

# Homeomorphic Embedding for Online Termination

Michael Leuschel  
mal@ecs.soton.ac.uk

Declarative Systems and Software Engineering Group  
Technical Report DSSE-TR-98-11  
October 13 1998

[www.dsse.ecs.soton.ac.uk/techreports/](http://www.dsse.ecs.soton.ac.uk/techreports/)

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

# Homeomorphic Embedding for Online Termination

Michael Leuschel\*

Department of Electronics and Computer Science  
University of Southampton, UK  
e-mail: `mal@ecs.soton.ac.uk`

**Abstract.** Recently well-quasi orders in general, and homeomorphic embedding in particular, have gained popularity to ensure the termination of program analysis, specialisation and transformation techniques. In this paper we investigate and clarify for the first time, both intuitively and formally, the advantages of such an approach over one using well-founded orders. Notably we show that the homeomorphic embedding relation is strictly more powerful than a large class of involved well-founded approaches. We, however, also illustrate that the homeomorphic embedding relation suffers from several inadequacies which are unsatisfactory in contexts, such as logic or functional & logic programming, where logical variables arise. We therefore also present new, extended homeomorphic embedding relations to remedy this problem.

**Keywords:** Termination, Well-quasi orders, Program Analysis, Specialisation and Transformation, Logic Programming, Functional & Logic Programming.

## 1 Introduction

The problem of ensuring termination arises in many areas of computer science and a lot of work has been devoted to proving termination of term rewriting systems (e.g. [9–11, 52] and references therein) or of logic programs (e.g. [8, 55] and references therein). It is also an important issue within all areas of program analysis, specialisation and transformation: one usually strives for methods which are guaranteed to terminate. It can also be an issue in model checking when infinite state systems are involved. One can basically distinguish between two kinds of techniques for ensuring termination:

- *static* techniques, which prove or ensure termination of a program or process *beforehand* (i.e. *off-line*) without any kind of execution, and
- *online* (or *dynamic*) techniques, which ensure termination of a process *during* its execution. (The process itself can of course be, e.g., performing a static analysis.)

Static approaches have less information at their disposal but do not require runtime intervention (which might be impossible). Which of the two approaches

---

\* Part of the work was done while the author was Post-doctoral Fellow of the Fund for Scientific Research - Flanders Belgium (FWO) at the K.U. Leuven, Belgium and at DIKU, University of Copenhagen, Denmark.



problem very similar to global termination of partial deduction. We also believe our discussions are relevant to other areas, such as infinite *model checking* and *theorem proving*, where termination has to be insured in non-trivial ways.

**Depth Bounds** One, albeit ad-hoc, way to solve the local termination problem is to simply impose an arbitrary *depth bound*. Such a depth bound is of course not motivated by any property, structural or otherwise, of the program or goal under consideration. In the context of local termination, the depth bound will therefore typically lead either to too little or too much unfolding.

**Determinacy** Another approach, often used in partial evaluation of functional programs [26] is to (only) expand a tree while it is *determinate* (i.e. it only has one non-failing branch). However, this approach can be very restrictive and in itself does not guarantee termination, as there can be infinitely failing determinate computations at specialisation time. In (strict) functional programs such a condition can be seen as an error in the original program (the corresponding run-time computation will not terminate). In logic programming the situation is somewhat different: a goal can infinitely fail (in a deterministic way) at partial deduction time while its run-time instances finitely fail. In applications like theorem proving, even infinite determinate failures at run-time do not necessarily indicate a programmer’s error: they might simply be due to unprovable statements. This is why, contrary to maybe functional programming, measures in addition to determinacy have to be adopted to ensure local termination.

**Well-founded Orders** Luckily, more refined approaches to ensure local termination exist. The first non-ad-hoc methods [7, 48, 47, 44] in logic and [58, 67] functional programming were based on *well-founded orders*, inspired by their usefulness in the context of static termination analysis. These techniques ensure termination, while at the same time allowing unfolding related to the structural aspect of the program and goal to be specialised, e.g., permitting the consumption of static input within the atoms of  $\mathcal{A}$ .

Formally, well-founded orders are defined as follows:

**Definition 1.** (**wfo**) A (strict) partial order  $>_S$  on a set  $S$  is an *anti-reflexive, anti-symmetric and transitive binary relation* on  $S \times S$ . A sequence of elements  $s_1, s_2, \dots$  in  $S$  is called *admissible wrt  $>_S$*  iff  $s_i > s_{i+1}$ , for all  $i \geq 1$ . We call  $>_S$  a *well-founded order (wfo)* iff there is no infinite admissible sequence wrt  $>_S$

To ensure local termination, one has to find a sensible well-founded order on atoms and then only allow SLDNF-trees in which the sequence of selected atoms is admissible wrt the well-founded order. If an atom that we want to select is not strictly smaller than its ancestors, we either have to select another atom or stop unfolding altogether.

*Example 1.* Let  $P$  be the *reverse* program using an accumulator:

$$\begin{aligned} rev(\square, Acc, Acc) &\leftarrow \\ rev([H|T], Acc, Res) &\leftarrow rev(T, [H|Acc], Res) \end{aligned}$$

A simple well-founded order on atoms of the form  $rev(t_1, t_2, t_3)$  might be based on comparing the term-size (i.e., the number of function and constant symbols) of the first argument. We then define the wfo on atoms by:

$$rev(t_1, t_2, t_3) > rev(s_1, s_2, s_3) \text{ iff } term\_size(t_2) > term\_size(s_2).$$

Based on that wfo, the goal  $\leftarrow rev([a, b|T], \square, R)$  can be unfolded into the goal  $\leftarrow rev([b|T], [a], R)$  and further into  $\leftarrow rev(T, [b, a], R)$  because the term-size of the first argument strictly decreases at each step (even though the overall term-size does not decrease). However,  $\leftarrow rev(T, [b, a], R)$  cannot be further unfolded into  $\leftarrow rev(T', [H', b, a], R)$  because there is no such strict decrease.

Much more elaborate techniques, which e.g. split the expressions into classes, use lexicographical ordering on subsequences of the arguments and even continuously refine the orders during the unfolding process, exist [7, 48, 47, 44] for precise details. These works also present some further refinements on *how to apply* wfo's, especially in the context of partial deduction. For instance, instead of requiring a decrease wrt every ancestor, one can only request a decrease wrt the *covering ancestors*, i.e. one only compares with the ancestor atoms from which the current atom descends (via resolution). Other refinements consist in allowing the wfo's not only to depend upon the selected atom but on the *context* as well [47] or to ignore calls to *non-recursive* predicates. [47] also discusses a way to relax the condition of a "strict decrease" when refining a wfo. (Most of these refinements can also be applied to other approaches, notably the one we will present in the next section.)

However, it has been felt by several researchers that well-founded orders are sometimes too rigid or (conceptually) too complex in an online setting. Recently, well-quasi orders have therefore gained popularity to ensure online termination of program manipulation techniques [5, 59, 61, 38, 39, 14, 27, 2, 29, 1, 65]. Unfortunately, this move to well-quasi orders has never been formally justified nor has the relation to well-founded approaches been investigated. We strive to do so in the first part of this paper and will actually prove that a rather simple well-quasi approach—the *homeomorphic embedding* relation—is strictly more powerful than a large class of involved well-founded approaches. Nonetheless, despite its power, we will show that the homeomorphic embedding is still unsatisfactory when it comes to variables. This paper aims at improving this situation by developing more adequate refinements of the homeomorphic embedding relation.

This paper is structured as follows. In Sections 2 and 3 we provide a gentle introduction to well-quasi orders and then formally compare the power of well-quasi orders with the power of well-founded orders for online termination. In Section 4 we provide some additional investigation, discussing the concept of "near-foundedness" [44]. In Section 5 we show that, despite its power, the homeomorphic embedding is still unsatisfactory when it comes to variables. We provide a first solution, which we then improve in Section 6, notably to be able to cope

with infinite alphabets. Finally in Section 7 we provide some further discussion about applying homeomorphic embedding in the context of metaprogramming.

This paper is an extended and revised version of [35] and [34].

## 2 Well-quasi orders and homeomorphic embedding

Formally, well-quasi orders can be defined as follows.

**Definition 2. (quasi order)** *A quasi order  $\geq_S$  on a set  $S$  is a reflexive and transitive binary relation on  $S \times S$ .*

Henceforth, we will use symbols like  $<$ ,  $>$  (possibly annotated by some subscript) to refer to strict partial orders and  $\leq$ ,  $\geq$  to refer to quasi orders. We will use either “directionality” as is convenient in the context. We also define an *expression* to be either a *term* (built-up from variables and function symbols of arity  $\geq 0$ ) or an *atom* (a predicate symbol applied to a, possibly empty, sequence of terms), and then treat predicate symbols as functors, but suppose that no confusion between function and predicate symbols can arise (i.e., predicate and function symbols are distinct).

**Definition 3. (wbr, wqo)** *Let  $\leq_S$  be a binary relation on  $S \times S$ . A sequence of elements  $s_1, s_2, \dots$  in  $S$  is called admissible wrt  $\leq_S$  iff there are no  $i < j$  such that  $s_i \leq_S s_j$ . We say that  $\leq_S$  is a well-binary relation (wbr) on  $S$  iff there are no infinite admissible sequences wrt  $\leq_S$ . If  $\leq_S$  is a quasi order on  $S$  then we also say that  $\leq_S$  is a well-quasi order (wqo) on  $S$ .*

Observe that, in contrast to wfo’s, non-comparable elements are allowed within admissible sequences. An admissible sequence is sometimes called *bad* while a non-admissible one is called *good*. A well-binary relation is then such that all infinite sequences are good. There are several other equivalent definitions of well-binary relations and well-quasi orders. Higman [20] used an alternate definition of well-binary relations and well-quasi orders in terms of the “finite basis property” (or “finite generating set” in [28]). Both definitions are equivalent by Theorem 2.1 in [20]. A different (but also equivalent) definition of a wqo is (e.g., [30, 66]): A quasi-order  $\leq_V$  is a wqo iff for all quasi-orders  $\preceq_V$  which contain  $\leq_V$  (i.e.  $v \leq_V v' \Rightarrow v \preceq_V v'$ ) the corresponding strict partial order  $<_V$  is a wfo. This property has been exploited in the context of *static* termination analysis to dynamically construct well-founded orders from well-quasi ones and led to the initial use of wqo’s in the offline setting [9, 10]. The use of well-quasi orders in an *online* setting has only emerged recently (it is mentioned, e.g., in [5] but also [59]) and has never been compared to well-founded approaches. There has been some comparison between wfo’s and wqo’s in the offline setting, e.g., in [52] it is argued that (for “simply terminating” rewrite systems) approaches based upon quasi-orders are less interesting than ones based upon a partial orders. In this paper we will show that the situation is somewhat reversed in an online setting. Furthermore, in the online setting, transitivity of a wqo is not really interesting and one can therefore

drop this requirement, leading to the use of wbr's. Later on in Sections 5 and 6 we will actually develop wbr's which are not wqo's.

An interesting wqo is the homeomorphic embedding relation  $\sqsubseteq$ , which derives from results by Higman [20] and Kruskal [28]. It has been used in the context of term rewriting systems in [9, 10], and adapted for use in supercompilation [64] in [61]. Its usefulness as a stop criterion for partial evaluation is also discussed and advocated in [43]. Some complexity results can be found in [63] and [17] (also summarised in [43]).

The following is the definition from [61], which adapts the pure homeomorphic embedding from [10] by adding a rudimentary treatment of variables.

**Definition 4.** ( $\sqsubseteq$ ) *The (pure) homeomorphic embedding relation  $\sqsubseteq$  on expressions is defined inductively as follows (i.e.  $\sqsubseteq$  is the least relation satisfying the rules):*

1.  $X \sqsubseteq Y$  for all variables  $X, Y$
2.  $s \sqsubseteq f(t_1, \dots, t_n)$  if  $s \sqsubseteq t_i$  for some  $i$
3.  $f(s_1, \dots, s_n) \sqsubseteq f(t_1, \dots, t_n)$  if  $\forall i \in \{1, \dots, n\} : s_i \sqsubseteq t_i$ .

The second rule is sometimes called the *diving* rule, and the third rule is sometimes called the *coupling* rule. When  $s \sqsubseteq t$  we also say that  $s$  is *embedded in*  $t$  or  $t$  is *embedding*  $s$ . By  $s \triangleleft t$  we denote that  $s \sqsubseteq t$  and  $t \not\sqsubseteq s$ . By  $s \approx_{\sqsubseteq} t$  we denote that both  $s \sqsubseteq t$  and  $t \sqsubseteq s$ .

*Example 2.* The intuition behind the above definition is that  $A \sqsubseteq B$  iff  $A$  can be obtained from  $B$  by “striking out” certain parts, or said another way, the structure of  $A$  reappears within  $B$ . For instance we have  $p(a) \sqsubseteq p(f(a))$  and indeed  $p(a)$  can be obtained from  $p(f(a))$  by “striking out” the  $f$ . Observe that the “striking out” corresponds to the application of the diving rule 2 (cf. Lemma 2) and that we even have  $p(a) \triangleleft p(f(a))$ . We also have, e.g., that:  $X \sqsubseteq X$ ,  $p(X) \triangleleft p(f(Y))$ ,  $p(X, X) \sqsubseteq p(X, Y)$  and  $p(X, Y) \sqsubseteq p(X, X)$ .

**Proposition 1.** *The relation  $\sqsubseteq$  is a wqo on the set of expressions over a finite alphabet.*

For a complete proof, reusing Higman's and Kruskal's results [20, 28] in a very straightforward manner, see, e.g., [33]. (For constructive proofs of Higman's Lemma [20] see [60, 53]. See also [13] and [57]. Another, non-constructive one can be found in [54].)

To ensure, e.g., local termination of partial deduction, we have to ensure that the constructed SLDNF-trees are such that the selected atoms do *not embed* any of their ancestors (when using a well-founded order as in Example 1, we had to require a *strict decrease* at every step). If an atom that we want to select embeds one of its ancestors, we either have to select another atom or stop unfolding altogether. For example, based on  $\sqsubseteq$ , the goal  $\leftarrow \text{rev}([a, b|T], [], R)$  of Example 1 can be unfolded into  $\leftarrow \text{rev}([b|T], [a], R)$  and further into  $\leftarrow \text{rev}(T, [b, a], R)$  as  $\text{rev}([a, b|T], [], R) \triangleleft \text{rev}([b|T], [a], R)$ ,  $\text{rev}([a, b|T], [], R) \triangleleft \text{rev}(T, [b, a], R)$  and  $\text{rev}([b|T], [a], R) \triangleleft \text{rev}(T, [b, a], R)$ . However,  $\leftarrow \text{rev}(T, [b, a], R)$  cannot be further

unfolded into  $\leftarrow \text{rev}(T', [H', b, a], R)$  as  $\text{rev}(T, [b, a], R) \trianglelefteq \text{rev}(T', [H', b, a], R)$ . Observe that, in contrast to Example 1, we did not have to choose how to measure which arguments. We further elaborate on the inherent flexibility of  $\trianglelefteq$  in the next section.

The homeomorphic embedding relation is also useful for handling structures other than expressions. It has, e.g., been successfully applied in [38, 33, 39] to detect (potentially) non-terminating sequences of characteristic trees. Also,  $\trianglelefteq$  seems to have the desired property that very often only “real” loops are detected and that they are detected at the earliest possible moment (see [43]).

### 3 Comparing wbr’s and wfo’s

#### 3.1 General Comparison

It follows from Definitions 1 and 3 that if  $\leq_V$  is a wqo then  $<_V$  (defined by  $v_1 <_V v_2$  iff  $v_1 \leq_V v_2 \wedge v_1 \not\geq_V v_2$ ) is a wfo, but not vice versa. The following shows how to obtain a wbr from a wfo.

**Lemma 1. (wbr from wfo)** *Let  $<_V$  be a well-founded order on  $V$ . Then  $\preceq_V$ , defined by  $v_1 \preceq_V v_2$  iff  $v_1 \not\succ_V v_2$ , is a wbr on  $V$ . Furthermore,  $<_V$  and  $\preceq_V$  have the same set of admissible sequences.*

This means that, in an online setting, the approach based upon wbr’s is in theory at least as powerful as the one based upon wfo’s. Further below we will actually show that wbr’s are strictly more powerful.

Observe that  $\preceq_V$  is not necessarily a wqo: transitivity is not ensured as  $t_1 \not\prec t_2$  and  $t_2 \not\prec t_3$  do not imply  $t_1 \not\prec t_3$ . Let, e.g.,  $s < t$  denote that  $s$  is strictly more general than  $t$ . Then  $<$  is a wfo [24] but  $p(X, X, a) \not\prec p(X, Z, b)$  and  $p(X, Z, b) \not\prec p(X, Y, a)$  even though  $p(X, X, a) > p(X, Y, a)$ . Also observe that the proof of Lemma 1 requires transitivity of  $<_V$ , i.e., the lemma does not hold simply for a well-founded relation.

Let us now examine the power of one particular wqo, the earlier defined  $\trianglelefteq$ .

#### 3.2 Homeomorphic Embedding and Monotonic Wfo’s

The homeomorphic embedding  $\trianglelefteq$  relation is very flexible. It will for example, when applied to the sequence of covering ancestors, permit the full unfolding of most terminating Datalog programs, the quicksort or even the mergesort program when the list to be sorted is known (the latter poses problems to some static termination analysis methods [55, 40]; for some experiments see Appendix B). Also, the *produce-consume* example from [44] (see also [50]) requires rather involved techniques (considering the context) to be solved by wfo’s. Again, this example poses no problem to  $\trianglelefteq$  (cf. Appendix B).

The homeomorphic embedding  $\trianglelefteq$  is also very powerful in the context of metaprogramming [21]. Notably, it has the ability to “penetrate” layers of (non-ground) meta-encodings (see Appendix B for some computer experiments; see

also Section 7). For instance,  $\trianglelefteq$  will admit the following sequences (where, among others, Example 1 is progressively wrapped into “vanilla” metainterpreters [45, 46, 21] counting resolution steps and keeping track of the selected predicates respectively):

Sequence
$rev([a, b T], [], R) \rightsquigarrow rev([b T], [a], R)$
$solve(rev([a, b T], [], R), 0) \rightsquigarrow solve(rev([b T], [a], R), s(0))$
$solve'(solve(rev([a, b T], [], R), 0), []) \rightsquigarrow solve'(solve(rev([b T], [a], R), s(0)), [rev])$
$path(a, b, []) \rightsquigarrow path(b, a, [a])$
$solve'(solve(path(a, b, []), 0), []) \rightsquigarrow solve'(solve(path(b, a, [a]), s(0)), [rev])$

Observe that there was no need to dynamically focus on subarguments or to reconfigure weights. Again, this is very difficult for wfo’s and requires refined and involved techniques (of which to our knowledge no implementation in the online setting exists). For example, to admit the third sequence we have to measure something like the “termsize of the first argument of the first argument of the first argument.” For the fifth sequence this gets even more difficult.

We have intuitively demonstrated the usefulness of  $\trianglelefteq$  and that it is often more flexible than wfo’s. But can we prove some “hard” results? It turns out that we can and we now establish that — in the online setting —  $\trianglelefteq$  is strictly more generous than a large class of refined wfo’s.

**Definition 5.** *A well-founded order  $\prec$  on expressions is said to be monotonic iff the following rules hold:*

1.  $X \not\prec Y$  for all variables  $X, Y$ ,
2.  $s \not\prec f(t_1, \dots, t_n)$  whenever  $f$  is a function symbol and  $s \not\prec t_i$  for some  $i$  and
3.  $f(s_1, \dots, s_n) \not\prec f(t_1, \dots, t_n)$  whenever  $\forall i \in \{1, \dots, n\} : s_i \not\prec t_i$ .

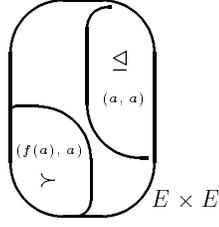
Note the similarity of structure with the definition of  $\trianglelefteq$  (but, contrary to  $\trianglelefteq$ ,  $\not\prec$  does not have to be the least relation satisfying the rules). This similarity of structure will later enable us to prove that any sequence admissible wrt  $\prec$  must also be admissible wrt  $\trianglelefteq$  (by showing that  $s \trianglelefteq t \Rightarrow s \not\prec t$ ; see Figure 2). Also observe that point 2 need not hold for predicate symbols and that point 3 implies that  $c \not\prec c$  for all constant and proposition symbols  $c$ . Finally, there is a subtle difference between monotonic wfo’s as of Definition 5 and wfo’s which possess the replacement property (such orders are called rewrite orders in [52] and monotonic in [9]). More on that below.

Similarly, we say that a *norm*  $\|\cdot\|$  (i.e. a mapping from expressions to  $\mathbb{N}$ ) is said to be monotonic iff the associated wfo  $\prec_{\|\cdot\|}$  is monotonic ( $t_1 \prec_{\|\cdot\|} t_2$  iff  $\|t_1\| < \|t_2\|$ ).

For instance the termsize norm (see below) is trivially monotonic. More generally, any semi-linear norm of the following form is monotonic:

**Proposition 2.** *Let the norm  $\|\cdot\| : Expr \rightarrow \mathbb{N}$  be defined by:*

- $\|t\| = c_f + \sum_{i=1}^n c_{f,i} \|t_i\|$  if  $t = f(t_1, \dots, t_n)$ ,  $n \geq 0$
- $\|t\| = c_v$  otherwise (i.e.  $t$  is a variable)



**Fig. 2.** Every pair admitted by a monotonic  $\succ$  is not prohibited by  $\triangleleft$

Then  $\|\cdot\|$  is monotonic if all coefficients  $c_v, c_f, c_{f,i}$  are  $\geq 0$  and  $c_{f,i} \geq 1$  for all function symbols  $f$  of arity  $\geq 1$  (but not necessarily for all predicate symbols).

*Proof.* As  $<$  on  $\mathbb{N}$  is total we have that  $s \not\prec t$  is equivalent to  $s \leq t$ . The proof proceeds by induction on the structure of the expressions and examines every rule of Definition 5 separately:

1.  $X \leq Y$  for all variables  $X, Y$   
this trivially holds as we use the same constant  $c_v$  for all variables.
2.  $s \leq f(t_1, \dots, t_n)$  whenever  $s \leq t_i$  for some  $i$   
This holds trivially if all coefficients are  $\geq 0$  and if  $c_{f,i} \geq 1$ . This is verified, as the rule only applies if  $f$  is a function symbol.
3.  $f(s_1, \dots, s_n) \leq f(t_1, \dots, t_n)$  whenever  $\forall i \in \{1, \dots, n\} : s_i \leq t_i$   
This holds trivially, independently of whether  $f$  is a function or predicate symbol, as all coefficients are positive (and the same coefficient is applied to  $s_i$  and  $t_i$ ).

By taking  $c_v = 0$  and  $c_{f,i} = c_f = 1$  for all  $f$  we get the *termsize* norm  $\|\cdot\|_{ts}$ , which by the above proposition is thus monotonic. Also, by taking  $c_v = 1$  and  $c_{f,i} = c_f = 1$  for all  $f$  we also get a monotonic norm, counting symbols. We denote this norm by  $\|\cdot\|_{sym}$ . Finally, a *linear norm* can always be obtained [55] by setting  $c_v = 0$ ,  $c_{f,i} = 1$  and  $c_f \in \mathbb{N}$  for all  $f$ . Thus, as another corollary of the above, any linear norm is monotonic.

**Proposition 3.** Let  $\|\cdot\|_1, \dots, \|\cdot\|_k$  be monotonic norms satisfying Proposition 2. Then the lexicographical ordering  $\prec_{lex}$  defined by  $s \prec_{lex} t$  iff  $\exists i \in \{1, \dots, k\}$  such that  $\|s\|_i < \|t\|_i$  and  $\forall j \in \{1, \dots, i-1\} : \|s\|_j = \|t\|_j$  is a monotonic wfo.

*Proof.* By standard results (see, e.g., [10]) we know that  $\prec_{lex}$  is a wfo (as  $<$  is a wfo on  $\mathbb{N}$ ). We will prove that  $\prec_{lex}$  satisfies all the rules of Definition 5.

1. First, rule 1 is easy as  $\|X\|_i = \|Y\|_i$  for all  $i$  and variables  $X, Y$  and therefore we never have  $X \prec_{lex} Y$ .
2. Before examining the other rules, let us note that  $s \not\prec_{lex} t$  is equivalent to saying that either
  - a)  $\forall j \in \{1, \dots, k\} \ \|s\|_j = \|t\|_j$  or
  - b) there exists an  $j \in \{1, \dots, k\}$  such that  $\|s\|_j < \|t\|_j$  and  $\forall l \in \{1, \dots, j-1\} : \|s\|_l = \|t\|_l$ .

Let us now examine rule 2 of Definition 5. We have to prove that whenever  $s \not\prec_{lex} t_i$  the conclusion of the rule holds.

Let us first examine case a) for  $s \not\prec_{lex} t_i$ . We have  $\|s\|_j = \|t_i\|_j$  and thus we know that  $\|s\|_j \leq \|f(t_1, \dots, t_n)\|_j$  by monotonicity of  $\|\cdot\|_j$  (as  $<$  on  $\mathbb{N}$  is total we have that  $s \not\prec t$  is equivalent to  $s \leq t$ ). As this holds for all  $\|\cdot\|_j$  we cannot have  $s_j \succ_{lex} f(t_1, \dots, t_n)$ .

Let us now examine the second case b) for  $s \not\prec_{lex} t_i$ . Let  $j \in \{1, \dots, k\}$  such that  $\|s\|_j < \|t_i\|_j$  and  $\forall l \in \{1, \dots, j-1\}: \|s\|_l = \|t_i\|_l$ . For all  $l$  we can deduce as above that  $\|s\|_l \leq \|f(t_1, \dots, t_n)\|_l$ . However, we still have to prove that  $\|s\|_j < \|f(t_1, \dots, t_n)\|_j$ . By monotonicity of  $\|\cdot\|_j$  we only know that  $\|s\|_j \leq \|f(t_1, \dots, t_n)\|_j$ . But we can also apply monotonicity of  $\|\cdot\|_j$  to deduce that  $\|t_i\|_j \leq \|f(t_1, \dots, t_n)\|_j$  and hence we can infer the desired property (as  $\|s\|_j < \|t_i\|_j$ ).

**3.** Now, for rule 3 we have to prove that whenever  $s_i \not\prec_{lex} t_i$  for all  $i \in \{1, \dots, n\}$  the conclusion of the rule holds. There are again two cases.

a) We can have  $\|s_i\|_j = \|t_i\|_j$  for all  $i, j$ . By monotonicity of each  $\|\cdot\|_j$  we know that  $\|f(s_1, \dots, s_n)\|_j \leq \|f(t_1, \dots, t_n)\|_j$  for all  $j \in \{1, \dots, k\}$ . Hence, we cannot have  $f(s_1, \dots, s_n) \succ_{lex} f(t_1, \dots, t_n)$ .

b) In the other case we know that there must be a value  $j' \in \{1, \dots, k\}$  such that for some  $i$ :  $\|s_i\|_{j'} < \|t_i\|_{j'}$  and  $\forall l \in \{1, \dots, j'-1\}: \|s_i\|_l = \|t_i\|_l$ . I.e., by letting  $j$  denote the minimum value  $j'$  for which this holds, we know that for some  $i$ :  $\|s_i\|_j < \|t_i\|_j$  and for all  $i'$ :  $\forall l \in \{1, \dots, j\}: \|s_{i'}\|_l \leq \|t_{i'}\|_l$ . By monotonicity of each  $\|\cdot\|_l$  we can therefore deduce that  $\forall l \in \{1, \dots, j\}: \|f(s_1, \dots, s_n)\|_l \leq \|f(t_1, \dots, t_n)\|_l$ . We can also deduce by monotonicity of  $\|\cdot\|_j$  that  $\|f(s_1, \dots, s_n)\|_j \leq \|f(t_1, \dots, t_n)\|_j$ . We can even deduce that  $\|f(s_1, \dots, s_n)\|_j \leq \|f(t_1, \dots, t_{i-1}, s_i, t_{i+1}, \dots, t_n)\|_j \leq \|f(t_1, \dots, t_n)\|_j$ . Now, we just have to prove that:

$\|f(t_1, \dots, t_{i-1}, s_i, t_{i+1}, \dots, t_n)\|_j < \|f(t_1, \dots, t_n)\|_j$  in order to affirm that

$\|f(s_1, \dots, s_n)\|_j \not\prec_{lex} \|f(t_1, \dots, t_n)\|_j$ . This does not hold for all monotonic norms, but as we know that  $\|\cdot\|_j$  satisfies Proposition 2, this can be deduced by the fact that the coefficient  $c_{f,i}$  in  $\|\cdot\|_j$  must be  $\geq 1$ .

It is important that the norms  $\|\cdot\|_1, \dots, \|\cdot\|_k$  satisfy Proposition 2. Otherwise, a counterexample would be as follows. Let  $\|a\|_1 = 1$ ,  $\|b\|_1 = 2$  and  $\|f(a)\|_1 = \|f(b)\|_1 = 5$ . Also let  $\|a\|_2 = 2$ ,  $\|b\|_2 = 1$  and  $\|f(a)\|_2 = 3$ ,  $\|f(b)\|_2 = 2$ . Now we have  $a \prec_{lex} b$ , i.e.  $a \not\prec_{lex} b$ , but also  $f(a) \succ_{lex} f(b)$  and condition 3 of monotonicity for  $\prec_{lex}$  is violated.

One could make Proposition 3 slightly more general, but the current version is sufficient to show the desired result, namely that most of the wfo's used in online practice are actually monotonic. For example almost all of the refined wfo's defined in [7, 48, 47, 44] are monotonic:

- Definitions 3.4 of [7], 3.2 of [48] and 2.14 of [47] all sum up the number of function symbols (i.e. termsize) of a subset of the argument positions of atoms. These wfo's are therefore immediately covered by Proposition 2. The algorithms only differ in the way of choosing the positions to measure. The early algorithms simply measure the input positions, while the later ones dynamically refine the argument positions to be measured (but which are still measured using the termsize norm).
- Definitions 3.2 of [47] as well as 8.2.2 of [44] use the lexicographical order on the termsizes of some selected argument positions. These wfo's are therefore monotonic as a corollary to Propositions 2 and 3.

The only non-monotonic wfo in that collection of articles is the one defined specifically for metainterpreters in Definition 3.4 of [7] (also in Section 8.6 of [44]) which uses selector functions to focus on subterms to be measured. We will return to this approach below.

Also, as already mentioned, some of the techniques in [47, 44] (in sections 3.4 and 8.2.4 respectively) do not require the *whole* sequence to be admissible wrt a unique wfo, i.e. one can split up a sequence into a (finite) number of subsequences and apply different (monotonic) wfo's on these subsequences. Similar refinements can also be developed for wqo's and the formal study of these refinements are (thus) not the main focus of the paper (see, however, the discussion in Section 4).

Before showing that  $\trianglelefteq$  is strictly more powerful than the union of all monotonic wfo's, we adapt the class of simplification orderings from term rewriting systems. It will turn out that the power of this class is also subsumed by  $\trianglelefteq$ .

**Definition 6.** *A simplification ordering is a wfo  $\prec$  on expressions which satisfies*

1.  $s \prec t \Rightarrow f(t_1, \dots, s, \dots, t_n) \prec f(t_1, \dots, t, \dots, t_n)$  (replacement property),
2.  $t \prec f(t_1, \dots, t, \dots, t_n)$  (subterm property) and
3.  $s \prec t \Rightarrow s\sigma \prec t\gamma$  for all variable only substitutions  $\sigma$  and  $\gamma$  (invariance under variable replacement).

The third rule of the above definition is new wrt term-rewriting systems and implies that all variables must be treated like a unique new constant. It turns out that a lot powerful wfo's are simplification orderings [9, 52]: recursive path ordering, Knuth-Bendix ordering or lexicographic path ordering, to name just a few. However, not all wfo's of Proposition 2 are simplification orderings: e.g., for  $c_f = 0, c_a = 1$  we have  $\|a\| = \|f(a)\|$  and the subterm property does not hold (for the associated wfo). In addition, Proposition 2 allows a special treatment for predicates. On the other hand, there are wfo's which are simplification orderings but are not monotonic according to Definition 5.

**Proposition 4.** *Let  $\prec$  be a wfo on expressions. Then any admissible sequence wrt  $\prec$  is also an admissible sequence wrt  $\trianglelefteq$  if  $\prec$  is **a)** monotonic or if it is **b)** a simplification ordering.*

*Proof.* First, let us observe that for a given wfo  $\prec$  on expressions, any admissible sequence wrt  $\prec$  is also an admissible sequence wrt  $\trianglelefteq$  iff  $s \succ t \Rightarrow s \not\trianglelefteq t$ . Indeed ( $\Rightarrow$ ), whenever  $s \trianglelefteq t$  then  $s \not\succ t$  (i.e. the picture in Figure 2 is applicable), and this trivially implies (by transitivity of  $\prec$ ) that any sequence not admissible wrt  $\trianglelefteq$  cannot be strictly descending wrt  $\prec$ . On the other hand ( $\Leftarrow$ ), let us assume that for some  $s$  and  $t$   $s \trianglelefteq t$  but  $s \succ t$ . This means that the sequence  $s, t$  is admissible wrt  $\succ$  but not wrt  $\trianglelefteq$  and we have a contradiction.

**a)** The proof that for a monotonic wfo  $\prec$  we have  $s \trianglelefteq t \Rightarrow s \not\succ t$  (and the picture in Figure 2 is applicable) is by straightforward induction on the structure of  $s$  and  $t$ . The only "tricky" aspect is that the second rule for monotonicity only holds if  $f$  is a function symbol. But if  $f$  is a predicate symbol, then  $s \trianglelefteq t$  cannot hold because we supposed that predicate and function symbols are distinct.

b) If  $\prec$  is a simplification ordering then we can apply Lemma 3.3 of [52] to deduce that  $\prec$  is the superset of the strict part of  $\trianglelefteq$  (i.e.,  $\prec \supseteq \trianglelefteq$ ). Let us examine the two possibilities for  $s \trianglelefteq t$ . First, we can have  $s \triangleleft t$ . In that case we can deduce  $s \prec t$  and thus  $s \not\prec t$ . Second, we can have  $s \trianglelefteq t$  and  $t \trianglelefteq s$ . In that case  $s$  and  $t$  are identical, except for the variables. If we now take the substitution  $\sigma$  which assigns all variables in  $s$  and  $t$  to a unique variable we have  $s\sigma = t\sigma$ , i.e.,  $s\sigma \not\prec t\sigma$ . This means that  $s \succ t$  cannot hold (because  $\succ$  is invariant under variable replacement).

Observe that transitivity of  $\prec$  is required in the proof and Proposition 4 does not simply hold for well-founded relations.

This means that the admissible sequences wrt  $\trianglelefteq$  are a superset of the union of all admissible sequences wrt simplification orderings and monotonic wfo's. In other words, no matter how much refinement we put into an approach based upon monotonic wfo's or upon simplification orderings we can only expect to approach  $\trianglelefteq$  in the limit. But by a simple example we can even dispel that hope.

*Example 3.* Take the sequence  $\delta = f(a), f(b), b, a$ . This sequence is admissible wrt  $\trianglelefteq$  as  $f(a) \trianglelefteq f(b)$ ,  $f(a) \trianglelefteq b$ ,  $f(a) \trianglelefteq a$ ,  $f(b) \trianglelefteq b$ ,  $f(b) \trianglelefteq a$  and  $a \trianglelefteq b$ . However, there is no monotonic wfo  $\prec$  which admits this sequence. More precisely, to admit  $\delta$  we must have  $f(a) \succ f(b)$  as well as  $b \succ a$ , i.e.  $a \not\prec b$ . But this violates rule 3 of Definition 5 and  $\prec$  cannot be monotonic. This also violates rule 1 of Definition 6 and  $\prec$  cannot be a simplification ordering.

These new results relating  $\trianglelefteq$  to monotonic wfo's shed light on  $\trianglelefteq$ 's usefulness in the context of ensuring online termination.

But of course the admissible sequences wrt  $\trianglelefteq$  are *not* a superset of the union of all admissible sequences wrt *any* wfo.<sup>1</sup> For instance the list-length norm  $\|\cdot\|_{len}$  is not monotonic, and indeed we have for  $t_1 = [1, 2, 3]$  and  $t_2 = [[1, 2, 3], 4]$  that  $\|t_1\|_{len} = 3 > \|t_2\|_{len} = 2$  although  $t_1 \trianglelefteq t_2$ . So there are sequences admissible wrt list-length but not wrt  $\trianglelefteq$ . The reason is that  $\|\cdot\|_{len}$  in particular and non-monotonic wfo's in general can completely ignore certain parts of the term, while  $\trianglelefteq$  will always inspect that part. E.g., if we have  $s \succ f(\dots t \dots)$  and  $\succ$  ignores the subterm  $t$  then it will also be true that  $s \succ f(\dots s \dots)$  while  $s \trianglelefteq f(\dots s \dots)$ ,<sup>2</sup> i.e. the sequence  $s, f(\dots s \dots)$  is admissible wrt  $\succ$  but not wrt  $\trianglelefteq$ .

For that same reason the wfo's for metainterpreters defined in Definition 3.4 of [7] mentioned above are not monotonic, as they are allowed to completely focus on subterms, fully ignoring other subterms. However, automation of that technique is not addressed in [7]. E.g., for this wfo one cannot immediately apply the idea of continually refining the measured subterms, because otherwise one might simply plunge deeper and deeper into the terms and termination would not be ensured. A step towards an automatic implementation is presented in Section 8.6 of [44] and it will require further work to formally compare it with wqo-based approaches and whether the ability to completely ignore certain parts

<sup>1</sup> Otherwise  $\trianglelefteq$  could not be a wqo, as *all* finite sequences without repetitions are admissible wrt some wfo (map last element to 1, second last element to 2, ...).

<sup>2</sup> Observe that if  $f$  is a predicate symbols then  $f(\dots s \dots)$  is not a valid expression, which enabled us to ignore arguments to predicates in e.g. Proposition 2.

of an expression can be beneficial for practical programs. But, as we have seen earlier,  $\trianglelefteq$  alone is already very flexible for metainterpreters, even more so when combined with characteristic trees [39] (see also [65]). We will return to the issue of metaprogramming in Section 7.

Of course, for any wfo (monotonic or not) one can devise a wbr (cf. Lemma 1) which has the same admissible sequences. Still there are some feats that are easily attained, even by using  $\trianglelefteq$ , but which *cannot* be achieved by a wfo approach (monotonic or not). Take the sequences  $S_1 = p([], [a]), p([a], [])$  and  $S_2 = p([a], []), p([], [a])$ . Both of these sequences are admissible wrt  $\trianglelefteq$ . This illustrates the flexibility of using well-quasi orders compared to well-founded ones in an online setting, as there exists *no* wfo (monotonic or not) which will admit *both* these sequences.<sup>3</sup> It however also illustrates why, when using a wqo in that way, one has to compare with every predecessor state of a process. Otherwise one can get infinite derivations of the form  $p([a], []) \rightarrow p([], [a]) \rightarrow p([a], []) \rightarrow \dots$ <sup>4</sup>

**Short Note on Offline Termination** This example also shows why  $\trianglelefteq$  (or well-quasi orders in general) cannot be used *directly* for static termination analysis. Let us explain what we mean. Take, e.g., a program containing the clauses  $C_1 = p([a], []) \leftarrow p([], [a])$  and  $C_2 = p([], [a]) \leftarrow p([a], [])$ . Then, in both cases the body is not embedding the head, but still the combination of the two clauses leads to a non-terminating program. However,  $\trianglelefteq$  can be used to *construct well-founded* orders for static termination analysis. Take the clause  $C_1$ . The head and the body are incomparable according to  $\trianglelefteq$ . So, we can simply extend  $\trianglelefteq$  by stating that  $p([a], []) \triangleright p([], a)$  (thus making the head strictly larger than the body atom). As already mentioned, for any extension  $\leq$  of a wqo we have that  $<$  is a wfo. Thus we know that the program just consisting of  $C_1$  is terminating. If we now analyse  $C_2$  we have that, according to the extended wqo, the body is strictly larger than the head and (luckily) we cannot prove termination (i.e. there is no way of extending  $\trianglelefteq$  so that for both  $C_1$  and  $C_2$  the head is strictly larger than the body).

## 4 Nearly-Foundedness and Over-Eagerness

In [47], as well as in Section 8.2.4 of [44], a technique for wfo's is formally introduced, based upon *nearly-foundedness*. As already mentioned earlier, some of the techniques in [7, 48, 47, 44] start out with a very coarse wfo  $<_1$  which is

<sup>3</sup> To allow to go from  $p([], [a])$  to  $p([a], [])$  via a wfo  $<$  we need to have that  $p([], [a]) > p([a], [])$  (cf. Example 1). Now, if in another context we want to go from  $p([a], [])$  to  $p([], [a])$  this is clearly impossible (using  $<$ ) because we cannot have  $p([a], []) > p([], [a])$ .

<sup>4</sup> When using a wfo one has to compare only to the closest predecessor [47], because of the transitivity of the order and the strict decrease enforced at each step. However, wfo's are usually extended to incorporate variant checking and then require inspecting every predecessor anyway (though only when there is no strict weight decrease, see, e.g., [44, 47]).

then continuously refined, enabling a clever choice of weights for predicates and their respective arguments (deciding beforehand upon appropriate weights can be extremely difficult or impossible; see examples in Section 3.2). For example we might have that  $<_1$  is based upon measuring the sum of the termsize of all arguments. The process of refining might then consist in dropping one or more arguments. For example, suppose that we have some sequence  $s_1, s_2, \dots, s_i$  which is admissible wrt the initial wfo  $<_1$  but where  $s_{i+1} \not\prec_1 s_i$  with  $s_{i+1} = p(a, s(s(b)))$  and  $s_i = p(s(a), b)$ . In that case we can move to a refinement  $<_2$  of  $<_1$  in which only the termsize of the first argument is measured, making  $s_{i+1} <_2 s_i$  and enabling the move from  $s_i$  to  $s_{i+1}$ . This, however, does not guarantee that the *whole* sequence  $s_1, s_2, \dots, s_i, s_{i+1}$  is admissible wrt  $<_2$ . Indeed, we might for instance have the sequence  $p(s(a), s(s(s(c))))$ ,  $p(s(a), b)$ ,  $p(a, s(s(b)))$  with  $i = 2$  and  $s_2 \not\prec_1 s_1$  even though  $s_2 <_1 s_1$ .

To solve this problem, the earlier algorithms verified that a refinement keeps the whole sequence admissible (otherwise it was disallowed). The problem with this approach is that re-checking the entire sequence can be expensive. [47, 44] therefore advocates another solution: not re-checking the entire sequence on the grounds that it does not threaten termination (provided that the refinements themselves are well-founded). This leads to sequences  $s_1, s_2, \dots$  which are not well-founded but *nearly-founded* [47, 44] meaning that  $s_i \not\prec s_j$  only for a finite number of pairs  $(i, j)$  with  $i > j$ .

In summary, the motivation for nearly-foundedness lies in speeding up the construction of admissible sequences (not re-scanning the initial sequence upon refinement). As a side-effect, this approach will admit more sequences and, e.g., solve some of our earlier examples (as a suitably large depth bound would as well). However, from a theoretical point of view, we argue below that nearly-foundedness is difficult to justify and somewhat unsatisfactory.

First, we call a technique *over-eager* if it admits sequences which are not admitted by the variant test (i.e., it admits sequences containing variants). We call such a technique *strictly over-eager* if it admits sequences which contain more than 1 occurrence of the same syntactic term.

A depth bound based technique is strictly over-eager, which is obviously a very undesirable property indicating some ad-hoc behaviour. The same can (almost always)<sup>5</sup> also be said for over-eagerness. For instance, in the context of partial deduction or unfold/fold program transformation, over-eager unfolding will “hide” possibilities for perfect folding (namely the variants) and also lead to too large specialised programs. An approach based upon homeomorphic embedding is not over-eager nor is any other wfo/wqo based approach which does not distinguish between variants. However, as we show below, using nearly-foundedness leads to *strict* over-eagerness.

Let us first formalise the way nearly-founded sequences are constructed in [47, 44]. First, we define a well-founded order  $\prec_{\mathcal{W}}$  acting on a set  $\mathcal{W}$  of well-founded orders on expressions. To construct admissible sequences of expressions wrt  $\prec_{\mathcal{W}}$  we start by using one wfo  $<_1 \in \mathcal{W}$  until the sequence can no longer be

---

<sup>5</sup> See discussion in Section 5 concerning the variant test *on covering ancestors*.

extended. Once this is the case we can use another wfo  $\prec_2 \in \mathcal{W}$  which admits the offending step, provided that  $w_2 \prec_{\mathcal{W}} w_1$ . It is *not* required that the whole initial sequence is admissible wrt  $\prec_1$ , just the last step (not admitted by  $\prec_0$ ). We can now continue expanding the sequence until we again reach an offending step, where we can then try to refine  $\prec_1$  into some  $\prec_2$  with  $\prec_2 \prec_{\mathcal{W}} \prec_1$ , and so on, until no further expansion is possible.

*Example 4.* Take the following program.

$$\begin{aligned} p([a, a|T], [a|Y]) &\leftarrow p(T, Y) \\ p([b, b, b|T], Y) &\leftarrow p(T, [b, b|Y]) \\ p(T, [b, b|Y]) &\leftarrow p([a, a, b, b, b|T], [a|Y]) \end{aligned}$$

Let us define the following well-founded orders:

$$\begin{aligned} \prec_{1+2}: p(s_1, s_2) &< p(t_1, t_2) \text{ iff } \|s_1\|_{ts} + \|s_2\|_{ts} < \|t_1\|_{ts} + \|t_2\|_{ts} \\ \prec_1: p(s_1, s_2) &< p(t_1, t_2) \text{ iff } \|s_1\|_{ts} < \|t_1\|_{ts} \\ \prec_2: p(s_1, s_2) &< p(t_1, t_2) \text{ iff } \|s_2\|_{ts} < \|t_2\|_{ts} \end{aligned}$$

We also define the wfo  $\prec_{\mathcal{W}}$  on the above well-founded orders:  $\prec_1 \prec_{\mathcal{W}} \prec_{1+2}$  and  $\prec_2 \prec_{\mathcal{W}} \prec_{1+2}$ . In other words we can refine  $\prec_{1+2}$  into  $\prec_1$  or  $\prec_2$ , which in turn cannot be further refined.

We can now construct the following admissible sequence in which two terms ( $p([a, a, b, b, b], [a])$  and  $p([b, b, b], [])$ ) appear twice:

$$\begin{aligned} &p([a, a, b, b, b], [a]) \\ &\quad \downarrow \prec_{1+2} \\ &p([b, b, b], []) \\ &\quad \downarrow \prec_{1+2} \\ &p([], [b, b]) \\ &\quad \downarrow \prec_2 \text{ (refinement)} \\ &\underline{p([a, a, b, b, b], [a])} \\ &\quad \downarrow \prec_2 \\ &\underline{p([b, b, b], [])} \end{aligned}$$

Example 4 thus proves that nearly-foundedness results in strict over-eagerness. (As a side-effect, the example also shows that the nearly-foundedness approach cannot be mapped to a wfo-approach, however involved it might be.) Although it is unclear how often such situations will actually arise in practice, we believe that the strict over-eagerness is just one of the (mathematically) unsatisfactory aspects of nearly-foundedness.

## 5 A more refined treatment of variables

While  $\preceq$  has a lot of desirable properties it still suffers from some drawbacks. Indeed, as can be observed in Example 2, the homeomorphic embedding relation  $\preceq$  as defined in Definition 4 is rather crude wrt variables. In fact, all variables are treated as if they were the same variable, a practice which is clearly undesirable in a logic programming context. Intuitively, in the above example,  $p(X, Y) \preceq p(X, X)$  can be justified (see, however, Definition 8 below), while

$p(X, X) \sqsubseteq p(X, Y)$  is not. Indeed  $p(X, X)$  can be seen as standing for something like  $and(eq(X, Y), p(X, Y))$ , which embeds  $p(X, Y)$ , but the reverse does not hold.

Secondly,  $\sqsubseteq$  behaves in quite unexpected ways in the context of generalisation, posing some subtle problems wrt the termination of a generalisation process.

*Example 5.* Take for instance the following generalisation algorithm, which appears (in disguise) in a lot of partial deduction algorithms (e.g., [38, 33, 39]). (In that context  $\mathcal{A}$  is the set of atoms for which SLDNF-trees have already been constructed while  $\mathcal{B}$  are the atoms in the leaves of these trees. The goal of the algorithm is then to extend  $\mathcal{A}$  such that all leaf atoms are covered.)

**Input:** two finite sets  $\mathcal{A}, \mathcal{B}$  of atoms  
**Output:** a finite set  $\mathcal{A}' \supseteq \mathcal{A}$  s.t. every atom in  $\mathcal{B}$  is an instance of an atom in  $\mathcal{A}'$   
**Initialisation:**  $\mathcal{A}' := \mathcal{A}, \mathcal{B}' := \mathcal{B}$   
**while**  $\mathcal{B}' \neq \emptyset$  **do**  
    **remove** an element  $B$  from  $\mathcal{B}'$   
    **if**  $B$  is not an instance of an element in  $\mathcal{A}'$  **then**  
        **if**  $\exists A \in \mathcal{A}'$  such that  $A \sqsubseteq B$  **then**  
            **add**  $msg(A, B)$  to  $\mathcal{B}'$   
        **else add**  $B$  to  $\mathcal{A}'$

The basic idea of the algorithm is to use  $\sqsubseteq$  to keep the set  $\mathcal{A}'$  finite in the limit. However, although the above algorithm will indeed keep  $\mathcal{A}'$  finite, it still does not terminate. Take for example  $\mathcal{A} = \{p(X, X)\}$  and  $\mathcal{B} = \{p(X, Y)\}$ . We will remove  $B = p(X, Y)$  from  $\mathcal{B}' = \{p(X, Y)\}$  in the first iteration of the algorithm and we have that  $B$  is not an instance of  $p(X, X)$  and also that  $p(X, X) \sqsubseteq p(X, Y)$ . We therefore calculate the  $msg(\{p(X, X), p(X, Y)\}) = p(X, Y)$  and we have a loop (we get  $\mathcal{B}' = \{p(X, Y)\}$ ).

To remedy these problems, [38, 33, 39] introduced the so called strict homeomorphic embedding as follows:

**Definition 7.** ( $\sqsubseteq^+$ ) *Let  $A, B$  be expressions. Then  $B$  (strictly homeomorphically) embeds  $A$ , written as  $A \sqsubseteq^+ B$ , iff  $A \sqsubseteq B$  and  $A$  is not a strict instance of  $B$ .*

*Example 6.* We now still have that  $p(X, Y) \sqsubseteq^+ p(X, X)$  but not  $p(X, X) \sqsubseteq^+ p(X, Y)$ . Note that still  $X \sqsubseteq^+ Y$  and  $X \sqsubseteq^+ X$ .

A small experiment, specialising a query `rotate(X, X)` (using the ECCE system [32] with  $\sqsubseteq$  and  $\sqsubseteq^+$  respectively on conjunctions for global control; the `rotate` program, rotating a binary tree, can be found in [32]) demonstrates the interest of  $\sqsubseteq$ : when using  $\sqsubseteq^+$  we obtain an overall speedup of 2.5 compared to “only” 2.0 using  $\sqsubseteq$ .

Notice that, if we replace  $\sqsubseteq$  of Example 5 by  $\sqsubseteq^+$  we no longer have a problem with termination (see [39, 33] for a termination proof of an Algorithm containing the one of Example 5).

The following is proven in [33, 39].

**Theorem 1.** *The relation  $\sqsubseteq^+$  is a wbr on the set of expressions over a finite alphabet.*

Observe that  $\sqsubseteq^+$  is not a wqo as it is not transitive:<sup>6</sup> we have for example  $p(X, X, Y, Y) \sqsubseteq^+ p(X, Z, Z, X)$  and  $p(X, Z, Z, X) \sqsubseteq^+ p(X, X, Y, Z)$  but  $p(X, X, Y, Y) \not\sqsubseteq^+ p(X, X, Y, Z)$ . One might still feel dissatisfied with that definition for another reason. Indeed, although going from  $p(X)$  to  $p(f(X))$  (and on to  $p(f(f(X)))$ , ...) looks very dangerous, a transition from  $p(X, Y)$  to  $p(X, X)$  is often not dangerous, especially in a database setting. Take for example a simple Datalog program just consisting of the clause  $p(a, b) \leftarrow p(X, X)$ . Obviously  $P$  will terminate (i.e., fail finitely) for all queries but  $\sqsubseteq^+$  will not allow full unfolding of  $\leftarrow p(X, Y)$  and finite failure will not be detected by a specialiser relying (solely) on  $\sqsubseteq^+$ . On the practical side this means that neither  $\sqsubseteq^+$  nor  $\sqsubseteq$  will allow full unfolding of all terminating queries to Datalog programs (although they will allow full unfolding of terminating ground queries to range-restricted Datalog programs). To remedy this, we can develop the following refinement of  $\sqsubseteq^+$ .

**Definition 8.** *We define  $s \sqsubseteq_{var} t$  iff  $s \sqsubset t$  or  $s$  is a variant of  $t$ .*

We have  $p(X) \sqsubseteq_{var} p(f(X))$  and  $p(X, Y) \sqsubseteq_{var} p(Z, X)$  but  $p(X, X) \not\sqsubseteq_{var} p(X, Y)$  and  $p(X, Y) \not\sqsubseteq_{var} p(X, X)$ .

It is obvious that  $\sqsubseteq_{var}$  is strictly more powerful than  $\sqsubseteq^+$  (if  $t$  is strictly more general than  $s$ , then it is not a variant of  $s$  and it is also not possible to have  $s \sqsubset t$ ). It thus also solves the generalisation problems of  $\sqsubseteq$ . In addition  $\sqsubseteq_{var}$  has the following property: if we have a query  $\leftarrow Q$  to a Datalog program which left-terminates then the LD-tree for  $\leftarrow Q$  is admissible in the sense that, for every selected literal  $L$ , we have  $L \not\sqsubseteq_{var} A$  for all covering ancestors  $A$  of  $L$ . Indeed, whenever we have that  $L \sqsubseteq A$  for two Datalog atoms  $L$  and  $A$  then we must also have  $A \sqsubseteq L$  (as the diving rule of  $\sqsubseteq$  cannot be applied). Thus,  $\sqsubseteq_{var}$  is equivalent to the variant test. Now, if a derivation from  $\leftarrow A$  leads to a goal  $\leftarrow L_1, \dots, L_n$  where all  $L_i$  is a variant of a covering ancestor  $A$ , then it is possible to repeat this derivation again and again and we have a real loop.

**Theorem 2.** *The relation  $\sqsubseteq_{var}$  is a wqo on the set of expression over a finite alphabet.*

The proof can be found in Appendix A. Observe that  $\sqsubseteq_{var}$  is, like  $\sqsubseteq$  and  $\sqsubseteq^+$ , not a wqo over an infinite alphabet. More on that in Section 6.

## Discussion

Observe that the variant test is (surprisingly) not complete for Datalog in general (under arbitrary computation rules). Take the program just consisting of  $p(b, a) \leftarrow p(X, Z), p(Z, Y)$ . Then the query  $\leftarrow p(X, Y)$  is finitely failed as the

<sup>6</sup> Thanks to anonymous referees for pointing this out.

following derivation shows:

$$\underline{p(X, Y)} \rightsquigarrow \underline{p(X', Z')}, p(Z', Y') \rightsquigarrow p(X'', Z''), p(Z'', Y''), \underline{p(a, Y')} \rightsquigarrow \text{fail}.$$

However, at the second step (no matter what we do) we have to select a variant of the covering ancestor  $p(X, Y)$  and the variant test will prevent full unfolding.

An alternate approach to Definitions 7 and 8 — at least for the aspect of treating variables in a more refined way — might be based on numbering variables using some mapping  $\#(\cdot)$  and then stipulating that  $X \sqsubseteq^\# Y$  iff  $\#(X) \leq \#(Y)$ . For instance in [43] a de Bruijn numbering of the variables is proposed. Such an approach, however, has a somewhat ad hoc flavour to it. Take for instance the terms  $p(X, Y, X)$  and  $p(X, Y, Y)$ . Neither term is an instance of the other and we thus have  $p(X, Y, X) \sqsubseteq^+ p(X, Y, Y)$  and  $p(X, Y, Y) \sqsubseteq^+ p(X, Y, X)$ . Depending on the particular numbering we will either have that  $p(X, Y, X) \not\sqsubseteq^\# p(X, Y, Y)$  or that  $p(X, Y, Y) \not\sqsubseteq^\# p(X, Y, X)$ , while there is no apparent reason why one expression should be considered smaller than the other.<sup>7</sup>

While  $\sqsubseteq_{var}$  may be very interesting for local control compared to  $\sqsubseteq$ , e.g., due to its improved unfolding capabilities for Datalog programs,  $\sqsubseteq^+$  is much less interesting for *local control* of partial deduction. Indeed, when we go from  $p(X, X)$  to  $p(X, Y)$  then  $\sqsubseteq^+$  will allow further unfolding. But, if we are able to unfold  $p(X, X)$  to  $p(X, Y)$  then we will usually be able to go again from  $p(X, Y)$  to  $p(X', Y')$  and there is a loop anyhow. However, for *global control* it will often be interesting to specialise  $p(X, X)$  separately from  $p(X, Y)$ , which will be allowed by  $\sqsubseteq^+$  and  $\sqsubseteq_{var}$  but not by  $\sqsubseteq$  (see the `rotate` experiment above).

## 6 Extended homeomorphic embedding

Although  $\sqsubseteq^+$  from Definition 7 has a more refined treatment of variables and has a much better behaviour wrt generalisation than  $\sqsubseteq$  of Definition, it is still somewhat unsatisfactory.

One point is the restriction to a finite alphabet. Indeed, for a lot of practical logic programs, using, e.g., arithmetic built-ins or even  $= .. / 2$ , a finite alphabet is no longer sufficient. Luckily, the fully general definition of homeomorphic embedding as in [28, 10] remedies this aspect. It even allows functors with variable arity.<sup>8</sup> We will show below how this definition can be adapted to a logic programming context.

However, there is another unsatisfactory aspect of  $\sqsubseteq^+$  (and  $\sqsubseteq_{var}$ ). Indeed, it will ensure that  $p(X, X) \not\sqsubseteq^+ p(X, Y)$  while  $p(X, X) \sqsubseteq p(X, Y)$  but we still have that, e.g.,  $f(a, p(X, X)) \sqsubseteq^+ f(f(a), p(X, Y))$ . In other words, the more refined treatment of variables is only performed at the top, but not recursively within the structure of the expressions. For instance, this means that  $\sqsubseteq^+$  will handle `rotate(X, X)` much better than  $\sqsubseteq$  but this improvement will often go away when we add a layer of metainterpretation.

<sup>7</sup> [43] also proposes to consider all possible numberings (but leading to  $n!$  complexity, where  $n$  is the number of variables in the terms to be compared). It is unclear how such a relation compares to  $\sqsubseteq^+$  and  $\sqsubseteq_{var}$ .

<sup>8</sup> Which can also be seen as associative operators.

The following, new and more refined embedding relation remedies this somewhat ad hoc aspect of  $\leq^+$ .

**Definition 9.** ( $\leq^*$ ) *Given a wbr  $\preceq_{fun}$  on the functors and a wbr  $\preceq_{exp}$  on sequences of expressions, we define the extended homeomorphic embedding on expressions by the following rules:*

1.  $X \leq^* Y$  if  $X$  and  $Y$  are variables
2.  $s \leq^* f(t_1, \dots, t_n)$  if  $s \leq^* t_i$  for some  $i$
3.  $f(s_1, \dots, s_m) \leq^* g(t_1, \dots, t_n)$  if  $f \preceq_{fun} g$  and  $\exists 1 \leq i_1 < \dots < i_m \leq n$  such that  $\forall j \in \{1, \dots, m\} : s_j \leq^* t_{i_j}$  and  $\langle s_1, \dots, s_m \rangle \preceq_{exp} \langle t_1, \dots, t_n \rangle$

The above definition requires a wbr  $\preceq_{fun}$  on the functors. If we have a finite alphabet, then equality is a wqo (one can thus obtain the pure homeomorphic embedding as a special case). In the context of, e.g., partial deduction, we know that the functors occurring within the program (text) and goal to be analysed are of finite number. One might call these functors *static* and all others *dynamic*. A wqo can be obtained by defining  $f \preceq g$  if either  $f$  and  $g$  are dynamic or if  $f = g$ . For particular types of functors a natural wqo or wbr exists (e.g., for numbers) which can be used instead.

In the above definition we can instantiate  $\preceq_{exp}$  such that it performs a more refined treatment of variables, such as discussed in Section 5. For example we can define:  $\langle s_1, \dots, s_m \rangle \preceq_{exp} \langle t_1, \dots, t_n \rangle$  iff if  $\langle t_1, \dots, t_n \rangle$  is not strictly more general than  $\langle s_1, \dots, s_m \rangle$ . (Observe that this means that if  $m \neq n$  then  $\preceq_{exp}$  will hold.) This relation is a wbr (by Lemma 1 as the strictly more general relation is a wfo [24]). Then, in contrast to  $\leq^+$  and  $\leq_{var}$ , this refinement will be applied *recursively* within  $\leq^*$ . For example we now have  $f(a, p(X, X)) \not\leq^* f(f(a), p(X, Y))$ .

The reason why a recursive use of, e.g., the “not strict instance” test was not incorporated in [38, 33, 39] which use  $\leq^+$  was that the authors were not sure that this would remain a wbr (no proof was found yet). In fact, recursively applying the “not strict instance” looks very dangerous. Take, e.g., the following two atoms  $A_0 = p(X, X)$  and  $A_1 = q(p(X, Y), p(Y, X))$ . In fact, although  $A_0 \leq^+ A_1$  we do not have  $A_0 \leq^* A_1$  (when, e.g., considering both  $q$  and  $p$  as static functors) and one wonders whether it might be possible to create an infinite sequence of atoms by, e.g., producing  $A_2 = p(q(p(X, Y), p(Y, Z)), q(p(Z, V), p(V, X)))$ . We indeed have  $A_1 \not\leq^* A_2$ , but luckily  $A_0 \leq^* A_2$  and  $\leq^*$  satisfies the wqo requirement of Definition 3. But can we construct some sequence for which  $\leq^*$  does not conform to Definition 3?

The following Theorem 3 shows that such a sequence *cannot* be constructed. However, if we slightly strengthen point 3 of Definition 9 by requiring that  $\langle s_1, \dots, s_m \rangle$  is not a strict instance of the selected subsequence  $\langle t_{i_1}, \dots, t_{i_m} \rangle$ , we actually no longer have a wqo, as the following sequence of expression shows:  $A_0 = f(p(X, X))$ ,  $A_1 = f(p(X, Y), p(Y, X))$ ,  $A_2 = f(p(X, Y), p(Y, Z), p(Z, X))$ ,  $\dots$ . Using the slightly strengthened embedding relation no  $A_i$  would be embedded in any  $A_j$ , while using Definition 9 unmodified we have, e.g.,  $A_1 \leq^* A_2$  (but not  $A_0 \leq^* A_1$  or  $A_0 \leq^* A_2$ ).

**Theorem 3.**  $\triangleleft^*$  is a wbr on expressions. Additionally, if  $\preceq_{fun}$  and  $\preceq_{sexp}$  are wqo's then so is  $\triangleleft^*$ .

The proof can be found in Appendix A.

## 7 The Representation Problem for Metaprogramming

As we have seen earlier in Section 3.2,  $\triangleleft$  alone is already very flexible for metainterpreters, even more so when combined with characteristic trees [39] (see also [65]). This section will, however, show that some subtle problems remain.

Let us first discuss the issue of representing object-level programs at the meta-level. In setting with logical variables, there are basically two different approaches to representing an object level expression, say the atom  $p(X, a)$ , at the meta-level. In the first approach one uses the term  $p(X, a)$  as the object level representation. This is called a *non-ground* representation, because it represents an object level variable by a meta-level variable. In the second approach one would use something like the term  $struct(p, [var(1), struct(a, [])])$  to represent the object level atom  $p(X, a)$ . This is called a *ground* representation, as it represents an object level variable by a ground term.<sup>9</sup> Figure 3 contains some further examples of the particular ground representation which we will use in this section. From now on, we use “ $\mathcal{T}$ ” to denote the ground representation of an expression  $\mathcal{T}$ .

Object level	Ground representation
$X$	$var(1)$
$c$	$struct(c, [])$
$f(X, a)$	$struct(f, [var(1), struct(a, [])])$
$p \leftarrow q$	$struct(\text{clause}, [struct(p, []), struct(q, [])])$

**Fig. 3.** A ground representation

To abstract from the particular representation and the number of layers of metainterpretation, we define  $enc(\cdot)$  to be the encoding function used to go from a call at the object level to the meta-level.

For the non-ground vanilla metainterpreter [21] we have for example  $enc(p(X)) = solve(p(X))$ . If we have two nested vanilla like metainterpreters we might even get  $enc(p(X)) = solve(solve(p(X), Depth), Trace)$ . For the ground representation of Figure 3 we will get  $enc(p(X)) = solve(struct(p, [var(1)]), CAS)$ .

We say that a wfo or wbr is *invariant under a particular encoding*  $enc(\cdot)$  if whenever it admits a sequence of calls  $o_1, o_2, \dots, o_n$  then it also admits  $enc(o_1), enc(o_2), \dots, enc(o_n)$ . Solving the representation problem then amounts to finding

<sup>9</sup> For a more detailed discussion we refer the reader to [21], [45], [22, 6], [37, 36, 33].

an adequate wfo or wbr which is invariant under a given encoding and powerful enough at the object level (obviously the total relation  $\preceq_{\tau}$  with  $s \preceq_{\tau} t$  is a wqo which is invariant under any encoding  $enc(\cdot)$ ).

We conjecture (prove ?) that  $\trianglelefteq$  solves the representation problem for any vanilla like encoding. One might hope that the same holds for a ground representation, like the one depicted in Figure 3. Unfortunately, this is not the case due to several problems:

**Multiple arity** If the same predicate symbol can occur with multiple arity, then the following can happen. We have that  $p(X) \not\trianglelefteq p(X, X)$ , on the account that  $p/1$  is a different predicate than  $p/2$ . However, in the ground representation we have  $struct(p, [X]) \trianglelefteq struct(p, [X, X])$  because an arity of  $n$  is translated into one argument (i.e., we have a fixed arity) containing a list of length  $n$ . A simple solution to this problem is to use a predicate symbol only with a single arity.

**Variable encoding** If we represent variables as integers of the form  $\tau = 0||s(\tau)$  then we can have that  $X \trianglelefteq Y$  while  $var(s(0)) \not\trianglelefteq var(0)$ . (In that case  $\trianglelefteq$  on the encoding is actually more admissible.) One solution is to use a different functor for each distinct variable (meaning we have an infinite alphabet) and then using  $\trianglelefteq^*$  to define a wqo on these functors: either treating all variables as one or incorporating refinements similar to  $\trianglelefteq^+$  and  $\trianglelefteq_{var}$ .

**Multiple embeddings in the same argument** Unfortunately, even in the absence of variables and even if every function symbol only occurs with a single arity,  $\trianglelefteq$  is not invariant under the ground representation. For example, we have

$$f(a, b) \not\trianglelefteq f(g(a, b), c)$$

while

$$struct(f, ["a", "b"]) \trianglelefteq struct(f, [struct(g, ["a", "b"]), "c"]).$$

The reason is that the coupling rule 3. of Definition 4 of  $\trianglelefteq$  checks whether  $a$  is embedded in  $g(a, b)$  (which holds) and  $b$  is embedded in  $c$  (which does not hold). The rule *disallows* to search for *both*  $a$  and  $b$  in the *same argument*  $g(a, b)$  (which would hold). But this is exactly what *is allowed* when working with the ground representation due to the fact that an argument tuple is translated into a list. One solution to this problem is to weaken  $\trianglelefteq$  such that it allows to search the same argument multiple times for embedding:

**Definition 10.** ( $\trianglelefteq^-$ ) *We define the weak homeomorphic embedding on expressions by the following rules:*

1.  $X \trianglelefteq^- Y$  if  $X$  and  $Y$  are variables
2.  $s \trianglelefteq^- f(t_1, \dots, t_n)$  if  $s \trianglelefteq^- t_i$  for some  $i$
3.  $f(s_1, \dots, s_n) \trianglelefteq^- f(t_1, \dots, t_n)$  if  $\exists 1 \leq i_1 \leq \dots \leq i_n \leq n$  such that  $\forall j \in \{1, \dots, n\} : s_j \trianglelefteq^- t_{i_j}$ .

Note that, contrary to  $\trianglelefteq$ ,  $i_j$  can be equal to  $i_{j+1}$ . We now have  $f(a, b) \trianglelefteq^- f(g(a, b), c)$ . We also have that  $s \trianglelefteq t \Rightarrow s \trianglelefteq^- t$ .

We conjecture that  $\triangleleft^-$  is invariant under the ground representation  $enc(\cdot)$  of Figure 3 (provided that the problems concerning multiple arity and variables have been resolved).

We also conjecture that  $enc(s) \triangleleft enc(t)$  iff  $s \triangleleft^- t$ . This would have as corollary that  $enc(s) \triangleleft enc(t)$  iff  $enc(enc(s)) \triangleleft enc(enc(t))$

However, there still remains the open problem: Can we find a strengthening of  $\triangleleft$  which is invariant under  $enc(\cdot)$ . It might be possible to achieve this by using (a refinement of) [56], which extends Kruskal’s theorem. A pragmatic solution would of course be to simply de-encode data as much as possible in, e.g., the program specialiser, and then apply  $\triangleleft$  (or  $\triangleleft^*$ ) on the de-encoded data only. This, however, requires knowledge about the particular encodings that are likely to appear.

### The parsing problem

In the context of meta-programming, we also have what is called the *parsing problem* [44]. Below we provide another view on the parsing problem, in terms of invariance under representation.

In partial deduction one usually doesn’t compare entire goals but just the selected atoms. The initial motivation seems to have been to allow a more liberal unfolding. Take for example a program containing  $p(X) \leftarrow p(f(X))$  and take the goal  $\leftarrow p(a), q(b)$ . If we select  $p(a)$  we obtain the resolvent  $\leftarrow p(f(a)), q(b)$ . Now, when inspecting entire goals, one cannot help but notice a growth of structure and stop the unfolding. However, if we just examine the sequence of selected literals we will be allowed to select  $q(b)$ , which is a very sensible and safe thing to do as  $q(b)$  has not been touched yet.<sup>10</sup> As another example take the program consisting of the single clause  $p(f(X)) \leftarrow p(X), q$  and let us unfold  $\leftarrow p(Y)$  by just looking at  $\triangleleft$  on the selected atoms:  $\leftarrow p(Y) \rightsquigarrow \leftarrow p(X), q \rightsquigarrow fail$ . Again, if we look at the entire goal we will not be able to fully unfold  $\leftarrow p(Y)$ .

The problem in a meta-programming context, is that this refinement is not even invariant under the non-ground representation. Take the standard vanilla metainterpreter, together with an encoding of the above program consisting of the single clause  $p(f(X)) \leftarrow p(X), q$ :

$$\begin{array}{l} solve(true) \leftarrow \\ solve(A\&B) \leftarrow solve(A), solve(B) \end{array}$$

<sup>10</sup> However, even this refinement is not enough, and one usually only compares sequences of *covering ancestors* [7]. Take for example a program consisting of the single fact “ $p(X) \leftarrow$ ” and let us unfold the query  $\leftarrow p(a), p(f(a))$ . We start by selecting  $p(a)$  which gives us the goal  $\leftarrow p(f(a))$ . Let us now try to select  $p(f(a))$ , which is a sensible thing to do as it has not been touched yet. However, if we examine the sequence of selected literals we get  $\langle p(a), p(f(a)) \rangle$  which is not admissible wrt  $\triangleleft$  (or any of the other extensions discussed in the paper). To overcome this one should register descendency (via resolution) relationships and only check  $\triangleleft$  (or any other wfo/wqo approach) on sequences of covering ancestors. In this case,  $p(f(a))$  has no ancestor and we get the sequence  $\langle p(f(a)) \rangle$  which is of course admissible wrt any wfo or wqo.

$$\begin{aligned} & \text{solve}(H) \leftarrow \text{clause}(H, B), \text{solve}(B) \\ & \text{clause}(p(f(X)), p(X), q(X)) \leftarrow \end{aligned}$$

One would hope that, by looking at  $\sqsubseteq$  on the selected literals, it would be possible to fully unfold  $\leftarrow \text{solve}(p(Y))$  in a similar manner and detect finite failure as above. Let us examine the sequence of goals, needed to detect finite failure:  $\leftarrow \text{solve}(p(Y)) \rightsquigarrow \leftarrow \text{clause}(p(Y), B), \text{solve}(B) \rightsquigarrow \leftarrow \text{solve}(p(X)\&q) \rightsquigarrow \leftarrow \text{solve}(p(X)), \text{solve}(q) \rightsquigarrow \leftarrow \text{solve}(p(X)), \text{clause}(q, B'), \text{solve}(B') \rightsquigarrow \text{fail}$ . Unfortunately we have at the third step that  $\text{solve}(p(Y)) \sqsubseteq \text{solve}(p(X)\&q)$  (the same holds for  $\sqsubseteq^+$  or  $\sqsubseteq^*$ ) and we are unable to fully unfold  $\leftarrow \text{solve}(p(Y))$  using  $\sqsubseteq$  on selected literals (or even on entire goals). The problem is that in the metainterpreter multiple atoms can be mopped together in a single term, and the refinement of looking only at the selected atom does not scale up to the metalevel!

It is hard to see how this problem can be solved in a general manner: one would have to know that  $\text{solve}$  will eventually decompose  $p(X)\&q$  into its constituents  $p(X)$  and  $q$  giving us the opportunity to continue with  $\text{solve}(q)$  while stopping the unfolding of  $\text{solve}(p(X))$ . [65] presents a solution to this problem for the particular vanilla metainterpreter above, but it does not scale up to other, slightly more involved metainterpreters.

## 8 Discussion and Conclusion

Of course  $\sqsubseteq^*$  is not the ultimate relation for ensuring online termination. Although it has proven to be extremely useful superimposed e.g. on determinate unfolding, on its own in the context of local control of partial deduction,  $\sqsubseteq^*$  (as well as  $\sqsubseteq^+$  and  $\sqsubseteq$ ) will sometimes allow too much unfolding than desirable for efficiency concerns: more unfolding does not always imply a better specialised program. We refer to the solutions developed in, e.g., [39, 27]. Similar problems can arise in the setting of global control and we again refer to [39, 27] for discussions and experiments. Also, the issue of an efficient implementation of the homeomorphic embedding relation still remains open. (However, in Section 4 we have shown that the efficient way to use wfo's, which avoids re-scanning the entire sequence upon refinement, has very undesirable properties.)

For some applications,  $\sqsubseteq$  as well as  $\sqsubseteq^+$  and  $\sqsubseteq^*$  remain too restrictive. In particular, they do not always deal satisfactorily with fluctuating structure (arising, e.g., for certain metainterpretation tasks) [65]. The use of characteristic trees [33, 39] remedies this problem to some extent, but not totally. A further step towards a solution is presented in [65]. In that light, it might be of interest to study whether the extensions of the homeomorphic embedding relation proposed in [56] and [31] (in the context of static termination analysis of term rewrite systems) can be useful in an online setting.

In summary, we have shed new light on the relation between wqo's and wfo's and have formally shown why wqo's are more interesting than wfo's for ensuring termination in an online setting (such as program specialisation or analysis). We have illustrated the inherent flexibility of  $\sqsubseteq$  and proved that, despite its

simplicity, it is strictly more generous than the class of monotonic wfo's. As all the wfo's used for automatic online termination (so far) are actually monotonic, this formally establishes the interest of  $\sqsubseteq$  in that context. We have also compared to techniques based upon nearly-foundedness, and have shown that such techniques—contrary to  $\sqsubseteq$ — can lead to the undesirable property of strict over-eagerness.

We have also presented new embedding relations  $\sqsubseteq^+$ ,  $\sqsubseteq_{var}$  and  $\sqsubseteq^*$ , which inherit all the good properties of  $\sqsubseteq$  while providing a refined treatment of (logical) variables. We believe that these refinements can be of value in other contexts and for other languages (such as in the context of partial evaluation of functional-logic programs [3, 2, 29] or of supercompilation [64, 16, 62] of functional programming languages, where — at specialisation time — variables also appear). We also believe that  $\sqsubseteq^*$  provides both a theoretically and practically more satisfactory basis than  $\sqsubseteq^+$  or  $\sqsubseteq$ .

We also believe that  $\sqsubseteq^*$  can play a role in other areas, such as ensuring termination and controlling abstraction of model checking of infinite systems. First promising results were obtained in [15] using ECCE [32] applied to systems expressed using Petri nets and the  $\pi$ -calculus.

### Acknowledgements

I would like to thank Danny De Schreye, Robert Glück, Jesper Jørgensen, Bern Martens, Maurizio Proietti, Jacques Riche and Morten Heine Sørensen for all the discussions and joint research which led to this paper. Anonymous referees, Patrick Cousot, Renaud Marlet, and Bern Martens provided extremely useful feedback on this paper.

### References

1. E. Albert, M. Alpuente, M. Falaschi, P. Julián, and G. Vidal. Improving control in functional logic program specialization. In G. Levi, editor, *Static Analysis. Proceedings of SAS'98*, LNCS 1503, pages 262–277, Pisa, Italy, September 1998. Springer-Verlag.
2. M. Alpuente, M. Falaschi, P. Julián, and G. Vidal. Specialisation 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.
3. M. Alpuente, M. Falaschi, and G. Vidal. Narrowing-driven partial evaluation of functional logic programs. In H. Riis Nielson, editor, *Proceedings of the 6th European Symposium on Programming, ESOP'96*, LNCS 1058, pages 45–61. Springer-Verlag, 1996.
4. K. R. Apt. Introduction to logic programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 10, pages 495–574. North-Holland Amsterdam, 1990.
5. R. Bol. Loop checking in partial deduction. *The Journal of Logic Programming*, 16(1&2):25–46, 1993.
6. A. F. Bowers. Representing Gödel object programs in Gödel. Technical Report CSTR-92-31, University of Bristol, November 1992.

7. 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.
8. 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.
9. N. Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, 3:69–116, 1987.
10. N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B*, pages 243–320. Elsevier, MIT Press, 1990.
11. N. Dershowitz and Z. Manna. Proving termination with multiset orderings. *Communications of the ACM*, 22(8):465–476, 1979.
12. 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.
13. J. H. Gallier. What's so special about Kruskal's theorem and the ordinal  $\Gamma_0$ ? A survey of some results in proof theory. Technical report, University of Pennsylvania, October 1993.
14. 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.
15. R. Glück and M. Leuschel. Abstraction-based partial deduction for solving inverse problems – a transformational approach to software verification. Submitted, 1998.
16. R. Glück and M. H. Sørensen. A roadmap to supercompilation. In O. Danvy, R. Glück, and P. Thiemann, editors, *Proceedings of the 1996 Dagstuhl Seminar on Partial Evaluation*, LNCS 1110, pages 137–160, Schloß Dagstuhl, 1996. Springer-Verlag.
17. J. Gustedt. *Algorithmic Aspects of Ordered Structures*. PhD thesis, Technische Universität Berlin, 1992.
18. M. Hanus. The integration of functions into logic programming. *The Journal of Logic Programming*, 19 & 20:583–628, May 1994.
19. P. Henderson. *Functional Programming: Application and Implementation*. International Series in Computer Science. Prentice-Hall, Inc., Englewood Cliffs, N.J., 1980.
20. G. Higman. Ordering by divisibility in abstract algebras. *Proceedings of the London Mathematical Society*, 2:326–336, 1952.
21. P. Hill and J. Gallagher. Meta-programming in logic programming. In D. M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 5, pages 421–497. Oxford Science Publications, Oxford University Press, 1998.
22. P. Hill and J. W. Lloyd. *The Gödel Programming Language*. MIT Press, 1994.
23. S. Hölldobler. *Foundations of Equational Logic Programming*. LNAI 353. Springer-Verlag, 1989.
24. G. Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *Journal of the ACM*, 27(4):797–821, 1980.
25. J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *The Journal of Logic Programming*, 19 & 20:503–581, 1994.
26. N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.

27. J. Jørgensen, M. Leuschel, and B. Martens. Conjunctive partial deduction in practice. In J. Gallagher, editor, *Proceedings of the International Workshop on Logic Program Synthesis and Transformation (LOPSTR'96)*, LNCS 1207, pages 59–82, Stockholm, Sweden, August 1996. Springer-Verlag.
28. J. B. Kruskal. Well-quasi ordering, the tree theorem, and Vazsonyi's conjecture. *Transactions of the American Mathematical Society*, 95:210–225, 1960.
29. L. Lafave and J. Gallagher. Constraint-based partial evaluation of rewriting-based functional logic programs. In N. Fuchs, editor, *Proceedings of the International Workshop on Logic Program Synthesis and Transformation (LOPSTR'97)*, LNCS 1463, Leuven, Belgium, July 1998.
30. P. Lescanne. Rewrite orderings and termination of rewrite systems. In A. Tarlecki, editor, *Mathematical Foundations of Computer Science 1991*, LNCS 520, pages 17–27, Kazimierz Dolny, Poland, September 1991. Springer-Verlag.
31. P. Lescanne. Well rewrite orderings and well quasi-orderings. Technical Report N° 1385, INRIA-Lorraine, France, January 1991.
32. M. Leuschel. The ECCE partial deduction system and the DPPD library of benchmarks. Obtainable via <http://www.cs.kuleuven.ac.be/~dtai>, 1996.
33. M. Leuschel. *Advanced Techniques for Logic Program Specialisation*. PhD thesis, K.U. Leuven, May 1997. Accessible via <http://www.cs.kuleuven.ac.be/~michael>.
34. M. Leuschel. A formal comparison of well-founded and well-quasi orders for online termination. In P. Flener, editor, *Logic Program Synthesis and Transformation. Pre-Proceedings of LOPSTR'98*, Manchester, UK, June 1998.
35. 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.
36. M. Leuschel and D. De Schreye. Creating specialised integrity checks through partial evaluation of meta-interpreters. *The Journal of Logic Programming*, 36:149–193, 1998.
37. M. Leuschel and B. Martens. Partial deduction of the ground representation and its application to integrity checking. In J. W. Lloyd, editor, *Proceedings of ILPS'95, the International Logic Programming Symposium*, pages 495–509, Portland, USA, December 1995. MIT Press.
38. M. Leuschel and B. Martens. Global control for partial deduction through characteristic atoms and global trees. In O. Danvy, R. Glück, and P. Thiemann, editors, *Proceedings of the 1996 Dagstuhl Seminar on Partial Evaluation*, LNCS 1110, pages 263–283, Schloß Dagstuhl, 1996. Springer-Verlag.
39. 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.
40. N. Lindenstrauss, Y. Sagiv, and A. Serebrenik. Unfolding the mystery of mergesort. In N. Fuchs, editor, *Proceedings of the International Workshop on Logic Program Synthesis and Transformation (LOPSTR'97)*, LNCS 1463, Leuven, Belgium, July 1998.
41. J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987.
42. J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *The Journal of Logic Programming*, 11(3& 4):217–242, 1991.
43. R. Marlet. *Vers une Formalisation de l'Évaluation Partielle*. PhD thesis, Université de Nice - Sophia Antipolis, December 1994.
44. B. Martens. *On the Semantics of Meta-Programming and the Control of Partial Deduction in Logic Programming*. PhD thesis, K.U. Leuven, February 1994.

45. B. Martens and D. De Schreye. Two semantics for definite meta-programs, using the non-ground representation. In K. R. Apt and F. Turini, editors, *Meta-logics and Logic Programming*, pages 57–82. MIT Press, 1995.
46. B. Martens and D. De Schreye. Why untyped non-ground meta-programming is not (much of) a problem. *The Journal of Logic Programming*, 22(1):47–99, 1995.
47. B. Martens and D. De Schreye. Automatic finite unfolding using well-founded measures. *The Journal of Logic Programming*, 28(2):89–146, August 1996.
48. B. Martens, D. De Schreye, and T. Horváth. Sound and complete partial deduction with unfolding based on well-founded measures. *Theoretical Computer Science*, 122(1–2):97–117, 1994.
49. B. Martens and J. Gallagher. Ensuring global termination of partial deduction while allowing flexible polyvariance. In L. Sterling, editor, *Proceedings ICLP'95*, pages 597–613, Kanagawa, Japan, June 1995. MIT Press.
50. J. Martin. Sonic partial deduction. Technical report, Department of Electronics and Computer Science, University of Southampton, 1998.
51. P.-A. Melliès. On a duality between Kruskal and Dershowitz theorems. In K. G. Larsen, editor, *Proceedings of ICALP'98*, LNCS, Aalborg, Denmark, 1998. Springer-Verlag.
52. A. Middeldorp and H. Zantema. Simple termination of rewrite systems. *Theoretical Computer Science*, 175(1):127–158, 1997.
53. C. R. Murthy and J. R. Russel. A constructive proof of Higman's lemma. In *Proceedings, Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 257–267, Alamo, California, 1990. IEEE Computer Society Press.
54. C. Nash-Williams. On well-quasi-ordering finite trees. *Proc. Cambridge Phil. Soc.*, 59:833–835, 1963.
55. L. Plümer. *Termination Proofs for Logic Programs*. LNCS 446. Springer-Verlag, 1990.
56. L. Puel. Using unavoidable set of trees to generalize Kruskal's theorem. *Journal of Symbolic Computation*, 8:335–382, 1989.
57. J. Riche. *Decidability, Complexity and Automated Reasoning in Relevant Logic*. PhD thesis, Australian National University, September 1991.
58. E. Ruf. *Topics in Online Partial Evaluation*. PhD thesis, Stanford University, March 1993.
59. D. Sahlin. Mixtus: An automatic partial evaluator for full Prolog. *New Generation Computing*, 12(1):7–51, 1993.
60. S. G. Simpson. Ordinal numbers and the Hilbert basis theorem. *Journal of Symbolic Logic*, 53(3):961–974, 1988.
61. 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.
62. M. H. Sørensen, R. Glück, and N. D. Jones. A positive supercompiler. *Journal of Functional Programming*, 6(6):811–838, 1996.
63. J. Stillman. *Computational Problems in Equational Theorem Proving*. PhD thesis, State University of New York at Albany, 1988.
64. V. F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, 1986.
65. W. Vanhoof and B. Martens. To parse or not to parse. In N. Fuchs, editor, *Proceedings of the International Workshop on Logic Program Synthesis and Transformation (LOPSTR'97)*, LNCS 1463, Leuven, Belgium, July 1997.

66. A. Weiermann. Complexity bounds for some finite forms of Kruskal's theorem. *Journal of Symbolic Computation*, 18(5):463–488, November 1994.
67. D. Weise, R. Conybeare, E. Ruf, and S. Seligman. Automatic online partial evaluation. In *Proceedings of the Conference on Functional Programming Languages and Computer Architectures*, LNCS 523, pages 165–191, Harvard University, 1991. Springer-Verlag.

## A Assorted proofs

**Lemma 1 (wbr from wfo)** Let  $<_V$  be a well-founded order on  $V$ . Then  $\preceq_V$ , defined by  $v_1 \preceq_V v_2$  iff  $v_1 \not>_V v_2$ , is a wbr on  $V$ . Furthermore,  $<_V$  and  $\preceq_V$  have the same set of admissible sequences.

*Proof.* Suppose that there is an infinite sequence  $v_1, v_2, \dots$  of elements of  $V$  such that, for all  $i < j$ ,  $v_i \not\preceq_V v_j$ . By definition this means that, for all  $i < j$ ,  $v_i >_V v_j$ . In particular this means that we have an infinite sequence with  $v_i >_V v_{i+1}$ , for all  $i \geq 1$ . We thus have a contradiction with Definition 1 of a well-founded order and  $\preceq_V$  must be a wbr on  $V$ .

The above already shows that every admissible sequence of  $\preceq_V$  is also an admissible sequence of  $<_V$ . Finally, it is obvious, by transitivity of  $>_V$ , that whenever we have an admissible sequence  $v_1, v_2, \dots$  of  $<_V$  (i.e.  $v_i >_V v_{i+1}$  for all  $i \geq 1$ ) that for all  $i < j$ :  $v_i \not\preceq_V v_j$

**Corollary 1.** Let  $\|\cdot\|$  be a monotonic norm on expressions. Then  $s \approx_{\preceq} t \Rightarrow \|s\| = \|t\|$ .

*Proof.* If  $s \approx_{\preceq} t$  we know by Proposition 4 that neither  $\|s\| < \|t\|$  nor  $\|s\| > \|t\|$  and we must thus have  $\|s\| = \|t\|$ .

**Lemma 2.** If  $s \approx_{\preceq} t$  then the diving rule (2.) was not used in Definition 4.

*Proof.* The wfo  $\|\cdot\|_{sym}$  is monotonic (see Proposition 2). Hence, by Corollary 1, we know that  $\|s\|_{sym} = \|t\|_{sym}$ . Now, if we try to apply the diving rule (2.) to establish that  $s \preceq t$  with  $t = f(t_1, \dots, t_n)$  we have that  $s \preceq t_i$ , i.e.  $\|s\|_{sym} \leq \|t_i\|_{sym}$  and we have a contradiction as  $\|t_i\|_{sym} \leq \|t\|_{sym} - 1 = \|s\|_{sym} - 1$ . Thus, at the outermost level one can only apply rule (1.)— and we are finished— or the coupling rule (3.). If we apply rule (3.) then we actually have  $s_i \approx_{\preceq} t_i$  (as the rule must have also been applied to deduce  $t \preceq s$ ) and the lemma holds by induction on the structure of  $s$  and  $t$ .

**Theorem 2** The relation  $\preceq_{var}$  is a wqo on the set of expression over a finite alphabet.

*Proof.* First  $\preceq_{var}$  is trivially transitive as we have the following (where we denote that  $s$  and  $t$  are variants by  $s \approx_{var} t$ ):

- $s \preceq t \wedge t \preceq u \Rightarrow s \preceq u$
- $s \preceq t \wedge t \approx_{var} u \Rightarrow s \preceq u$
- $s \approx_{var} t \wedge t \preceq u \Rightarrow s \preceq u$
- $s \approx_{var} t \wedge t \approx_{var} u \Rightarrow s \approx_{var} u$

Now let us take an infinite sequence  $\omega = v_1, v_2, \dots$ . In order for  $\trianglelefteq_{var}$  to satisfy the wbr requirement we have to show that for some  $i < j$  we have  $v_i \trianglelefteq_{var} v_j$ . If there two variants  $v_i$  and  $v_j$  in the sequence  $\omega$  then  $v_i \trianglelefteq_{var} v_j, v_j \trianglelefteq_{var} v_i$  and  $\trianglelefteq_{var}$  thus satisfies the wbr requirement. So let us assume that there are no variants in  $v_1, v_2, \dots$ . Let us now remove all  $v_i, i > 1$  such that  $v_i \approx_{\trianglelefteq} v_1$  from the sequence. By Lemma 2 we know that there can only be finitely many such  $v_i$  (because the diving rule was not used and because there are no variants) and the remaining sequence  $\omega'$  is still infinite. Now let us repeat this construction with the second element of  $\omega'$ , then with the third element of  $\omega''$ , then  $\dots$ . We will thus obtain an infinite sequence  $w_1, w_2, w_3, \dots$ . We know that  $\trianglelefteq$  is a wqo. Thus for some  $i < j$   $w_i \trianglelefteq w_j$ . By construction we know that  $w_j \not\trianglelefteq w_i$ , hence  $w_i \triangleleft w_j$  and thus  $w_i \trianglelefteq_{var} w_j$ .

**Theorem 3**  $\trianglelefteq^*$  is a wbr on expressions. Additionally, if  $\preceq_{fun}$  and  $\preceq_{sexp}$  are wqo's then so is  $\trianglelefteq^*$ .

*Proof.* Let us first suppose that  $\preceq_{fun}$  and  $\preceq_{sexp}$  are wqo's. We know by Higman-Kruskal's theorem [28] that without the extra condition of point 3 requiring  $\langle s_1, \dots, s_m \rangle \preceq_{sexp} \langle t_1, \dots, t_n \rangle$  we have a wqo. Let us refer to this wqo by  $\trianglelefteq$ . We now simply define the following mapping  $\|\cdot\|$  from expressions to expressions.

- $\|X\| = X$
- $\|f(t_1, \dots, t_n)\| = c(\llbracket \langle t_1, \dots, t_n \rangle \rrbracket, f(\|t_1\|, \dots, \|t_n\|))$

where  $c$  is a new unused binary functor and  $\llbracket \cdot \rrbracket$  is a an encoding generating new distinct unused functors for each sequence of expressions. We also extend  $\preceq_{fun}$  for these functors by saying that  $\llbracket s \rrbracket \preceq_{fun} \llbracket t \rrbracket$  iff  $s \preceq_{sexp} t$  as well as  $c \preceq c$  and  $c$  is incomparable to any other symbol. Given any two expressions  $s$  and  $t$  we now have that  $\|s\| \trianglelefteq \|t\| \Rightarrow s \trianglelefteq^* t$ , as can be shown by structural induction (sketch):

- Diving (rule 2., concluding  $s \trianglelefteq^* f(t_1, \dots, t_n)$ ) into the first argument of  $\|t\|$  will never be successful, as we supposed that the constants  $\llbracket \langle t_1, \dots, t_n \rangle \rrbracket$  are distinct from the initial functors. We therefore have to dive into the second argument of  $\|t\| = c(\llbracket \langle t_1, \dots, t_n \rangle \rrbracket, f(\|t_1\|, \dots, \|t_n\|))$  and as we have  $c \not\preceq_{fun} f$  we have to dive into one of the arguments  $\|t_i\|$  at the next step as well. Hence the corresponding diving into  $t_i$  must be applicable for  $s$  and  $t = f(t_1, \dots, t_n)$  and we will be able to deduce  $s \trianglelefteq^* t$  by the induction hypothesis.
- For the coupling rule 3 (concluding  $f(s_1, \dots, s_m) \trianglelefteq^* g(t_1, \dots, t_n)$ ), we know that  $\llbracket \langle s_1, \dots, s_m \rangle \rrbracket \trianglelefteq \llbracket \langle t_1, \dots, t_n \rangle \rrbracket$  as there is no choice for the coupling of  $c$  (it occurs only as a binary functor, i.e., the first argument of  $\|s\|$  has to be compared with the first argument of  $\|t\|$ ). In other words, as  $\llbracket \cdot \rrbracket$  generates only constants, we know that  $\llbracket \langle s_1, \dots, s_m \rangle \rrbracket \preceq_{fun} \llbracket \langle t_1, \dots, t_n \rangle \rrbracket$  which is equivalent to  $\langle s_1, \dots, s_m \rangle \preceq_{sexp} \langle t_1, \dots, t_n \rangle$ . So the additional criterion of the rule 3. of  $\trianglelefteq^*$  is satisfied. We can now apply the induction hypothesis for each  $\|s_j\| \trianglelefteq \|t_i\|$  to conclude that  $s_j \trianglelefteq^* t_i$  must hold. Hence, we can conclude that  $f(s_1, \dots, s_m) \trianglelefteq^* g(t_1, \dots, t_n)$ .

Thus, as  $\trianglelefteq$  is a wqo so is  $\trianglelefteq^*$  (for every infinite sequence of terms  $t_1, t_2, \dots$  we must have some  $i < j$  such that  $\|t_i\| \trianglelefteq \|t_j\|$  and hence  $t_i \trianglelefteq^* t_j$  as well).

In case that  $\preceq_{fun}$  and  $\preceq_{sexp}$  are not (both) wqo's we can still apply the same reasoning as above as Kruskal's theorem extends to wbr's (see [51]).

## B Small Experiments with the ECCE system

The purpose of this appendix is to illustrate the flexibility which the homeomorphic embedding relation provides straight “out of the box” (other more

intricate well-quasi orders, like the one used by Mixtus [59], can handle some of the examples below as well).

For that we experiment with the ECCE partial deduction system [32] using an unfolding rule based on  $\triangleleft$  which allows the selection of either determinate literals or left-most literals within a goal, given that no covering ancestor [7] is embedded (via  $\triangleleft$ ). To ease readability, the specialised programs are sometimes presented in unrenamed form.

First, let us take the *mergesort* program, which is somewhat problematic for a lot of static termination analysis methods [55, 40]. Let us first illustrate the generosity of  $\triangleleft$  on the *quicksort* program (as taken from [55]):

```

quicksort([], []).
quicksort([H|L], S) :-
    split(H, L, [], [], A, B),
    quicksort(A, A1), quicksort(B, B1),
    append(A1, [H|B1], S).

append([], L, L).
append([H|X], Y, [H|Z]) :- append(X, Y, Z).

split(H, [X|L], A, B, A1, B1) :- X <= H, split(H, L, [X|A], B, A1, B1).
split(H, [X|L], A, B, A1, B1) :- X > H, split(H, L, A, [X|B], A1, B1).
split(H, [], A, B, A, B).

```

The partial evaluation query:

```
?- quicksort([3, 1, 2], X).
```

The resulting specialised program is as follows (and full unfolding has been achieved):

```
quicksort([3, 1, 2], [1, 2, 3]).
```

Next, let us take the *mergesort* program, which is somewhat problematic for a lot of static termination analysis methods [55, 40].

```

mergesort([], []).
mergesort([X], [X]).
mergesort([X, Y|Xs], Ys) :-
    split([X, Y|Xs], X1s, X2s),
    mergesort(X1s, Y1s), mergesort(X2s, Y2s),
    merge(Y1s, Y2s, Ys).

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

merge([], Xs, Xs).
merge(Xs, [], Xs).
merge([X|Xs], [Y|Ys], [X|Zs]) :- X <= Y, merge(Xs, [Y|Ys], Zs).
merge([X|Xs], [Y|Ys], [Y|Zs]) :- X > Y, merge([X|Xs], Ys, Zs).

```

The partial evaluation query:

```
?- mergesort([3,1,2],X).
```

As the following resulting specialised program shows, homeomorphic embedding allowed the full unfolding of *mergesort*:

```
mergesort([3,1,2],[1,2,3]).
```

It took ECCE less than 0.5 s on a Sparc Classic to produce the above program (including post-processing and writing to file).

We can even achieve this same feat even if we interpose one or more levels of metainterpretation! Take a vanilla solve metainterpreter with mergesort at the object-level:

```
solve([]).
solve([H|T]) :-
    claus(H,Bdy),solve(Bdy),solve(T).

claus(mergesort([],[]), []).
claus(mergesort([X],[X]), []).
claus(mergesort([X,Y|Xs],Ys),
      [split([X,Y|Xs],X1s,X2s),
       mergesort(X1s,Y1s),mergesort(X2s,Y2s),
       merge(Y1s,Y2s,Ys) ]).
claus(split([],[],[]), []).
claus(split([X|Xs],[X|Ys],Zs), [split(Xs,Zs,Ys)]).
claus(merge([],Xs,Xs), []).
claus(merge(Xs,[],Xs), []).
claus(merge([X|Xs],[Y|Ys],[X|Zs]),
      [X =< Y, merge(Xs,[Y|Ys],Zs)]).
claus(merge([X|Xs],[Y|Ys],[Y|Zs]),
      [X>Y, merge([X|Xs],Ys,Zs)]).
claus('=<'(X,Y),[]) :- X =< Y.
claus('>'(X,Y),[]) :- X > Y.

mergesort_test(X) :- solve([mergesort([3,1,2],X)]).
```

The partial evaluation query:

```
?- mergesort_test(X).
```

Again homeomorphic embedding allowed the full unfolding:

```
mergesort_test([1,2,3]).
```

It took ECCE 2.86 s on a Sparc Classic to produce the above program (including post-processing and writing to file).

Next, take this small Datalog program computing the transitive closure of a graph:

```

arc(a,b).
arc(b,c).

trans(X,Y) :- arc(X,Y).
trans(X,Z) :- arc(X,Y),trans(Y,Z).

```

The partial evaluation query:

```
?- trans(a,X).
```

Again full unfolding was accomplished, as illustrated by the following specialised program:

```

trans(a,b).
trans(a,c).

```

The following example is taken from [44].

```

produce([], []).
produce([X|Xs],[X|Ys]) :- produce(Xs,Ys).

consume([]).
consume([X|Xs]) :- consume(Xs).

```

The partial evaluation query:

```
?- produce([1,2|X],Y), consume(Y).
```

To solve it in the setting of unfolding based upon wfo's one needs both partition based measure functions as well as taking the context into account. The same adequate unfolding can simply be achieved by  $\leq$  based determinate unfolding, as illustrated by the following specialised program derived by ECCE (default settings):

```

produce_conj__1([], [1,2]).
produce_conj__1([X1|X2],[1,2,X1|X3]) :-
    produce_conj__2(X2,X3).
produce_conj__2([], []).
produce_conj__2([X1|X2],[X1|X3]) :-
    produce_conj__2(X2,X3).

```