3 > |t_2|_{len} =$

2 although $t_1 \trianglelefteq t_2$ (because t_1 can be obtained from t_2 by striking out part of the term). In other words $|.|_{len}$ will admit the sequence t_1, t_2 while $\trianglelefteq$ does not. As will be shown below, there are other cases where the sonic approach is more powerful than using homeomorphic embedding on covering ancestors.

6.4 Coroutining

The increased power offered by partition based measure functions can still be insufficient when unfolding under a coroutining computation rule. The following example, again from [35], illustrates the problem.

Example 6. Consider the program Co-ProduceConsume below

```
cpc1  ProduceConsume(x, y) ←
          Produce(x, y) ∧
          Consume(y).

cpc2  Produce([], []).
cpc3  Produce([x|xs], [x|ys]) ←
          Produce(xs, ys).

cpc4  Consume([]).
cpc5  Consume([x|xs]) ←
          Consume(xs).
```

and the goal ← ProduceConsume([1,2|x], y). Figure 3 illustrates the desired unfolding for this goal and program (again the additional third argument in each atom should be momentarily ignored). Observe that any (sensible) measure function which only considers the selected atom when assigning a weight to a goal will map the two goals containing the selected atoms Consume([1|y']) and Consume([2|y"]) to the same weight and consequently unfolding would stop on reaching the second of these goals. Observe that an unfolding rule based upon $\trianglelefteq$ *would* allow the Consume([2|y"]) to be unfolded (Consume([1|y"]) $\not\trianglelefteq$ Consume([2|y"]) as 1 and 2 are uncomparable). However, if the initial goal is slightly changed to ProduceConsume([1,1|x], y) then the same problem also arises for $\trianglelefteq$ (now Consume([1|y"]) $\trianglelefteq$ Consume([1|y"]) and further unfolding is prevented). □

The solution proposed in [35] is to further refine partition based measure functions to take into account other atoms in a goal besides the selected one. The details are somewhat complicated and space restrictions prohibit a detailed description here. Figure 3 shows one possible assignment of weights to the goals in the SLD-tree under the scheme of [35]. The weight associated to each goal is a 2-tuple where the first argument of the tuple is the size of the first argument of the Produce atom in the goal and the second argument is the size of the (first) argument of the Consume atom in the goal. The symbol $\perp$ is used to register the disappearance of the Produce atom, and in addition the ordering on the natural numbers is extended by defining $\perp < 0$.

An offline approach may exploit information from a static analysis to accurately control the unfolding in this example. In particular, interargument relationships allow depth information to be shared between the coroutining atoms in the computation. The interargument relationships

$$\{\text{ProduceConsume(x, y)} \mid x' = y'\} \text{ and } \{\text{Produce(x, y)} \mid x' = y'\}$$

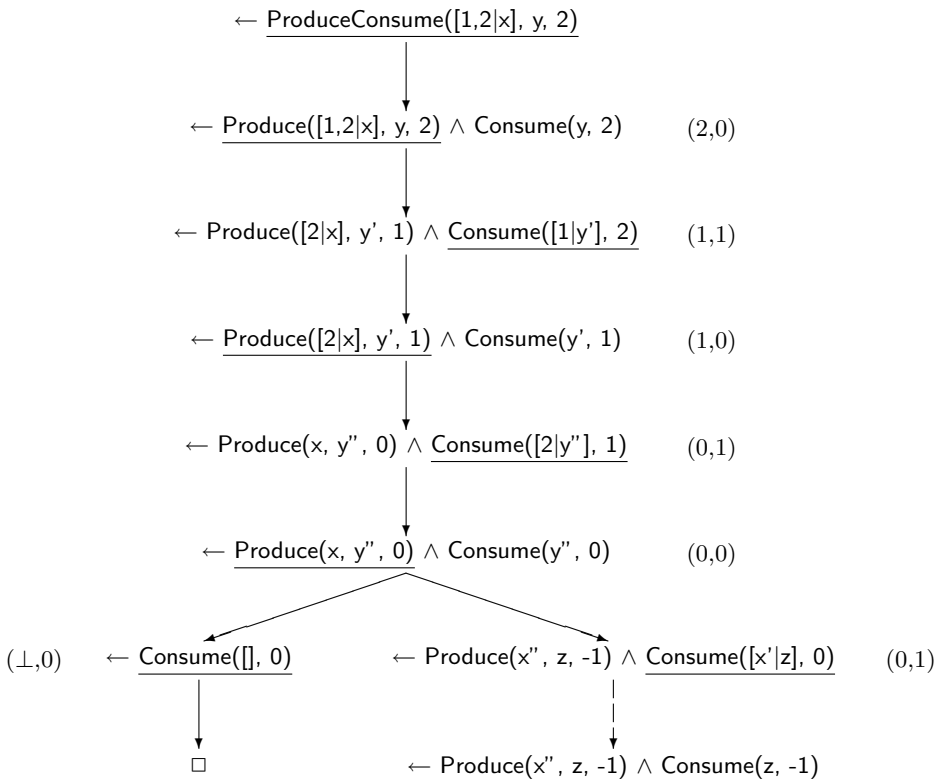

Fig. 3. Unfolding of ← ProduceConsume([1,2|x], y, 2)

may be derived for the program where $x' = |\mathsf{x}|_{len}$ and $y' = |\mathsf{y}|_{len}$. Let $|.|$ be the level mapping defined by $|\mathsf{Produce}(x, y)| = |x|_{len}$ and $|\mathsf{Consume}(y)| = |y|_{len}$. Then for any *successful* refutation of the goal ← Produce(x, y) ∧ Consume(y) the equation $|\mathsf{Produce(x, y)}| = |\mathsf{x}|_{len} = |\mathsf{y}|_{len} = |\mathsf{Consume(y)}|$ must hold. Hence the program can be transformed into the following (with leaf generators for Produce/3 and Consume/2).

cpc^* ProduceConsume(x, y) ←
SetDepth_PC(x, y, d) ∧ ProduceConsume(x, y, d).

cpc_1^* ProduceConsume(x, y, d) ←
Produce(x, y, d) ∧ Consume(y, d).

$prod_1^*$ Produce([], [], d) ← d ≥ 0.
$prod_2^*$ Produce([x|xs], [x|ys], d) ← d ≥ 0 ∧ Produce(xs, ys, d − 1).

$cons_1^*$ Consume([], d) ← d ≥ 0.
$cons_2^*$ Consume([x|xs], d − 1) ← d ≥ 0 ∧ Consume(xs, d).

In this program, the predicate SetDepth_PC(x, y, d) is effectively defined by the equation $d = max(|\mathsf{Produce(x, y)}|, |\mathsf{Consume(y)}|)$. By choosing the maximum of the levels of the two atoms (which is always finite - see section 4) the greatest potential for unfolding is obtained. Thus the initial goal ← ProduceConsume([1,2|x], y) gives $max(|\mathsf{Produce([1,2|x], y)}|, |\mathsf{Consume(y)}|) = max(2, 0) = 2$ and consequently the goal ← Produce([1,2|x], y, 2) ∧ Consume(y, 2) is obtained. Unfolding this goal wrt the above

program leads to the construction of the whole SLD-tree depicted in figure 3. Using the context considering partition based measure functions of [35] the final unfolding step on the right hand branch of the tree (indicated by the dashed arrow) is not permitted since the weight of this goal is the same as the weight of its direct covering ancestor ← Produce(x, y'', 0) ∧ Consume([2|y''], 1).

The key issue here is not that a single extra unfolding step is obtained in this example but the fact that this demonstrates that the unfolding capability of an offline technique may surpass that of an online one and the reason for this. The "sharing" of depth information between atoms is possible through the use of inter-argument relationships which describe the success set of the program. Information relating to the success set is not available to a (pure) online technique. Thus in sonic partial deduction a strictly broader context is considered than in the online case when making unfolding decisions. Finally it is worth remarking that the derivation of interargument relationships forms a core part of many of the termination analyses found in the literature.

6.5 Back Propagation

A generating extension for the naive reverse program using Append is shown below.

rev^* Rev(x, y) ←
    SetDepth_R(x, y, d) ∧
    Rev(x, y, d).

rev_1^* Rev([], [], d) ←
    d ≥ 0.
rev_2^* Rev([x|xs], y, d) ←
    d ≥ 0 ∧
    Rev(xs, z, d − 1) ∧
    App(z, [x], y, d − 1).
rev_3^* Rev(x, y, d) ←
    d < 0 ∧
    Rev(x, y, _).

Unfolding the goal Rev([1,2|x], y) wrt this program results in an SLD-tree with associated resultants $r_1, \ldots, r_4$ below. To give some idea of how these are obtained, the SLD-derivation associated to r_2 is roughly depicted in figure 4.

r_1 Rev([1,2|x], y) ← x = [] ∧ y = [2,1].
r_2 Rev([1,2|x], y) ← x = [a|b] ∧ y = [2,1] ∧ Rev(b,c) ∧ App(c,[a],[]).
r_3 Rev([1,2|x], y) ← x = [a|b] ∧ y = [c,1] ∧ App(d,[2],[]) ∧ Rev(b,e) ∧
    App(e,[a],[c|d]).
r_4 Rev([1,2|x], y) ← x = [a|b] ∧ y = [c,d|e] ∧ App(f,[1],e) ∧ App(g,[2],[d|f]) ∧
    Rev(b,h) ∧ App(h,[a],[c|g]).

Observe that r_2 and r_3 both contain atoms of the form App(x, [y], []) in their right hand sides. These atoms clearly lead to failure but this is not identified during the unfolding process. The decision to leave them as residual atoms takes place before they become instantiated enough to be unfolded. More precisely, when one of these atoms is first encountered it is of the form App(x, [y], z) and should not be unfolded further since there is danger of non-termination. Later in the computation z becomes bound to [] but the atom App(x, [y], []) is no longer a candidate for

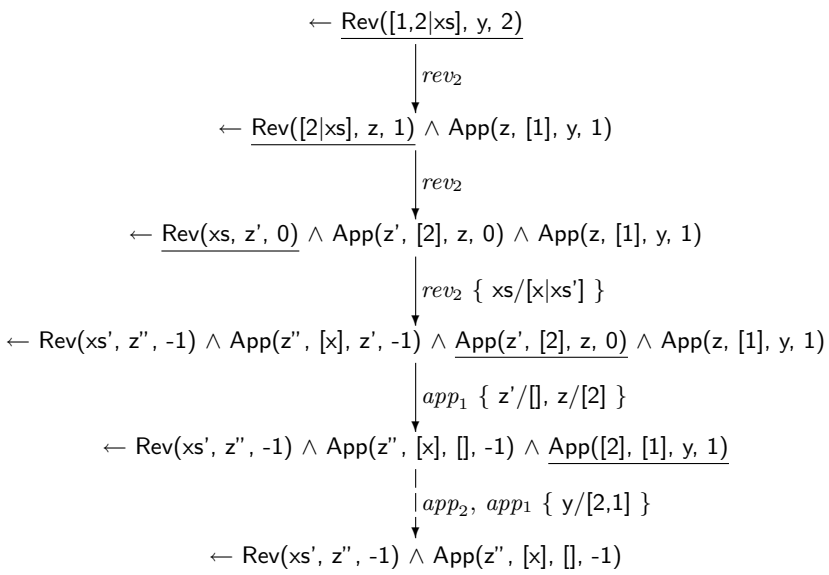

Fig. 4. Unfolding of ← Rev([1,2|xs], y, 2)

selection. This problem also arises in the online approach when using the measure functions described in the previous sections to control unfolding [3]. It is termed the *back propagation* problem in [35] since it is caused by a reverse flow of data.

To solve the problem [35] suggests yet another, even more complicated measure function refinement. The details of this refinement are not presented and it is not clear to what extent it satisfactorily deals with the problem. The solution in the present context is much simpler and consists of always unfolding any atoms that are bounded. Such atoms will lead to a (possibly empty) resultant whose atoms have predicate symbols lower down in the predicate dependency graph than the predicate symbol of the initial atom. It might also be possible to unfold these atoms further or as a result other atoms in the original resultant may have become bounded and can also be selected for unfolding. Note that this process is guaranteed to terminate.

Here again a greater unfolding potential is realised through the availability of global information, i.e. the knowledge that a bounded atom can be safely unfolded a finite number of steps. A pure online technique does not have this "look ahead" capability; it can only compare the present goal with the ones it has encountered in the past with no clue as to what may occur in the future computation.

6.6 Other related work

Loop checking Early work on termination in logic programming focused on detecting loops at runtime ([7], [11], [12], [39], [40], [44]). Simple adaptations of these techniques were proposed ([3], [4]) for controlling unfolding during partial deduction. [4], for example, give four criteria for controling the unfolding. At each unfolding step the selected literal is compared with the one previously selected on the same branch of the SLDNF-tree and unfolding is halted if the descendent literal is a

[3] The unfolding and the resultants obtained are slightly different, but the resultants still contain atoms of the form App(x, [y], []) in their right-hand sides.

variant of, an instance of, more general than or unifies with the ancestor literal. As illustrated in [8] these criteria are not comprehensive enough to prevent infinite unfolding. [6] describes some so called *complete* loop checks which ensure termination of unfolding. It is shown in [5], however, that it is not enough to look only at the selected literal; the context of the goal is needed, which makes the check more expensive.

As it is, the majority of loop checks rely on comparing the current goal with every preceding goal in the derivation, resulting in a number of checks which is quadratic in the length of the derivation. Some notable exceptions include the Tortoise and Hare technique of [44], which is not actually complete, and those proposed by [5]. Of these, it is suggested that the "triangular" loop check is perhaps, in general, the most efficient though this requires something on the order of $5n$ checks for a derivation of length n. Since each check can involve the comparison of goals, atom for atom, term for term, the cost of these loop checking techniques is still quite high.

Another major disadvantage of the linear time loop checks proposed in [5] is their random nature. Since they do not compare all goals, and are not tailored specifically to suit a given program and goal, when a loop occurs, it is largely a matter of luck as to when it is detected. In terms of partial deduction, this can lead to an unneccessary explosion of the search space during unfolding.

Finally, it may be remarked that these techniques are all designed to detect loops at run-time which in the partial deduction context translates as an online approach to unfolding. They cannot be used, for example, to ensure in advance, that a given goal, or class of goals can be completely unfolded.

Finiteness Analysis Offline partial evaluation has been studied extensively in functional programming though for some time consideration of termination was largely neglected. The partial evaluation stage is preceded by a binding time analysis which annotates each argument in the program as either static or dynamic. Functions with static arguments can be evaluated whilst residual code is produced for those with dynamic ones. The termination issue is addressed by *generalising* variables, that is by changing their binding time annotation from static to dynamic ([25, 2]). This occurs whenever execution of a piece of static code may lead to non-termination. Whilst this works well for functional programs, the static/dynamic divisions do not translate well for logic programs (see section **??**) and thus this approach to ensuring termination is not generally suitable for logic programming.

6.7 Offline vs. Online Conclusion

This section has compared sonic partial deduction with the state of the art online unfolding techniques as described in [35]. It has been shown that the approach is able to handle a variety of examples which are known to present difficulties in unfolding. The method is able to handle these examples in a uniform manner, whereas the work of [35] requires the use of increasingly complex measure functions to handle them. This increasing complexity introduces with it increasing overhead as well as the growing risk of programmer error in the actual coding. This is an important issue when considering the construction of a tool which one would like to prove terminating.

Not only is the approach much simpler, it also offers potential for unfolding unfulfilled by online methods. The reason why offline techniques can permit more

unfolding than online ones is the fact that they consider the global context. A global analysis can infer information which may not be avaialble locally when deciding on a particular atom to unfold. A number of fairly complex examples have been examined in the previous sections. The following is a simpler example:

```
UpToN(n, n, [n]).
UpToN(x, n, [x|xs]) ←
    x < n ∧
    UpToN(x + 1, n, xs).
```

Unfolding the goal ← UpToN(1, 3, x) leads to two other goals: ← UpToN(2, 3, x) and ← UpToN(3, 3, x). The only difference between the atoms in these goals is in the first argument which is increasing in value. To determine that the sequence of goals is finite (under a left-to-right computation rule) requires the global information that the first argument is bounded by the second argument.

Of course, an online technique may still be able to make refined unfolding decisions based on the availability of concrete data, not available to an offline one. Clearly, then, a more powerful technique may be obtained by a combination of these approaches.

7 Experiments and Benchmarks

To gauge the efficiency and power of the sonic approach, a prototype implementation has been devised and integrated into the ECCE partial deduction system ([27, 28, 32]). The latter is responsible for the global control and code generation and calls the sonic prototype for the local control. A comparison has been made with ECCE under the default settings, i.e. with ECCE also providing the local control using its default unfolding rule (based on a determinate unfolding rule which uses the homeomorphic embedding relation $\trianglelefteq$ on covering ancestors to ensure termination). For the global control, both specialisers used conjunctive partial deduction ([30, 21]) and characteristic trees ([32]).

All the benchmarks are taken from the DPPD library ([27]) and were run on a Power Macintosh G3 266 Mhz with Mac OS 8.1 using SICStus Prolog 3 #6 (Macintosh version 1.3). Tables 1 and 2 show respectively, the total specialisation times (without post-processing), and the time spent in unfolding during specialisation.[4] In Table 1 the times to produce the generating extensions for the sonic approach are not included, as this is still done by hand. It is possible to automate this process and one purpose of hand-coding the generating extensions was to gain some insight into how this could be best achieved. In any case, in situations where the same program is repeatedly respecialised, this time will become insignificant anyway. The precision of the timings, which were performed using the `statistics/2` predicate, seems to be approximately 1/60th of a second, i.e., about 16.7 ms. Hence "0 ms" in Table 2 should most likely be interpreted as "less than 16 ms". (The runtimes for the residual programs appear in Table 3 in the appendix for referees, which, for a more comprehensive comparison, also includes some results obtained by MIXTUS.)

[4] In the cogen approach, it is very covenient to build trace terms ([20]) for use in the global control and this was incorporated into the sonic prototype. As ECCE employs characteristic trees in a certain format, however, a conversion from trace terms into characteristic trees had to be added. Such a conversion will be unnecessary in an improved version of ECCE which is also able to handle trace terms.

Benchmark	sonic + ECCE	ECCE
advisor	**17** ms	150 ms
applast	83 ms	**33** ms
doubleapp	50 ms	**34** ms
map.reduce	**33** ms	50 ms
map.rev	**50** ms	67 ms
match.kmp	300 ms	**166** ms
matchapp	**66** ms	83 ms
maxlength	**184** ms	200 ms
regexp.r1	**34** ms	400 ms
relative	**50** ms	166 ms
remove	**367** ms	400 ms
remove2	1049 ms	**216** ms
reverse	50 ms	50 ms
rev_acc_type	316 ms	**83** ms
rotateprune	**67** ms	183 ms
ssupply	**34** ms	100 ms
transpose	**50** ms	467 ms
upto.sum1	**33** ms	284 ms
upto.sum2	**50** ms	83 ms

Table 1. Specialisation times (total w/o post-processing)

The sonic prototype implements a more agressive unfolding rule than the default determinate unfolding rule of ECCE. This is at the expense of total transformation time (see Table 1), as it often leads to increased polyvariance, but consequently the speed of the residual code is often improved, as can be seen in Table 3.[5] Default ECCE settings more or less guarantee no slowdown, and this is reflected in Table 3, whereas the general lack of determincay control in the prototype sonic unfolding rule leads to two small slowdowns.

There is plenty of room for improvement, however, on these preliminary results. The sonic approach is flexible enough to allow determinacy control to be incorporated within it, and this extra layer of control could help to guarantee no slowdown. Also, the sonic prototype has been built on the philosophy of "unfold finitely as much as possible". This bull-in-a-china-shop approach actually pays off much better than expected, but the results also indicate that some refinements might also lead to better specialisation times and more efficient residual code. There is plenty of scope for variation within the prototype which would allow these refinements to be made. The only potential problem is in identifying when it would be appropriate to use them.

All in all, the sonic approach provides extremely fast unfolding combined with very good specialisation capabilities. It is surprising that the sonic approach outperformed the (albeit conservative) default unfolding of ECCE. Also observe that the sonic approach even improves upon the `match.kmp` benchmark and passes the KMP test (even better than the online system does). The sonic approach is thus the first offline approach to our knowledge which passes the KMP test. [6] If it were

[5] A more agressive unfolding rule, in conjunctive partial deduction, did not lead to improved speed under compiled code of Prolog by BIM; see [28]. So, this also depends on the quality of the indexing generated by the compiler.

[6] One might argue that the global control is still online. Note, however, that for KMP no generalisation and thus no global control is actually needed.

Benchmark	sonic + ECCE	ECCE
advisor	0 ms	33 ms
applast	0 ms	16 ms
doubleapp	0 ms	0 ms
map.reduce	0 ms	17 ms
map.rev	0 ms	34 ms
match.kmp	0 ms	99 ms
matchapp	0 ms	33 ms
maxlength	0 ms	67 ms
regexp.r1	0 ms	383 ms
relative	0 ms	166 ms
remove	34 ms	201 ms
remove2	33 ms	50 ms
reverse	16 ms	33 ms
rev_acc_type	0 ms	32 ms
rotateprune	0 ms	99 ms
ssupply	0 ms	67 ms
transpose	16 ms	400 ms
upto.sum1	0 ms	168 ms
upto.sum2	0 ms	66 ms

Table 2. Specialisation times (unfolding)

possible to extend the sonic approach to the global control as well, one would hopefully obtain an extremely efficient specialiser producing highly optimised residual code.

8 Conclusion

The majority of termination analyses rely on the derivation of level mappings to prove termination. This paper has described how these level mappings may be used to obtain precise depth bounds for the control of unfolding during partial deduction. Thus, a solid link has been established between the fields of static termination analysis and partial deduction enabling existing and future termination analyses to be used to ensure finiteness of the unfolding process.

Furthermore, the paper has described now such depth bounds can be incorporated in generating extensions. The construction of these forms the foundation of any offline partial deduction method whether it is based on the self-application or the cogen approach. This is the first offline technique which allows arbitrarily partially instantiated goals to be sufficiently unfolded to achieve good specialisation results. The technique can, surprisingly, yield even better specialisation than a pure online technique. This is due to the availability of global information in the unfolding decision making process. It is also, to our knowledge, the first offline approach which passes the KMP test.

The framework admits elegant solutions to some problematic unfolding issues and these solutions are significantly less complex than their online counterparts. Of course, an online technique may still be able to make refined unfolding decisions based on the availabilty of concrete data. This strongly suggests that offline and online methods be combined to achieve maximal unfolding power. Another, possibly more challenging, avenue for further research is to extend the sonic approach for the global control, so that its advantages in terms of efficiency, termination, and specialisation power also apply at the global control level.

References

1. M. Alpuente, M. Falaschi, P. Julián, and G. Vidal. Spezialisation of lazy functional logic programs. In *Proceedings of PEPM'97, the ACM Sigplan Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 151–162, Amsterdam, The Netherlands, 1997. ACM Press.
2. P. Anderson and C. Holst. Termination analysis for offline partial evaluation of a higher order functional language. pages 67–82.
3. K. Apt, R. Bol, and J. W. Klop. On the safe termination of Prolog programs. In G. Levi and M. Martelli, editors, *Proceedings of the Sixth International Conference on Logic Programming*, pages 353–368, Lisbon, 1989. The MIT Press.
4. K. Benkerimi and J. W. Lloyd. A partial evaluation procedure for logic programs. In *NACLP'90*, pages 343–358, 1990.
5. R. N. Bol. *Loop checking in logic programming.* PhD thesis, CWI, Amsterdam, October 1991. CWI Tract 112.
6. R. N. Bol. Loop checking in partial deduction. *The Journal of Logic Programming*, 16(1 & 2):25–46, 1993.
7. D. Brough and A. Walker. Some practical properties of a Prolog interpreter. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, Tokyo, Japan, Nov. 1984.
8. M. Bruynooghe, D. De Schreye, and B. Martens. A general criterion for avoiding infinite unfolding during partial deduction of logic programs. In *ILPS'91*, pages 117–131. MIT Press, 1991.
9. M. Bruynooghe, D. De Schreye, and B. Martens. A general criterion for avoiding infinite unfolding during partial deduction. *New Generation Computing*, 11(1):47–79, 1992.
10. M. Bruynooghe, M. Leuschel, and K. Sagonas. A polyvariant binding-time analysis for off-line partial deduction. In C. Hankin, editor, *Proceedings of the European Symposium on Programming (ESOP'98)*, LNCS 1381, pages 27–41. Springer-Verlag, April 1998.
11. M. A. Covington. Eliminating unwanted loops in Prolog. *SIGPLAN Notices*, 20(1), Jan. 1985.
12. M. A. Covington. A further note on looping in Prolog. *SIGPLAN Notices*, 20(8), Aug. 1985.
13. D. De Schreye and S. Decorte. Termination of logic programs: The never-ending story. *The Journal of Logic Programming*, 19 & 20:199–260, May 1994.
14. S. Debray and N.-W. Lin. Automatic complexity analysis of logic programs. pages 599–613.
15. S. Decorte and D. De Schreye. Demand-driven and constraint-based automatic left-termination analysis for logic programs. pages 78–92.
16. N. Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, 3:69–116, 1987.
17. N. Dershowitz and Z. Manna. Proving termination with multiset orderings. *Communications of the ACM*, 22(8):465–476, 1979.
18. A. Ershov. On Futamura projections. *BIT (Japan)*, 12(14):4–5, 1982. In Japanese.
19. J. Gallagher. Tutorial on specialisation of logic programs. In *Proceedings of PEPM'93, the ACM Sigplan Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 88–98. ACM Press, 1993.
20. J. Gallagher and L. Lafave. Regular approximation of computation paths in logic and functional languages. pages 115–136.
21. R. Glück, J. Jørgensen, B. Martens, and M. H. Sørensen. Controlling conjunctive partial deduction of definite logic programs. In H. Kuchen and S. Swierstra, editors, *Proceedings of the International Symposium on Programming Languages, Implementations, Logics and Programs (PLILP'96)*, LNCS 1140, pages 152–166, Aachen, Germany, September 1996. Springer-Verlag.

22. C. A. Gurr. *A Self-Applicable Partial Evaluator for the Logic Programming Language Gödel*. PhD thesis, Department of Computer Science, University of Bristol, January 1994.
23. J. Jørgensen and M. Leuschel. Efficiently generating efficient generating extensions in Prolog. In O. Danvy, R. Glück, and P. Thiemann, editors, *Proceedings of the 1996 Dagstuhl Seminar on Partial Evaluation*, LNCS 1110, pages 238–262, Schloß Dagstuhl, 1996. Extended version as Technical Report CW 221, K.U. Leuven. Accessible via `http://www.cs.kuleuven.ac.be/~lpai`.
24. J. Jørgensen, M. Leuschel, and B. Martens. Conjunctive partial deduction in practice. pages 59–82. Also in the Proceedings of BENELOG'96. Extended version as Technical Report CW 242, K.U. Leuven.
25. C. Kehler Holst. Finiteness Analysis. In J. Hughes, editor, *FPCA'91*, volume 523, pages 473–495, Cambridge, Massachusetts, USA, Aug. 1991. Springer-Verlag.
26. L. Lafave and J. Gallagher. Constraint-based partial evaluation of rewriting-based functional logic programs. pages 168–188.
27. M. Leuschel. The ECCE partial deduction system and the DPPD library of benchmarks. Obtainable via `http://www.cs.kuleuven.ac.be/~dtai`, 1996.
28. M. Leuschel. *Advanced Techniques for Logic Program Specialisation*. PhD thesis, K.U. Leuven, May 1997. Accessible via `http://www.ecs.soton.ac.uk/~mal`.
29. M. Leuschel. On the power of homeomorphic embedding for online termination. In G. Levi, editor, Static Analysis. *Proceedings of SAS'98*, LNCS 1503, pages 230–245, Pisa, Italy, September 1998. Springer-Verlag.
30. M. Leuschel, D. De Schreye, and A. de Waal. A conceptual embedding of folding into partial deduction: Towards a maximal integration. In M. Maher, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming JICSLP'96*, pages 319–332, Bonn, Germany, September 1996. MIT Press.
31. M. Leuschel and B. Martens. Global control for partial deduction through characteristic atoms and global trees. In O. Danvy, R. Gl{"uck, and P. Thiemann, editors, *Proceedings of the 1996 Dagstuhl Seminar on Partial Evaluation*, LNCS 1110, pages 263–283, Schloß Dagstuhl, 1996. Extended version as Technical Report CW 220, K.U. Leuven. Accessible via `http://www.cs.kuleuven.ac.be/~lpai`.
32. M. Leuschel, B. Martens, and D. De Schreye. Controlling generalisation and polyvariance in partial deduction of normal logic programs. *ACM Transactions on Programming Languages and Systems*, 20(1):208–258, January 1998.
33. J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987.
34. J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *The Journal of Logic Programming*, 11(3& 4):217–242, 1991.
35. B. Martens and D. De Schreye. Automatic finite unfolding using well-founded measures. *Journal of Logic Programming*, 28(2):89–146, 1996.
36. B. Martens, D. De Schreye, and T. Horváth. Sound and complete partial deduction with unfolding based on well-founded measures. *Theoretical Comput. Sci.*, 122(1–2):97–117, 1994.
37. J. Martin and A. King. Generating Efficient, Terminating Logic Programs. In *TAPSOFT'97*. Springer-Verlag, 1997.
38. T. Mogensen and A. Bondorf. Logimix: A self-applicable partial evaluator for Prolog. In K.-K. Lau and T. Clement, editors, Logic Program Synthesis and Transformation. *Proceedings of LOPSTR'92*, pages 214–227. Springer-Verlag, 1992.
39. D. Nute. A programming solution to certain problems with loops in Prolog. *SIGPLAN Notices*, 20(8), Aug. 1985.
40. D. Poole and R. Goebel. On eliminating loops in Prolog. *SIGPLAN Notices*, 20(8), Aug. 1985.
41. S. A. Romanenko. A compiler generator produced by a self-applicable specializer can have a surprisingly natural and understandable structure. In D. Bjørner, A. P. Ershov, and N. D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 445–463. North-Holland, 1988.

42. D. Sahlin. Mixtus: An automatic partial evaluator for full Prolog. *New Generation Computing*, 12(1):7–51, 1993.
43. M. H. Sørensen and R. Glück. An algorithm of generalization in positive supercompilation. In J. W. Lloyd, editor, *Proceedings of ILPS'95, the International Logic Programming Symposium*, pages 465–479, Portland, USA, December 1995. MIT Press.
44. A. van Gelder. Efficient loop detection in Prolog using the Tortoise-and-Hare technique. *Journal Logic Programming*, 4(1):23–32, Mar. 1987.
45. W. Vanhoof and B. Martens. To parse or not to parse. pages 322–342. Also as Technical Report CW 251, K.U.Leuven.

Benchmark	Original	sonic + ECCE	ECCE	MIXTUS
advisor	1541 ms	483 ms	426 ms	471 ms
	1	3.19	3.62	
applast	1563 ms	491 ms	471 ms	1250 ms
	1	3.18	3.32	
doubleapp	1138 ms	700 ms	600 ms	854 ms
	1	1.63	1.90	
map.reduce	541 ms	100 ms	117 ms	383 ms
	1	5.41	4.62	
map.rev	221 ms	71 ms	83 ms	138 ms
	1	3.11	2.66	
match.kmp	4162 ms	1812 ms	3166 ms	2521 ms
	1	2.30	1.31	
matchapp	1804 ms	771 ms	1525 ms	1375 ms
	1	2.34	1.18	
maxlength	217 ms	283 ms	208 ms	213 ms
	1	0.77	1.04	
regexp.r1	3067 ms	396 ms	604 ms	
	1	7.74	5.08	
relative	9067 ms	17 ms	1487 ms	17 ms
	1	533.35	6.10	
remove	3650 ms	4466 ms	2783 ms	2916 ms
	1	0.82	1.31	
remove2	5792 ms	4225 ms	3771 ms	3017 ms
	1	1.37	1.54	
reverse	8534 ms	6317 ms	6900 ms	
	1	1.35	1.24	
rev_acc_type	37391 ms	26302 ms	26815 ms	25671 ms
	1	1.42	1.39	
rotateprune	7350 ms	5167 ms	5967 ms	5967 ms
	1	1.42	1.23	
ssupply	1150 ms	79 ms	92 ms	92 ms
	1	14.56	12.50	
transpose	1567 ms	67 ms	67 ms	67 ms
	1	–	–	
upto.sum1	6517 ms	4284 ms	4350 ms	4716 ms
	1	1.52	1.50	
upto.sum2	1479 ms	1008 ms	1008 ms	1008 ms
	1	1.47	1.47	

Table 3. Speed of the residual programs (in ms, for a large number of queries, interpreted code) and Speedups