Homeomorphic Embedding for Online Termination of Symbolic Methods

Michael Leuschel

Department of Electronics and Computer Science
University of Southampton, Southampton SO17 1BJ, UK
`mal@ecs.soton.ac.uk`

Abstract. Well-quasi orders in general, and homeomorphic embedding in particular, have gained popularity to ensure the termination of techniques for program analysis, specialisation, transformation, and verification. In this paper we survey and discuss this use of homeomorphic embedding and clarify the advantages of such an approach over one using well-founded orders. We also discuss various extensions of the homeomorphic embedding relation. We conclude with a study of homeomorphic embedding in the context of metaprogramming, presenting some new (positive and negative) results and open problems.
Keywords: Termination, Well-quasi orders, Program Analysis, Specialisation and Transformation, Logic Programming, Functional & Logic Programming, Metaprogramming, Infinite Model Checking.

1 Introduction

The problem of ensuring termination arises in many areas of computer science. It is especially important within all areas of automatic program analysis, synthesis, verification, specialisation, and transformation: one usually strives for methods which are guaranteed to terminate. It can also be an issue for other symbolic methods such as model checking, e.g., for infinite state systems. One can basically distinguish two kinds of techniques for guaranteeing termination:

- *offline* (or *static*) techniques (e.g., [12, 15–17, 64]), which *prove* termination of a program or process *beforehand* without any kind of execution, and
- *online* (or *dynamic*) techniques, which *ensure* termination of a process *during* its execution.

Offline approaches have less information at their disposal but do not require runtime intervention (which might be impossible). Which of the two approaches is taken depends entirely on the application area.

Well-quasi orders [10, 70] and in particular *homeomorphic embedding* [74, 53, 6, 2, 13] have become very popular to ensure online termination. In this paper we survey the now widespread use of these techniques and try to clarify (formally and informally) why these approaches have become so popular. We will also pay particular attention to the issue of metaprogramming.

2 Symbolic Methods and Online Termination

In this section we first introduce the general notion of a symbolic method which manipulates symbolic expressions.

Symbolic Expressions First, an *alphabet* consists of the following classes of symbols: 1) *variables*; and 2) *function symbols*. Function symbols have an associated *arity*, a natural number indicating how many arguments they take in the definitions below. *Constants* are function symbols with arity 0.

In the remainder, we suppose the set of variables is countably infinite and the set of function and predicate symbols is countable. In addition, alphabets with a finite set of function symbols will be called *finite*.

We will adhere as much as possible to the following syntactical conventions (stemming from logic programming [7, 56]):
– Variables will be denoted by upper-case letters like X, Y, Z, usually taken from the end of the (Latin) alphabet.
– Constants will be denoted by lower-case letters like a, b, c, usually taken from the beginning of the (Latin) alphabet.
– The other function symbols will be denoted by lower-case letters like f, g, h.

Definition 1. *The set of (symbolic) expressions $\mathcal{E}$ (over some given alphabet) is inductively defined as follows:*
– *a variable is an expression,*
– *a function symbol f of arity $n \geq 0$ applied to a sequence $t_1, \ldots, t_n$ of n expressions, denoted by $f(t_1, \ldots, t_n)$, is also an expression.*

For terms representing lists we will use the usual Prolog notation: e.g. $[\,]$ denotes the empty list, $[H|T]$ denotes a non-empty list with first element H and tail T. As usual we can apply substitutions to expressions, and we will use the conventions of [7, 56].

Symbolic Methods Inspired by [67, 73], we now present a general definition of symbolic methods:

Definition 2. *A* configuration *is a finite tree whose nodes are labelled with expressions. Given two configurations τ_1, τ_2, we say that $\tau_1 \preceq \tau_2$ iff τ_2 can be obtained from τ_1 by attaching sub-trees to the leaves of τ_1.*
A symbolic method *sm is a map from configurations to configurations such that $\forall \tau : \tau \preceq sm(\tau)$. The symbolic method is said to* terminate *for some initial configuration τ_0 iff for some $i \in \mathbb{N}$: $sm^i(\tau_0) = sm^{i+1}(\tau_0)$, where we define sm^i inductively by $sm^0(\tau_0) = \tau_0$ and $sm^{i+1}(\tau_0) = sm(sm^i(\tau_0))$.*

This definition of a symbolic method is very general and naturally covers a wide range of interesting techniques, as shown in Fig. 1. For example, for partial evaluation [38, 36, 65], an expression is a partially specified call and the symbolic method will perform evaluation or unfolding. Another symbolic method is (conjunctive) partial deduction [57, 25, 48, 13, 49] of logic programs. Here the expressions are (conjunctions of) atoms and children are derived using resolution.[1]

---

[1] One could argue that there are actually two symbolic methods: one relating to the so-called local control and the other to the global control.

Other techniques are, e.g., supercompilation of functional programs [79, 30, 76, 75] and partial evaluation of functional logic programs [5, 6, 2, 42].

Also, a lot of program transformation techniques can be cast into the above form [66, 67]. Moreover, several algorithms for (infinite) model checking naturally follow the above form. Examples are the Karp-Miller procedure [40] for Petri nets, Finkel's minimal coverability procedure [18] for Petri nets, and also the backwards reachability algorithm for well-structured transition systems [1, 19]. Here, expressions in the tree are symbolic representations of sets of states of a system to be analysed and the symbolic method is either symbolically computing new successor or predecessor states. Probably, many other techniques (e.g., theorem proving) can be cast quite naturally into the above form.

Symbolic Method	Expressions	$sm(.)$
partial evaluation	terms with variables	evaluation + unfolding
supercompilation	terms with variables	driving
(conjunctive) partial deduction	conjunctions of atoms	resolution
Karp-Miller procedure	markings over $I\!N \cup \{\omega\}$	firing transitions

Fig. 1. Some symbolic methods

Whistles and online termination Quite often, symbolic methods do not terminate on their own, and we need some kind of supervisory process which spots potential non-termination (and then initiates some appropriate action). Formally, this supervision process can be defined as follows:

Definition 3. *A* whistle W *is a subset of all configurations. The trees in W are called* inadmissible *wrt W, the others are called* admissible *wrt W.*

Usually whistles are upwards-closed, i.e., $\tau_1 \preceq \tau_2 \wedge \tau_1 \in W \Rightarrow \tau_2 \in W$. In other words, if a whistle considers τ_1 to be dangerous then it will also consider any extension of τ_1 dangerous.

Now, a combination of a symbolic method sm with a whistle W is said to *terminate* iff either sm terminates or for some i: $sm^i(\tau_0) \in W$.

Intuitively, we can view this as the whistle supervising the method sm and "blowing" if it produces an inadmissible configuration ($sm^i(\tau_0) \in W$). What happens after the whistle blows depends on the particular application, and we will not go into great detail. In program specialisation or transformation one would usually generalise and restart. To ensure termination of the whole process one has to ensure that we cannot do an infinite number of restarts. This can basically be solved by viewing this generalisation and restart procedure as another symbolic method and applying the whistle approach to it as well; see [62, 73] for more details.

In a lot of cases, whistles are first defined on sequences of expressions, and then extended to configurations:

Definition 4. *A* whistle over sequences W *is a subset of all finite sequences of expressions. The sequences in W are called* inadmissible *wrt W*.

Given W, one will then often use the extension W_c to configurations , defined as follows: $W_c = \{\tau \mid \exists$ a branch γ of τ s.t. $\gamma \in W\}$. One can also use more refined extensions For example, one might focus on subsequences (see, e.g., the focus on selected literals and covering ancestors concept in $[11, 61, 60, 59]$).

In the remainder of the paper, we will mainly focus on whistles over sequences, and only occasionally discuss their extension to trees.

3 Subsumption, Depth Bounds, and Wfos

We now present some of the whistles used in early approaches to online termination.

Subsumption and Variant Checking The idea is to use subsumption testing to detect dangerous sequences. Formally, such a whistle is defined as follows: $W_{Sub} = \{e_1, e_2, \dots \mid \exists i < j$ such that $e_i\theta = e_j\}$ In other words, a sequence is inadmissible if an element is an instance of a preceding expression. This approach was used, e.g., in early approaches to partial deduction (e.g., $[78, 23, 9]$). It fares pretty well on some simple examples, but, as shown in $[11]$ (see also Ex. 1 below), it is not sufficient to ensure termination in the presence of accumulators. Sometimes variant testing is used instead of subsumption, making more sequences admissible but also worsening the non-termination problem.

Depth Bounds One, albeit ad-hoc, way to solve the local termination problem is to simply impose an arbitrary depth bound D as follows: $W_D = \{e_1e_2 \dots e_k \dots \mid k > D\}$. This approach is taken, e.g., in early partial deduction systems which unfold every predicate at most once. A depth bound is of course not motivated by any property, structural or otherwise, of the program or system under consideration, and is therefore both practically and theoretically unsatisfactory.

Well-founded Orders A more refined approach to ensure termination is based on well-founded orders. Such an approach was first used for partial evaluation of functional programs in $[69, 82]$ and for partial deduction in $[11, 61, 60, 59]$. These techniques ensure termination, while at the same time allowing symbolic manipulations related to the structural aspect of the program or system under consideration.

Formally, well-founded orders are defined as follows:

Definition 5. *A* strict partial order $<$ *is an irreflexive, transitive, and thus asymmetric, binary relation on $\mathcal{E}$. The whistle $W_<$ associated with $<$ is defined by $W_< = \{e_1, e_2, \dots \mid \exists i$ such that $e_{i+1} \not< e_i\}$. We call $<$ a* well-founded order *(wfo) iff all infinite sequences of expressions are contained in $W_<$*.

In practice, wfos can be defined by so-called *norms*, which are mappings from expressions to $I\!N$ and thus induce an associated wfo.

Example 1. The following sequence of symbolic expressions arises when partially deducing the "reverse with accumulator" logic program [11]:

$$rev([a, b | T], [\,], R), \; rev([b | T], [a], R), \; rev(T, [b, a], R), \; rev(T', [H', b, a], R), \, \ldots$$

A simple well-founded order on expressions of the form $rev(t_1, t_2, t_3)$ might be based on comparing the *termsize* norm (i.e., the number of function and constant symbols) of the first argument. We then define the wfo for this example by:

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

Based on that wfo, the subsequence consisting of the first 3 expressions is admissible, but any further extension is not. At that point the partial deduction would stop and initiate a generalisation step.

In the above example we have fixed the wfo order beforehand. This is often quite difficult and (with the possible exception of [22] [63]) not very often done in practice. A more widely used approach for generating suitable wfos is to determine them while running the symbolic method. [11, 61, 60, 59], start off with a simple, but safe wfo and then refine this wfo during the unfolding process.

However, it has been felt by several researchers that well-founded orders are sometimes too rigid or (conceptually) too complex in an online setting. In the next section we introduce a more flexible approach which has gained widespread popularity to ensure online termination of symbolic techniques.

4 Wqos and Homeomorphic Embedding

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

Definition 6. *A* quasi order *is a reflexive and transitive binary relation on $\mathcal{E}$.*

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 and binary relations. We will use either "directionality" as is convenient in the context.

Definition 7. (wbr,wqo) *Let $\leq$ be a binary relation on $\mathcal{E}$. The whistle $W_{\leq}$ associated with $\leq$ is defined as follows $W_{\leq} = \{e_1, e_2, \ldots \mid \exists i < j$ such that $e_i \leq e_j\}$. We say that $\leq$ is a* well-binary relation (wbr) *iff all infinite sequences of expressions are contained in $W_{\leq}$. If $\leq$ is also a quasi order then $\leq$ is called a* well-quasi order (wqo).*

Observe that, in contrast to wfos, non-comparable elements are allowed within admissible sequences. There are several other equivalent definitions of well-binary relations and well-quasi orders [33, 44, 81]. Traditionally, wqos have been used within *static* termination analysis to construct well-founded orders [15, 16]. The use of well-quasi orders in an *online* setting has only emerged quite

recently. In that setting, transitivity of a wqo is usually not that interesting (because one does not have to generate wfos) and one can therefore drop this requirement, leading to the use of wbr's (see also Sect. 7).

An interesting wqo is the *homeomorphic embedding* relation $\trianglelefteq$. The following is the definition from [74], which adapts the pure[2] homeomorphic embedding from [16] by adding a simple treatment of variables.

Definition 8. *The* (pure) homeomorphic embedding *relation $\trianglelefteq$ on expressions is inductively defined as follows (i.e. $\trianglelefteq$ is the least relation satisfying the rules):*

    1. $X \trianglelefteq Y$ for all variables X, Y
    2. $s \trianglelefteq f(t_1, \ldots, t_n)$ if $s \trianglelefteq t_i$ for some i
    3. $f(s_1, \ldots, s_n) \trianglelefteq f(t_1, \ldots, t_n)$ if $n \geq 0$ and $\forall i \in \{1, \ldots, n\} : s_i \trianglelefteq t_i$.

The second rule is sometimes called the *diving* rule, and the third rule is sometimes called the *coupling* rule (notice that n is allowed to be 0 and we thus have $c \trianglelefteq c$ for all constants). When $s \trianglelefteq t$ we also say that s is *embedded in* t or t is *embedding* s. By $s \lhd t$ we denote that $s \trianglelefteq t$ and $t \ntrianglelefteq s$.

The intuition behind the above definition is that $A \trianglelefteq B$ iff A can be obtained from B by removing some symbols, i.e., that the structure of A, split in parts, reappears within B. For instance, we have $p(a) \trianglelefteq p(f(a))$ because $p(a)$ can be obtained from $p(f(a))$ by removal of "$f(.)$" Observe that the removal corresponds to the application of rule 2 and that we also have $p(a) \lhd p(f(a))$. Another example is $f(a, b) \lhd p(f(g(a)), b)$ but $p(a) \ntrianglelefteq p(b)$ and $f(a, b) \ntrianglelefteq p(f(a), f(b))$. Finally, when adding variables, we have, e.g.: $p(X) \lhd p(f(Y))$, $p(X, X) \trianglelefteq p(X, Y)$, and $p(X, Y) \trianglelefteq p(X, X)$.

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

In the presence of an infinite alphabet $\trianglelefteq$ is obviously not a wqo, (take, e.g, $0, 1, 2, \ldots$ where we then have $i \ntrianglelefteq j$ for $i \neq j$).

Example 2. Let us reconsider the sequence from Ex. 1:

    $rev([a, b|T], [\,], R)$, $rev([b|T], [a], R)$, $rev(T, [b, a], R)$, $rev(T', [H', b, a], R)$, $\ldots$

The sequence consisting of the first three elements is admissible wrt $W_{\trianglelefteq}$, while the sequence consisting of the first four elements is not because $rev(T, [b, a], R) \trianglelefteq rev(T', [H', b, a], R)$. Hence, we have obtained the same power as the wfo in Ex. 1, without having to chose which arguments to measure and how to measure them. We further elaborate on the inherent flexibility of $\trianglelefteq$ in the next section.

Also, $\trianglelefteq$ seems to have the desired property that often only "real" loops are detected and that they are detected at the earliest possible moment (see [58]).

---

[2] The full homeomorphic embedding makes use of an underlying wqo $\leq_F$ over the function symbols. The pure homeomorphic embedding uses equality for $\leq_F$, which is only a wqo for finite alphabets.

$\trianglelefteq$ also pinpoints where the dangerous growth has occurred; information which is vital for generalisation and restarting [74, 52, 53, 43].

Looking at Def. 8 one might think that the complexity of checking $s \trianglelefteq t$ for two expressions s and t is exponential. However, the complexity is actually linear [77, 31] (see also [58]); more precisely it is proportional to the size of s and t and to the maximum arity used within s and t.

History $\trianglelefteq$ was first defined over strings by Higman [33] and later extended by Kruskal [41] to ordered trees (and thus symbolic expressions). Since then, $\trianglelefteq$ has been used for many applications. Arguably, the heaviest use of $\trianglelefteq$ within computer science was made in the context of term rewriting systems [15, 16], to automatically derive wfos for static termination analysis. The usefulness of $\trianglelefteq$ as a whistle for partial evaluation was probably first discovered and advocated in [58]. It was then later, independently, rediscovered and adapted for supercompilation by Sørensen and Glück in [74]. Neil Jones played an important role in that re-discovery. Indeed, Neil was tidying his collection of articles and among those articles was a worn copy of [15]. Neil brought this article, containing a definition of homeomorphic embedding, to the attention of Morten Sørensen. Morten immediately realised that this was what he was looking for to ensure termination of supercompilation in an elegant, principled way.

In autumn 1995, Bern Martens and myself were working on the problem of ensuring termination of partial deduction with characteristic trees [26]. Up to then only ad hoc approaches using depth bounds existed. Luckily, Bern Martens visited Neil's group at DIKU in Copenhagen, and came into contact with $\trianglelefteq$. Upon his return, we realised that $\trianglelefteq$ was also exactly the tool we needed to solve our problem, leading to [52, 53]. Later came the realisation that $\trianglelefteq$ provided a mathematically simpler and still more powerful way than wfos of ensuring termination of partial deduction in general, which lead to the development of [47], written during a stay at DIKU.

5 On the Power of Homeomorphic Embedding

It follows from Definitions 5 and 7 that if $\leq$ is a wqo then $<$ (defined by $e_1 < e_2$ iff $e_1 \leq e_2 \wedge e_1 \not\geq e_2$) is a wfo, but not vice versa. However, if $<$ is a well-founded order then $\preceq$, defined by $e_1 \preceq e_2$ iff $e_1 \not> e_2$, is a wbr. Furthermore, $<$ and $\preceq$ 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 wfos. Let us now examine the power of $\trianglelefteq$ in more detail.

Let us examine the power of $\trianglelefteq$ on a few examples. For instance, $\trianglelefteq$ will admit all sequences in Fig. 2 (where, amongst others, Ex. 1 is progressively wrapped into so-called metainterpreters *solve* and *solve'*, counting resolution steps and keeping track of the selected predicates respectively):

Achieving the above is very difficult for wfos 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

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

Fig. 2. Sequences admissible wrt $W_{\unlhd}$

something like the "termsize of the first argument of the first argument of the first argument." For the sequences 6 and 7, things are even more involved. We will return to the particular issue of metaprogramming in more detail in Sect. 8.

The above examples highlight the flexibility of $\unlhd$ compared to wfos. But can one prove some "hard" results? It turns out that one *can* establish that — in the online setting — $\unlhd$ is strictly more generous than a large class of refined wfos.

Monotonic wfos and Simplification orderings [47] establishes that $\unlhd$ is strictly more powerful than the class of so-called *monotonic* wfos. In essence, a monotonic wfo $\succ$ has the property that if it considers the sequence s_1, s_2 to be inadmissible (i.e., $s_1 \not\succ s_2$) then it will also reject sequences such as $s_1, f(s_2)$ and $f(s_1), f(s_2)$. This is a quite natural requirement, which actually most wfos used in online practice satisfy. For instance, the wfo induced by the termsize norm is monotonic. Also, any linear norm induces a monotonic wfo [47]. Almost all of the refined wfos defined in [11, 61, 60, 59] are monotonic.[3]

Formally, monotonic wfos are defined as follows:

Definition 9. *A well-founded order $\prec$ on expressions is said to be* monotonic *iff the following rules hold:*

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

Note the similarity of structure with the definition of $\unlhd$ (but, contrary to $\unlhd$, $\not\succ$ does not have to be the least relation satisfying the rules), which means that $s \unlhd t \Rightarrow s \not\succ t$.

Another interesting class of wfos are the so called simplification orderings, which we adapt from term rewriting systems (to cater for variables). It will turn out that the power of this class is also subsumed by $\unlhd$.

Definition 10. *A* simplification ordering *is a wfo $\prec$ which satisfies*

---

[3] The only non-monotonic wfo in that collection of articles is the one devised for metainterpreters in Definition 3.4 of [11] (also in Section 8.6 of [59]) which allows to focus on subterms. We return to this approach below.

1. $s \prec t \Rightarrow f(t_1, \ldots, s, \ldots, t_n) \prec f(t_1, \ldots, t, \ldots, t_n)$ (replacement property),

2. $s \prec f(t_1, \ldots, s, \ldots, t_n)$ (subterm property) *and*

3. $s \prec t \Rightarrow s\sigma \prec t\gamma$ *for all variable only substitutions* σ *and* γ (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 of powerful wfos are simplification orderings [15, 64]: recursive path ordering, Knuth-Bendix ordering or lexicographic path ordering, to name just a few.

Observe that Def. 10 is also very similar to Def. 8 of $\unlhd$. Indeed, for variable-free expressions we have that $s \unlhd t$ implies $s \preceq t$ for all $\prec$ satisfying Def. 10 (by the Embedding Lemma from [14]). The following theorem is established in [47]. Transitivity of $\prec$ is required in the proof and Theorem 1 does not hold for well-founded relations. For simplification orderings on variable-free expressions, this theorem is a direct consequence of the Embedding Lemma from [14].

Theorem 1. *Let* $\prec$ *be a wfo on expressions which is either monotonic or a simplification ordering (or both). Then any admissible sequence wrt* $\prec$ *is also admissible wrt* $\unlhd$.

This theorem implies that, no matter how much refinement we put into an approach based upon monotonic wfos or upon simplification orderings, we can only expect to approach $\unlhd$ 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 $\unlhd$ as $f(a) \ntrianglelefteq f(b)$, $f(a) \ntrianglelefteq b$, $f(a) \ntrianglelefteq a$, $f(b) \ntrianglelefteq b$, $f(b) \ntrianglelefteq a$ and $a \ntrianglelefteq b$. However, there is no monotonic wfo $\prec$ which admits this sequence. More precisely, to admit δ we must have $f(a) \succ f(b)$ as well as $b \succ a$, i.e. $a \nsucc b$. Hence $\prec$ cannot be monotonic. This also violates rule 1 of Def. 10 and $\prec$ cannot be a simplification ordering.

Non-Monotonic Wfos There are natural wfos which are neither simplification orderings nor monotonic. For such wfos, there can be sequences which are not admissible wrt $W_{\unlhd}$ but which are admissible wrt the wfo. Indeed, $\unlhd$ takes the whole term structure into account while wfos in general can ignore part of the term structure. For example, the sequence $[1, 2], [[1, 2]]$ containing two expressions, is admissible wrt the "listlength" measure but not wrt $\unlhd$, where "listlength" measures a term as 0 if it is not a list and by the number of elements in the list if it is a list [60]. For that same reason the wfos for metainterpreters defined in Definition 3.4 of [11] are not monotonic, as they can focus on certain subterms, fully ignoring other subterms. It will require further work to automate that approach and to compare it with wqo-based approaches, both in theory and in practice.

Still there are some feats of $\unlhd$ 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 $\unlhd$ but there

exists *no* wfo which will admit *both* these sequences. By using a dynamic adjustment of wfos [11] it is possible to admit both sequences. However, for more complicated examples (e.g., from Fig. 2) the dynamic adjustment has to be very refined and one runs into the problem that infinitely many dynamic refinements might exist [60, 59], and to our knowledge no satisfactory solutions exists as of yet.

Finally, the above example also illustrates why, when using a wqo, one has to compare with every predecessor state of a process, whereas when using a wfo one has to compare only to the last predecessor. Hence, the online use of wqo is inherently more complex than the use of wfos.

6 Homeomorphic Embedding in Practice

In this section we survey the (now widespread) use of $\trianglelefteq$ for online termination, since [58, 74]. Works explicitly addressing metaprogramming will be discussed in Sect. 8.

Functional programming As already mentioned, the first fully worked out online use of $\trianglelefteq$ was within *supercompilation* [79, 30]. [74, 76] presented, for the first time, a fully formal definition of positive supercompilation, for a first-order functional language with a lazy semantics. $\trianglelefteq$ was applied on expressions with variables and used to guide the generalisation and ensuring the construction of finite (partial) process trees. The approach was then later generalised in [71] to cover negative supercompilation, where negative constraints are propagated. [75] presents a largely language independent framework for proving termination of program transformers, where $\trianglelefteq$ is one of the possible mechanisms.

Recently, [72] uses $\trianglelefteq$ in the context of *generalized partial computation* (GPC) [24] and presents a refined termination criterion. The main idea consists in measuring the distance between the current expression and base cases for recursion (which have to be identifiable). Homeomorphic embedding is then applied to this sequence of distances, which results in a powerful termination condition, whose effectiveness in practice still needs to be evaluated.

Logic Programming First use of $\trianglelefteq$ for *partial deduction* of logic programs occurred in [52, 53]. Here, $\trianglelefteq$ was not only used on selected literals (to ensure termination of the so-called local control), but also on the atoms in the so-called global tree, as well as on characteristic trees. The latter characterise the computational behaviour of atoms to be specialised, which is often a better basis for controlling polyvariance than the syntactic structure of expressions.

In [29, 39, 13] $\trianglelefteq$ (and the generalisation process) was then extended to cope with conjunctions of atoms, to provide a terminating procedure for *conjunctive partial deduction.*

All of the above led to the development of the ECCE partial deduction system, which can achieve both deforestation and tupling [48]. Moreover, the refined

way in which $\trianglelefteq$ ensures termination opened up new application areas, such as *infinite model checking* [54]. For some particular applications, such as coverability analysis of Petri nets, $\trianglelefteq$ is "fully precise," in the sense that it whistles *only* when a real infinite sequence is being constructed. As shown in [51, 50], one then obtains a decision procedure for these problems and one can establish some relatively surprising links with existing model checking algorithms such as the Karp-Miller procedure [40] or more recent techniques such as [18] and [1, 19].

In some cases, although $\trianglelefteq$ is fully precise, the associated generalisation operation of partial deduction is not. Hence, [43] defines a new generalisation operator which extrapolates the growth detected by $\trianglelefteq$. This enables to solve some new problems, such as the conjunctive planning problem for the fluent calculus.

In another line of work, one might use $\trianglelefteq$ as the basis for specialising and transforming *constraint logic programs*. First steps in that direction have been presented in [20, 21], where a wqo for constrained goals is developed and a generic algorithm is developed.

Functional logic programming Functional logic programming [32] extends both logic and functional programming. A lot of work has recently been carried out on partial evaluation of such programs [5, 4, 2, 6, 3], where $\trianglelefteq$ is used to ensure termination. This work has resulted in the INDY partial evaluation system, which has been successfully applied to a wide range of examples.

In another line of work, [42] has adapted $\trianglelefteq$ for constraint-based partial evaluation of functional logic programs.

7 Extensions of the Homemorphic Embedding

While $\trianglelefteq$ has a lot of desirable properties it still suffers from some drawbacks. First, the homeomorphic embedding relation $\trianglelefteq$ as defined in Def. 8 is rather unsophisticated when it comes to variables. In fact, all variables are treated as if they were identical, a practice which is often undesirable (namely when the same variable can appear multiple times within the same expression). Intuitively, $p(X,Y) \trianglelefteq p(X,X)$ could be justified, while $p(X,X) \trianglelefteq p(X,Y)$ can not. Indeed $p(X,X)$ could be seen as representing something like $and(p(X,Y), eq(X,Y))$, which embeds $p(X,Y)$, but not the other way around. Second, $\trianglelefteq$ behaves in quite unexpected ways in the context of generalisation, posing some subtle problems wrt the termination of a generalisation process [53, 46].

Strict Homeomorphic Embedding $\trianglelefteq^+$ To remedy these problems, [53] introduced the so called strict homeomorphic embedding, which was then taken up, e.g., by [29, 42, 13]. The definition is as follows:

Definition 11. *Let A, B be expressions. Then B (strictly homeomorphically) embeds A, written as $A \trianglelefteq^+ B$, iff $A \trianglelefteq B$ and A is not a strict instance of B.*[4]

---

[4] A is a strict instance of B iff there exists a substitution γ such that $A = B\gamma$ and there exists no substitution σ such that $B = A\sigma$.

Example 4. We now still have that $p(X, Y) \trianglelefteq^+ p(X, X)$ but not $p(X, X) \trianglelefteq^+ p(X, Y)$. Note that still $X \trianglelefteq^+ Y$ and $X \trianglelefteq^+ X$.

The following is proven in [53].

Theorem 2. *The relation $\trianglelefteq^+$ is a wbr on the set of expressions over a finite alphabet.*

Unfortunately, $\trianglelefteq^+$ is not a wqo as it is not transitive. For example, we have
- $p(X, X, Y, Y) \trianglelefteq^+ p(X, Z, Z, X)$ and $p(X, Z, Z, X) \trianglelefteq^+ p(X, X, Y, Z)$
- but $p(X, X, Y, Y) \not\trianglelefteq^+ p(X, X, Y, Z)$.

One might still feel dissatisfied with $\trianglelefteq^+$ for another reason. Indeed, although going from $p(X)$ to $p(f(X))$ looks very dangerous, a transition from $p(X, Y)$ to $p(X, X)$ is actually not dangerous, as there are only finitely many new variable links that can be created. To remedy this, [46] develops the following refinement of $\trianglelefteq^+$, which is useful, for example, in the context of Datalog programs (logic programs who operate on constants only).

Definition 12. *We define $s \trianglelefteq_{var} t$ iff $s \vartriangleleft t$ or s is a variant of t.*

It is obvious that $\trianglelefteq_{var}$ is strictly more powerful than $\trianglelefteq^+$ (if t is strictly more general than s, then it is not a variant of s and it is also not possible to have $s \vartriangleleft t$). For example, we have $p(X) \trianglelefteq_{var} p(f(X))$ as well as $p(X, Y) \trianglelefteq_{var} p(Z, X)$ but $p(X, X) \not\trianglelefteq_{var} p(X, Y)$ and $p(X, Y) \not\trianglelefteq_{var} p(X, X)$.

Theorem 3. *The relation $\trianglelefteq_{var}$ is a wqo on the set of expression over a finite alphabet.*

The Extended Homeomorphic Embedding $\trianglelefteq^*$ Although $\trianglelefteq^+$ has a more refined treatment of variables than $\trianglelefteq$, it is still somewhat unsatisfactory. One point is the restriction to a finite alphabet. Indeed, for a lot of practical programs, using, e.g., arithmetic built-ins, a finite alphabet is no longer sufficient. Luckily, the fully general definition of homeomorphic embedding [41, 16] remedies this aspect. It also allows function symbols with variable arity (which can also be seen as associative operators). We will show below how this definition can be adapted to incorporate a more refined treatment of variables.

However, there is another unsatisfactory aspect of $\trianglelefteq^+$ (and $\trianglelefteq_{var}$). Indeed, we have $p(X, X) \not\trianglelefteq^+ p(X, Y)$ and $p(X, X) \trianglelefteq p(X, Y)$ as expected, but we have, e.g., $f(a, p(X, X)) \trianglelefteq^+ f(f(a), p(X, Y))$, which is rather unexpected. In other words, the more refined treatment of variables is only performed at the root of expressions, but not recursively within the structure of the expressions.

The following, new and more refined embedding relation remedies this somewhat ad hoc aspect of $\trianglelefteq^+$ and adds support for infinite alphabets.

Definition 13. *Given a wbr $\preceq_F$ on the function symbols and a wbr $\preceq_S$ on sequences of expressions, we define the* extended homeomorphic embedding *on expressions by the following rules:*

1. $X \trianglelefteq^* Y$ if X and Y are variables

2. $s \trianglelefteq^* f(t_1, \ldots, t_n)$ if $s \trianglelefteq^* t_i$ for some i

3. $f(s_1, \ldots, s_m) \trianglelefteq^* g(t_1, \ldots, t_n)$ if $f \preceq_F g$ and $\exists 1 \leq i_1 < \ldots < i_m \leq n$ such that $\forall j \in \{1, \ldots, m\} : s_j \trianglelefteq^* t_{i_j}$ and $\langle s_1, \ldots, s_m \rangle \preceq_S \langle t_1, \ldots, t_n \rangle$

Observe that for rule 3 both n and m are allowed to be 0, but we must have $m \leq n$. In contrast to Def. 8 for $\trianglelefteq$, the left- and right-hand terms in rule 3 do not have to be of the same arity. The above rule therefore allows to ignore $n - m$ arguments form the right-hand term (by selecting the m indices $i_1 < \ldots < i_m$).

Furthermore, the left- and right-hand terms in rule 3 do not have to use the same function symbol: the function symbols are therefore compared using the wbr $\preceq_F$. If we have a finite alphabet, then equality is a wqo on the function symbols (one can thus obtain the pure homeomorphic embedding as a special case). In the context of, e.g., program specialisation or analysis, we know that the function symbols occurring within the program (text) and call to be analysed are of finite number. One might call these symbols *static* and all others *dynamic*. A wqo can then be obtained by defining $f \preceq g$ if either f and g are dynamic or if $f = g$. For particular types of symbols a natural wqo or wbr exists (e.g., for numbers) which can be used instead. Also, for associative symbols (such as the conjunction $\wedge$ in logic programming) one can represent $c_1 \wedge \ldots \wedge c_n$ by $\wedge(c_1, \ldots, c_n)$ and then use equality up to arities (e.g., $\wedge/2 = \wedge/3$) for $\preceq_F$.

Example 5. If we take $\preceq_F$ to be equality up to arities and ignore $\preceq_S$ (i.e., define $\preceq_S$ to be always true) we get all the embeddings of $\trianglelefteq$, for example, $p(a) \trianglelefteq^* p(f(a))$. But we also get

 – $p(a) \trianglelefteq^* p(b, f(a), c)$ (while $p(a) \trianglelefteq p(b, f(a), c)$ does not hold),
 – $\wedge(p(a), q(b)) \trianglelefteq^* \wedge(s, p(f(a)), r, q(b))$, and
 – $\wedge(a, b, c) \trianglelefteq^* \wedge(a, b, c, d)$.

One can see that $\trianglelefteq^*$ provides a convenient way to handle associative operators such as the conjunction $\wedge$. (Such a treatment of $\wedge$ has been used in [29, 39, 13] to ensure termination of conjunctive partial deduction.) Indeed, in the context of $\trianglelefteq$ one has to use, e.g., a binary representation. But then whether $\wedge(a, b, c)$ is embedded in $\wedge(a, b, c, d)$ depends on the particular representation, which is not very satisfactory:

 – $\wedge(a, \wedge(b, c)) \trianglelefteq \wedge(a, \wedge(\wedge(b, c), d))$, but
 – $\wedge(a, \wedge(b, c)) \ntrianglelefteq \wedge(\wedge(a, b), \wedge(c, d))$.

In the above definition we can now instantiate $\preceq_S$ such that it performs a more refined treatment of variables, as done for $\trianglelefteq^+$. For example, we can define: $\langle s_1, \ldots, s_m \rangle \preceq_S \langle t_1, \ldots, t_n \rangle$ iff $\langle t_1, \ldots, t_n \rangle$ is not strictly more general than $\langle s_1, \ldots, s_m \rangle$. (Observe that this means that if $m \neq n$ then $\preceq_S$ will hold.) This relation is a wbr (as the strictly more general relation is a wfo [35]). Then, in contrast to $\trianglelefteq^+$ and $\trianglelefteq_{var}$, this refinement will be applied *recursively* within $\trianglelefteq^*$. For example, we now not only have $p(X, X) \ntrianglelefteq^* p(X, Y)$ but also $f(a, p(X, X)) \ntrianglelefteq^* f(f(a), p(X, Y))$ whereas $f(a, p(X, X)) \trianglelefteq^+ f(f(a), p(X, Y))$.

The reason why a recursive "not strict instance" test was not incorporated in [53] (which uses $\trianglelefteq^+$) was that the authors were not sure that this would remain

a wbr (no proof was found yet). In fact, at first sight it looks like recursively applying the "not strict instance" might endanger termination.[5] But the following result, proven in [46], shows that this is not the case:

Theorem 4. $\trianglelefteq^*$ *is a wbr on expressions. Additionally, if $\preceq_F$ and $\preceq_S$ are wqos then so is $\trianglelefteq^*$.*

Other extensions Other extensions, in the context of static termination analysis of term rewriting systems, are proposed in [68] and [45]. Further research is required to determine their usefulness in an online setting.

[28] presents an extension of $\trianglelefteq$ which can handle multiple levels of encodings in the context of metaprogramming. We will examine the issue of metaprogramming in much more detail in the next section.

8 Metaprogramming: Some Results and Open Problems

As we have seen earlier in Sect. 5, $\trianglelefteq$ alone is already very flexible for metainterpreters, even more so when combined with characteristic trees [53] (see also [80]). This section we will study the issue of metaprogramming in more detail, present some new results, and show that some subtle problems still remain. This section is slightly more biased towards partial deduction of logic programs. The discussions should nonetheless be valid for most of the other application areas, namely when the symbolic method is not applied to a system/program directly but to an encoding of it.

Ground versus Non-Ground Representation Let us first discuss one of the main issues in the context of metaprogramming, namely the representation of object-level expressions at the metalevel. In setting with logical variables, there are basically two different approaches to representing an object level expression, say $p(X, a)$, at the metalevel. In the first approach one uses the expression itself as the object level representation. This is called a *non-ground* representation, because it represents an object level variable by a metalevel variable. In the second approach, one uses something like $struct(p, [var(1), struct(a, [])])$ to represent the object level expression. (Usually one does not use the representation $p(var(1), a)$, because then one cannot use the function symbol $var/1$ at the object level.) This is called a *ground* representation, as it represents an object level variable by a ground term. Fig. 3 contains some further examples of the particular ground representation which we will use in this section. For a more detailed discussion we refer the reader to [34, 8].

Of course, one is not restricted to just one level of metainterpretation; one can have a whole hierarchy of metainterpretation [27] where each layer adds its own functionality.

---

[5] If we slightly strengthen point 3 of Def. 13 by requiring that $\langle s_1, \ldots, s_m \rangle$ is not a strict instance of the selected subsequence $\langle t_{i_1}, \ldots, t_{i_m} \rangle$, we actually no longer have a wbr [46].

Object level	Ground representation
X	$var(1)$
c	$struct(c, [\,])$
$f(X, a)$	$struct(f, [var(1), struct(a, [\,])])$

Fig. 3. A ground representation

In this section we want to study the relationship between admissible sequences at the object and metalevel. The simplest setting is when one just uses metainterpreters which mimic the underlying execution and do not add any functionality (in functional programming, such interpreters are often called self-interpreters [37]). Now, the first question that comes to mind is: "If a sequence of evaluation steps at the object level is admissible wrt some well-quasi order, what about the corresponding sequence of evaluation steps at the meta level ?" Ideally, one would want a well-quasi order which is powerful enough to also admit the sequence at the metalevel. In the context of partial deduction this would ensure that if an object program and query can be fully unfolded then the same holds at the metalevel, no matter how many layers of interpretation we put on top of each other. Unfortunately, as we will see below, finding such a wqo turns out to be a daunting task.

8.1 The Representation Problem

In this subsection we will concentrate on the difficulties arising from the fact that object level expressions have to be represented in a different (and possibly more complex) manner at the metalevel. We will ignore for the moment that sequences of expressions at the metalevel might actually be even more involved (i.e., there might be intermediate expressions which have no counterpart at the object level; or a sequence at the object level might correspond to multiple sequences at the metalevel). We will return to some of these issues in Sect. 8.2 below.

To abstract from the number of layers of metainterpretation and the particular representation employed at each layer, we define a function $enc(.)$ which maps object level expressions to corresponding metalevel expressions. For example, if we just use the logic programming, (non-ground) vanilla metainterpreter [34, 8] depicted in Fig. 4 we have $enc(p(X)) = solve(p(X))$. If we use two nested vanilla metainterpreters we will get $enc(p(X)) = solve(solve(p(X)))$. More involved metainterpreters might actually have additional arguments, such as a debugging trace. For the ground representation of Fig. 3 we will get $enc(p(X)) = solve(struct(p, [var(1)]), CAS)$, where CAS is an output variable for the computed answer substitution, which has to be returned explicitly.

We say that a wfo or wbr is *invariant under a particular encoding* $enc(.)$ if whenever it admits a sequence of expressions $o_1, o_2, \ldots, o_n$ then it also admits $enc(o_1), enc(o_2), \ldots, enc(o_n)$, and vice-versa. Solving the representation problem then amounts to finding an adequate wfo or wbr which is invariant under a

$$solve(true) \leftarrow$$
$$solve(A \& B) \leftarrow solve(A) \wedge solve(B)$$
$$solve(H) \leftarrow clause(H, B) \wedge solve(B)$$

Fig. 4. The vanilla metainterpreter

given encoding and still powerful enough (obviously the total relation $\preceq_\top$ with $\forall s, t : s \preceq_\top t$ is a wqo which is invariant under any encoding $enc(.)$).

We now show that $\trianglelefteq$ solves the representation problem in the context of the vanilla metainterpreter of Fig. 4. In the following we use $f^n(t)$ as a shorthand for the expression $\underbrace{f(\dots(f(t)\dots)}_{n}$.

Proposition 2. *Let* $o_1, o_2, \dots, o_n$ *be a sequence of expressions.* $o_1, o_2, \dots, o_n$ *is admissible wrt* $\trianglelefteq$ *iff* $solve^n(o_1), solve^n(o_2), \dots, solve^n(o_n)$ *is admissible wrt* $\trianglelefteq$.

Proof. It is sufficient to show that $o_i \trianglelefteq o_j$ iff $solve(o_i) \trianglelefteq solve(o_j)$. Obviously $o_i \trianglelefteq o_j$ implies $solve(o_i) \trianglelefteq solve(o_j)$ by applying the coupling rule 3 of Def. 8. Now suppose that $solve(o_i) \trianglelefteq solve(o_j)$. Either rule 3 of Def. 8 was applied and we can conclude that $o_i \trianglelefteq o_j$. Or the diving rule 2 was applied. This means that $solve(o_i) \trianglelefteq o_j$. At that point it is possible that the diving rule 2 was further applied, but sooner or later rule 1 or 3 must be applied and we can then conclude that $o_i \trianglelefteq t$ for some subterm t of o_j. Hence we know that $o_i \trianglelefteq o_j$. □

The above result also holds for more involved vanilla-like encodings with extra-arguments provided that within every sequence under consideration an extra-argument does not use the *solve* function symbol and that it embeds all the earlier ones. This holds, e.g., for extra-arguments which contain constants, increasing counters, or histories. For example, the sequence p, q is admissible and so are

- $solve(p, 0), solve(q, s(0))$ (where the interpreter counts evaluation steps) and
- $solve(p, [\,]), solve(q, [p])$ (where the interpreter keeps track of the evaluation history).

However, if the extra argument does not necessarily embed the earlier ones, then more sequences might be admissible at the metalevel. For example, if we add to Fig. 4 a resolution counter, counting downwards, we have that $p(a), p(f(a))$ is not admissible while $solve(p(a), s(0)), solve(p(f(a)), 0)$ is. Alternatively, if the extra argument can contain the *solve* symbol then we can have sequences admissible at the object level but not at the meta level: p, q is admissible while $solve(p, 0), solve(q, solve(p, 0))$ is not.

One might hope that a similar invariance property holds for a ground representation. Unfortunately, this is not the case due to several problems, which we examine below.

Multiple arity If the same predicate symbol can occur with multiple arity, then the following problem can arise. Take the expressions $p(X)$ and $p(X, X)$. We

have that $p(X) \ntrianglelefteq p(X,X)$ ($\trianglelefteq$ inherently treats $p/1$ and $p/2$ as different symbols and the coupling rule 3 of Def. 8 cannot be applied). However, for the ground representation we have $struct(p,[X]) \trianglelefteq struct(p,[X,X])$ because all arguments of p/n are put into a single argument of $struct$ containing a list of length n.

A simple solution to this problem is to use a predicate symbol only with a single arity. Another one is to use $\trianglelefteq^*$ (instead of $\trianglelefteq$) with an underlying identity of function symbols up to arities. A third solution is to add the arity as an extra argument in the ground representation. For the above example, we then obtain $struct(p,1,[X]) \ntrianglelefteq struct(p,2,[X,X])$. This approach also solves more contrived examples such as $p(X) \ntrianglelefteq p(X,f(X))$, where we then have $struct(p,1,[X]) \ntrianglelefteq struct(p,2,[X,struct(f,1,[X])])$.

Variable encoding If we represent variables as integers of the form $\tau = 0 \mid s(\tau)$, we can have that $X \trianglelefteq Y$ while $var(s(0)) \ntrianglelefteq var(0)$. (In that case $\trianglelefteq$ on the encoding is actually more admissible.) One solution is to use a different function symbol for each distinct variable (meaning we have an infinite alphabet) and then use $\trianglelefteq^*$ with an underlying wqo on these new function symbols, e.g., either treating all encodings of variables as one fresh constant or even 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
$$f(a,b) \ntrianglelefteq f(g(a,b),c)$$
while
$$struct(f,[\text{``}a\text{''},\text{``}b\text{''}]) \trianglelefteq struct(f,[struct(g,[\text{``}a\text{''},\text{``}b\text{''}]),\text{``}c\text{''}])$$
where we have used "c" to denote the ground representation $struct(c,[\,])$ of a constant symbol c.

The reason for this odd behaviour is that the coupling rule 3 of Def. 8 checks whether the term 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 might think that one possible solution to this problem would be to adapt $\trianglelefteq$ such that one is allowed to examine the *same* argument multiple times for embedding. In other words one would define a relation $\trianglelefteq^-$ by adapting rule 3 of Def. 8 to (and keeping rules 1 and 2):

3. $f(s_1,\ldots,s_m) \trianglelefteq^- f(t_1,\ldots,t_n)$ if $\exists 1 \leq i_1 \leq \ldots i_m \leq n$ such that $\forall j \in \{1,\ldots,m\} : s_j \trianglelefteq^- t_{i_j}$.

Note that, contrary to $\trianglelefteq$, i_j can be equal to i_{j+1} and we now have $f(a,b) \trianglelefteq^- f(g(a,b),c)$. Unfortunately, this solution is not invariant under the ground representation either. A counterexample is as follows: $f(a) \ntrianglelefteq^- g(f(b),g(a))$ while $struct(f,[\text{``}a\text{''}]) \trianglelefteq^- struct(g,[struct(f,[\text{``}b\text{''}]),struct(g,[\text{``}a\text{''}])])$.

So, despite the usefulness of $\trianglelefteq$ for metaprogramming exhibited earlier in the paper, there still remains the open problem: Can we find a strengthening or useful adaptation of $\trianglelefteq$ which is invariant under the ground representation? It might be possible to achieve this by using (a refinement of) [68], which extends Kruskal's theorem. A pragmatic solution would of course be to simple decode data as much as possible in, e.g., the program specialiser, and then apply $\trianglelefteq$ (or $\trianglelefteq^*$) on the de-encoded data only. This, however, requires knowledge about the particular encodings that are likely to appear. A more refined and promising approach for handling multiple levels of encodings within $\trianglelefteq$ is presented in [28].

8.2 The Parsing Problem

In the context of metaprogramming. we also encounter the so-called *parsing problem* [59]. Below we provide another view of the parsing problem, in terms of invariance under representation.

In partial deduction of logic programs, nobody has found it useful to compare complete goals, only the *selected* atoms within the goals are compared. Also, it was quickly realised that it was difficult to define an order relation on the full sequence that was giving good results and that it was sufficient and easier to do so on certain subsequences containing the so-called covering ancestors [11].

Take, for example, the program P consisting of the two clauses:

$$p(f(X)) \leftarrow p(b) \wedge p(f(X))$$
$$p(a) \leftarrow$$

Let us unfold the goal $\leftarrow p(f(Y))$ by using $\trianglelefteq$ on the selected atoms (selected atoms are underlined):

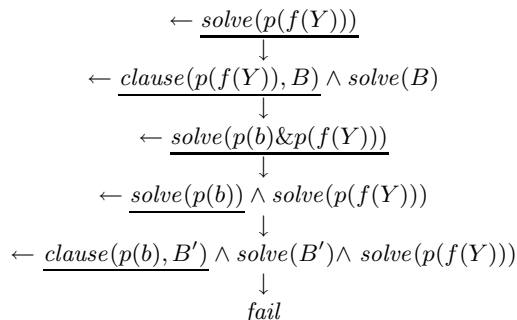$$\leftarrow \underline{p(f(Y))}$$
$$\downarrow$$
$$\leftarrow \underline{p(b)} \wedge p(f(Y))$$
$$\downarrow$$
$$fail$$

The covering ancestor sequence for $p(b)$ is $p(f(Y)), p(b)$ which is admissible wrt $\trianglelefteq$. We were thus successful in fully unfolding $\leftarrow p(f(Y))$ and detecting finite failure. If we had looked at the entire goals, we would not have been able to fully unfold $\leftarrow p(f(Y))$, because $p(f(Y)) \trianglelefteq p(b) \wedge p(f(Y))$.

Let us examine how this refinement fares in the context of metaprogramming. For this we take the standard vanilla metainterpreter of Fig. 4, together with the following encoding of the above program P:

$$clause(p(f(X)), (p(b)\&p(f(X)))) \leftarrow$$
$$clause(p(a), true) \leftarrow$$

One would hope that, by using $\trianglelefteq$ on the selected atoms and comparing with the covering ancestors, it would be possible to fully unfold $\leftarrow solve(p(f(Y)))$ in a similar manner as for $\leftarrow p(f(Y))$ above. Let us examine the sequence of goals, needed to detect finite failure:

$$\leftarrow \underline{solve(p(f(Y)))}$$
$$\downarrow$$
$$\leftarrow \underline{clause(p(f(Y)), B)} \land solve(B)$$
$$\downarrow$$
$$\leftarrow \underline{solve(p(b)\&p(f(Y)))}$$
$$\downarrow$$
$$\leftarrow \underline{solve(p(b))} \land solve(p(f(Y)))$$
$$\downarrow$$
$$\leftarrow \underline{clause(p(b), B')} \land solve(B') \land solve(p(f(Y)))$$
$$\downarrow$$
$$fail$$

Unfortunately, even if we ignore the intermediate goals containing *clause* atoms (they have no counterpart at the object level) we have a problem: at the third step we select $solve(p(b)\&p(f(Y)))$ who has the covering ancestor $solve(p(f(Y)))$ with $solve(p(f(Y))) \trianglelefteq solve(p(b)\&p(f(Y)))$. The same embedding holds for $\trianglelefteq^+$ or $\trianglelefteq^*$. We are thus unable to fully unfold $\leftarrow solve(p(f(Y)))$. The problem is that, in the metainterpreter, multiple atoms can be put together in a single term, and the refinement of looking only at the selected atoms and their covering ancestors is not even invariant under the non-ground representation!

Solving this problem in a general manner is a non-trivial task: one would have to know that *solve* will eventually decompose $p(b)\&p(f(Y))$ into its constituents $p(b)$ and $p(f(Y))$ giving us the opportunity to continue with $solve(p(b))$ while stopping the unfolding of $solve(p(f(Y)))$. [80] presents a solution to this problem for the particular vanilla metainterpreter above, but unfortunately it does not scale up to other, more involved metainterpreters.

9   Discussion and Conclusion

Critical Evaluation and Future Work In theory, existing online systems, such as INDY and ECCE, based on $\trianglelefteq$, ensure termination in a fully automatic manner and can thus be used even by a naïve user. However, for more involved tasks, these systems can lead to substantial code explosion, meaning that some user expertise is still required to prevent such cases. Also, these systems might fail to provide a specialised program that is more efficient in practice, because existing control techniques fail to take certain pragmatic issues into account [49]. Indeed, although $\trianglelefteq$ has proven to be extremely useful superimposed, e.g., on determinate unfolding, on its own it will sometimes allow too much unfolding than desirable for efficiency concerns: more unfolding does not always imply a better specialised program and it can also negatively affect the efficiency of the specialisation process itself. Moreover, $\trianglelefteq$ can be used to fully evaluate the non primitive recursive Ackerman function (this is a corollary of Theorem 1; see also [58, p. 186–187]). Hence, $\trianglelefteq$ on its own can lead to a worst case complexity for the transformation/specialisation process which is not primitive recursive. Although cases of bad complexity seem to be relatively rare in practice, this still means that $\trianglelefteq$ should better not be used as is in a context (such as within a

compiler) where a tight upper-bound on memory and time requirements is essential. However, we hope that it is going to be possible to engineer an efficient approximation of $\trianglelefteq$ (or $\trianglelefteq^*$), which takes more pragmatic issues into account.

On the other hand, for some applications, $\trianglelefteq$ as well as $\trianglelefteq^+$ and $\trianglelefteq^*$ remain too restrictive. As we have discussed in Sect. 8, they do not always perform satisfactorily in the context of arbitrary metainterpretation tasks. Only future research can tell whether one can solve these problems, while not (substantially) deteriorating the complexity. A completely different approach to termination has very recently been presented in [55]. It will be interesting to see how it compares to $\trianglelefteq$ and its derivatives.

Conclusion In summary, we have shed new light on the relation between wqos and wfos and have formally shown why wqos are more interesting, at least in theory, than wfos for ensuring termination in an online setting. We have illustrated the inherent flexibility of $\trianglelefteq$ and shown that, despite its simplicity, it is strictly more generous than a large class of wfos. We have surveyed the usage of $\trianglelefteq$ in existing techniques, and have touched upon some new application areas such as infinite model checking.

We have also discussed extensions of $\trianglelefteq$, which inherit all the good properties of $\trianglelefteq$ while providing a refined treatment of (logical) variables. We believe that these refinements can be of value in contexts such as partial evaluation of (functional and) logic programs or supercompilation of functional programming languages, where — at specialisation time – variables also appear. One can also simply plug $\trianglelefteq^*$ into the language-independent framework of [73]. We also believe that $\trianglelefteq^*$ provides both a theoretically and practically more satisfactory basis than $\trianglelefteq^+$ or $\trianglelefteq$.

Finally, we have discussed the use of $\trianglelefteq$ in the context of metaprogramming, proving some positive results for the non-ground representation, but also some negative results for both the ground and non-ground representation.

In summary, $\trianglelefteq$ is an elegant and very powerful tool to ensure online termination, but a lot of research is still needed to make it efficient enough for full practical use and powerful enough to cope with arbitrary metalevel encodings.

Acknowledgements

References

1. P. A. Abdulla, K. Čerāns, B. Jonsson, and Y.-K. Tsay. General decidability theorems for infinite-state systems. In *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science*, pages 313–321, New Brunswick, New Jersey, 27–30 July 1996. IEEE Computer Society Press.
2. 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.
3. E. Albert, M. Alpuente, M. Hanus, and G. Vidal. A partial evaluation framework for curry programs. In *Proc. of the 6th International Conference on Logic for Programming and Automated Reasoning (LPAR'99)*, LNCS 1705, pages 376–395. Springer-Verlag, 1999.
4. M. Alpuente, M. Falaschi, P. Julián, and G. Vidal. Spezialisation of lazy functional logic programs. In *Proceedings of PEPM'97, the ACM Sigplan Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 151–162, Amsterdam, The Netherlands, 1997. ACM Press.
5. 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.
6. M. Alpuente, M. Falaschi, and G. Vidal. Partial Evaluation of Functional Logic Programs. *ACM Transactions on Programming Languages and Systems*, 20(4):768–844, 1998.
7. 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.
8. K. R. Apt and F. Turini. *Meta-logics and Logic Programming*. MIT Press, 1995.
9. K. Benkerimi and J. W. Lloyd. A partial evaluation procedure for logic programs. In S. Debray and M. Hermenegildo, editors, *Proceedings of the North American Conference on Logic Programming*, pages 343–358. MIT Press, 1990.
10. R. Bol. Loop checking in partial deduction. *The Journal of Logic Programming*, 16(1&2):25–46, 1993.
11. 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.
12. 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.
13. D. De Schreye, R. Glück, J. Jørgensen, M. Leuschel, B. Martens, and M. H. Sørensen. Conjunctive partial deduction: Foundations, control, algorithms and experiments. *The Journal of Logic Programming*, 41(2 & 3):231–277, November 1999.
14. N. Dershowitz. Orderings for term-rewriting systems. *Theoretical Computer Science*, 17(3):279–301, Mar. 1982.
15. N. Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, 3:69–116, 1987.
16. 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.

17. N. Dershowitz and Z. Manna. Proving termination with multiset orderings. *Communications of the ACM*, 22(8):465–476, 1979.

18. A. Finkel. The minimal coverability graph for Petri nets. *Lecture Notes in Computer Science*, 674:210–243, 1993.

19. A. Finkel and P. Schnoebelen. Fundamental structures in well-structured infinite transition systems. In *Proceedings of LATIN'98*, LNCS 1380, pages 102–118. Springer-Verlag, 1998.

20. F. Fioravanti, A. Pettorossi, and M. Proietti. Rules and strategies for contextual specialization of constraint logic programs. *Electronic Notes in Theoretical Computer Science*, 30(2), December 1999.

21. F. Fioravanti, A. Pettorossi, and M. Proietti. Automated strategies for specializing constraint logic programs. In *Logic Based Program Synthesis and Transformation. Proceedings of Lopstr'2000*, LNCS 1207, pages 125–146, 2000.

22. H. Fujita and K. Furukawa. A self-applicable partial evaluator and its use in incremental compilation. *New Generation Computing*, 6(2 & 3):91–118, 1988.

23. D. A. Fuller and S. Abramsky. Mixed computation of Prolog programs. *New Generation Computing*, 6(2 & 3):119–141, June 1988.

24. Y. Futamura, K. Nogi, and A. Takano. Essence of generalized partial computation. *Theoretical Computer Science*, 90(1):61–79, 1991.

25. 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.

26. J. Gallagher and M. Bruynooghe. The derivation of an algorithm for program specialisation. *New Generation Computing*, 9(3 & 4):305–333, 1991.

27. R. Glück. On the mechanics of metasystem hierarchies in program transformation. In M. Proietti, editor, Logic Program Synthesis and Transformation. *Proceedings of LOPSTR'95*, LNCS 1048, pages 234–251, Utrecht, The Netherlands, September 1995. Springer-Verlag.

28. R. Glück, J. Hatcliff, and J. Jørgensen. Generalization in hierarchies of online program specialization systems. In P. Flener, editor, Logic-Based Program Synthesis and Transformation. *Proceedings of LOPSTR'98*, LNCS 1559, pages 179–198, Manchester, UK, June 1998. Springer-Verlag.

29. 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.

30. R. Glück and M. H. Sørensen. A roadmap to supercompilation. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation, International Seminar*, LNCS 1110, pages 137–160, Schloß Dagstuhl, 1996. Springer-Verlag.

31. J. Gustedt. *Algorithmic Aspects of Ordered Structures*. PhD thesis, Technische Universität Berlin, 1992.

32. M. Hanus. The integration of functions into logic programming. *The Journal of Logic Programming*, 19 & 20:583–628, May 1994.

33. G. Higman. Ordering by divisibility in abstract algebras. *Proceedings of the London Mathematical Society*, 2:326–336, 1952.

34. 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.

35. G. Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *Journal of the ACM*, 27(4):797–821, 1980.

36. N. D. Jones. An introduction to partial evaluation. *ACM Computing Surveys*, 28(3):480–503, September 1996.

37. N. D. Jones. *Computability and Complexity: From a Programming Perspective*. MIT Press, 1997.

38. N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.

39. J. Jørgensen, M. Leuschel, and B. Martens. Conjunctive partial deduction in practice. In J. Gallagher, editor, *Logic Program Synthesis and Transformation. Proceedings of LOPSTR'96*, LNCS 1207, pages 59–82, Stockholm, Sweden, August 1996. Springer-Verlag.

40. R. M. Karp and R. E. Miller. Parallel program schemata. *Journal of Computer and System Sciences*, 3:147–195, 1969.

41. J. B. Kruskal. Well-quasi ordering, the tree theorem, and Vazsonyi's conjecture. *Transactions of the American Mathematical Society*, 95:210–225, 1960.

42. L. Lafave and J. Gallagher. Constraint-based partial evaluation of rewriting-based functional logic programs. In N. Fuchs, editor, *Logic Program Synthesis and Transformation. Proceedings of LOPSTR'97*, LNCS 1463, pages 168–188, Leuven, Belgium, July 1997.

43. H. Lehmann and M. Leuschel. Solving planning problems by partial deduction. In M. Parigot and A. Voronkov, editors, *Proceedings of the International Conference on Logic for Programming and Automated Reasoning (LPAR'2000)*, LNAI 1955, pages 451–468, Reunion Island, France, 2000. Springer-Verlag.

44. 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.

45. P. Lescanne. Well rewrite orderings and well quasi-orderings. Technical Report N° 1385, INRIA-Lorraine, France, January 1991.

46. M. Leuschel. Improving homeomorphic embedding for online termination. In P. Flener, editor, Logic-Based Program Synthesis and Transformation. *Proceedings of LOPSTR'98*, LNCS 1559, pages 199–218, Manchester, UK, June 1998. Springer-Verlag.

47. 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.

48. M. Leuschel. Logic program specialisation. In J. Hatcliff, T. Æ. Mogensen, and P. Thiemann, editors, *Partial Evaluation: Practice and Theory*, LNCS 1706, pages 155–188, Copenhagen, Denmark, 1999. Springer-Verlag.

49. M. Leuschel and M. Bruynooghe. Logic program specialisation through partial deduction: Control issues. *Theory and Practice of Logic Programming*, 2(4 & 5):461–515, July & September 2002.

50. M. Leuschel and H. Lehmann. Coverability of reset Petri nets and other well-structured transition systems by partial deduction. In J. Lloyd, editor, *Proceedings of the International Conference on Computational Logic (CL'2000)*, LNAI 1861, pages 101–115, London, UK, 2000. Springer-Verlag.

51. M. Leuschel and H. Lehmann. Solving coverability problems of Petri nets by partial deduction. In M. Gabbrielli and F. Pfenning, editors, *Proceedings of PPDP'2000*, pages 268–279, Montreal, Canada, 2000. ACM Press.

52. 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, *Partial Evaluation, International Seminar*, LNCS 1110, pages 263–283, Schloß Dagstuhl, 1996. Springer-Verlag.

53. 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.

54. M. Leuschel and T. Massart. Infinite state model checking by abstract interpretation and program specialisation. In A. Bossi, editor, Logic-Based Program Synthesis and Transformation. *Proceedings of LOPSTR'99*, LNCS 1817, pages 63–82, Venice, Italy, September 1999.

55. C. S. Lii, N. D. Jones, and A. M. Ben-Amram. The size-change principle for program termination. In *Proceedings of POPL'01*. ACM Press, January 2001.

56. J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987.

57. J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *The Journal of Logic Programming*, 11(3& 4):217–242, 1991.

58. R. Marlet. *Vers une Formalisation de l'Évaluation Partielle*. PhD thesis, Université de Nice - Sophia Antipolis, December 1994.

59. B. Martens. *On the Semantics of Meta-Programming and the Control of Partial Deduction in Logic Programming*. PhD thesis, K.U. Leuven, February 1994.

60. B. Martens and D. De Schreye. Automatic finite unfolding using well-founded measures. *The Journal of Logic Programming*, 28(2):89–146, August 1996.

61. 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.

62. 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.

63. J. Martin and M. Leuschel. Sonic partial deduction. In *Proceedings of the Third International Ershov Conference on Perspectives of System Informatics*, LNCS 1755, pages 101–112, Novosibirsk, Russia, 1999. Springer-Verlag.

64. A. Middeldorp and H. Zantema. Simple termination of rewrite systems. *Theoretical Computer Science*, 175(1):127–158, 1997.

65. T. Mogensen and P. Sestoft. Partial evaluation. In A. Kent and J. G. Williams, editors, *Encyclopedia of Computer Science and Technology*, pages 247–279. Marcel Decker, 270 Madison Avenue, New York, New YOrk 10016, 1997.

66. A. Pettorossi and M. Proietti. Transformation of logic programs: Foundations and techniques. *The Journal of Logic Programming*, 19& 20:261–320, May 1994.

67. A. Pettorossi and M. Proietti. A comparative revisitation of some program transformation techniques. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation, International Seminar*, LNCS 1110, pages 355–385, Schloß Dagstuhl, 1996. Springer-Verlag.

68. L. Puel. Using unavoidable set of trees to generalize Kruskal's theorem. *Journal of Symbolic Computation*, 8:335–382, 1989.

69. E. Ruf. *Topics in Online Partial Evaluation*. PhD thesis, Stanford University, March 1993.

70. D. Sahlin. Mixtus: An automatic partial evaluator for full Prolog. *New Generation Computing*, 12(1):7–51, 1993.

71. J. P. Secher and M. H. Sørensen. On perfect supercompilation. In *Proceedings of the Third International Ershov Conference on Perspectives of System Informatics*, LNCS 1755, pages 113–127, Novosibirsk, Russia, 1999. Springer-Verlag.

72. L. Song and Y. Futamura. A new termination approach for specialization. In W. Taha, editor, *Proceedings of SAIG'00*, LNCS 1924, pages 72–91. Springer-Verlag, 2000.

73. M. H. Sørensen. Convergence of program transformers in the metric space of trees. In *Mathematics of Program Construction, Proceedings of MPC'98*, LNCS 1422, pages 315–337. Springer-Verlag, 1998.

74. 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.

75. M. H. Sørensen and R. Glück. Introduction to supercompilation. In J. Hatcliff, T. Æ. Mogensen, and P. Thiemann, editors, *Partial Evaluation — Practice and Theory*, LNCS 1706, pages 246–270, Copenhagen, Denmark, 1999. Springer-Verlag.

76. M. H. Sørensen, R. Glück, and N. D. Jones. A positive supercompiler. *Journal of Functional Programming*, 6(6):811–838, 1996.

77. J. Stillman. *Computational Problems in Equational Theorem Proving*. PhD thesis, State University of New York at Albany, 1988.

78. A. Takeuchi and K. Furukawa. Partial evaluation of Prolog programs and its application to meta programming. In H.-J. Kugler, editor, *Information Processing 86*, pages 415–420, 1986.

79. V. F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, 1986.

80. W. Vanhoof and B. Martens. To parse or not to parse. In N. Fuchs, editor, *Logic Program Synthesis and Transformation. Proceedings of LOPSTR'97*, LNCS 1463, pages 322–342, Leuven, Belgium, July 1997.

81. A. Weiermann. Complexity bounds for some finite forms of Kruskal's theorem. *Journal of Symbolic Computation*, 18(5):463–488, November 1994.

82. 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.