Type-based Homeomorphic Embedding for Online Termination

Elvira Albert [a] John Gallagher [b,d] Miguel Gómez-Zamalloa [a] Germán Puebla [c]

[a] *DSIC, Complutense University of Madrid, E-28040 Madrid, Spain*
[b] *CBIT, Roskilde University, DK-4000 Roskilde, Denmark*
[c] *CLIP, DLSIIS, Technical University of Madrid, E-28660 Boadilla del Monte, Spain*
[d] *IMDEA Software,E-28660 Boadilla del Monte, Spain*

Abstract

Online termination techniques dynamically guarantee termination of computations by *supervising* them in such a way that computations whose termination can no longer be guaranteed are stopped. *Homeomorphic Embedding* (HEm) has proven to be very useful for online termination provided that the computations supervised are performed over a *finite signature*, i.e., the number of constants and function symbols involved is finite. However, there are many situations, for example numeric computations, which involve an infinite signature and thus HEm does not guarantee termination. Some extensions to HEm for the case of infinite signatures have been proposed which guarantee termination. However, the existing techniques either do not provide systematic means for generating such extensions or the extensions are too simplistic and do not produce the expected results in practice. We propose *Type-based Homeomorphic Embedding* (TbHEm) as an extension of the standard, untyped, HEm. By taking static information about the behavior of the computation into account, expressed as types, TbHEm allows obtaining more precise results than those of the previous extensions to HEm for the case of infinite signatures. We show that the existing extensions to HEm which are currently used in state-of-the-art specialization tools can be reconstructed as instances of TbHEm. We illustrate the applicability of our proposal in a realistic case study: partial evaluation of an interpreter. We argue that the results obtained provide empirical evidence of the interest of our proposal.

Key words: Termination, Well-quasi Orders, Homeomorphic Embedding, Program Transformation, Partial Evaluation

1. Introduction

Guaranteeing termination is a key aspect of areas of computer science which have to deal with possibly infinite computations, namely in all areas of automatic program analysis, synthesis, verification, specialization and transformation. Broadly speaking, guaranteeing termination can be tackled in an *offline* or an *online* fashion. The main difference between these is that in offline termination we aim at statically determining termination. This means that we do not have the concrete values of arguments at each point of the computation but rather just *abstractions* of them. Usually these abstractions refer to the *size* of values under some measure, such as list length, term size, numeric value for natural numbers, etc. In contrast, in online termination, we guarantee termination by *supervising* the computation and stopping it as soon as we can no longer guarantee termination.

The main advantage of the offline approach is that if we can prove termination statically, there is no longer any need to supervise the computation for termination, which results in performance gains. In the offline setting, powerful semi-automated termination proof techniques have been developed in the context of term rewrite systems (TRS), the most popular one being the *recursive path ordering* [6].

On the other hand, the online approach is more precise, since we have the concrete values at hand

and thus we can compare actual values instead of abstractions of values. Thus, the online approach is of interest in applications where precision is of great importance and where the offline approach tends to behave too conservatively in order to guarantee termination. Another advantage of online techniques is that they are usually simpler to implement than offline techniques, which are based on sophisticated static analyses. As a result, which of the two approaches to take greatly depends on the application area. For example, in the context of online supervision of symbolic computations, *well-founded orders* (wfos) [18] and especially *well-quasi orders* (wqos) [4,21] have become widely used.

In order theory, a wqo is a quasi-order with an additional restriction on sequences that ensures that for any infinite sequence $x_1, x_2, \ldots$, there exists $i < j$ with $x_i \leq x_j$. In this article, we focus on the *homeomorphic embedding* (HEm) relation [11,13,14], a wqo used in state-of-the-art online specialization tools. Intuitively, HEm is a structural ordering under which an expression e_2 is greater than or equal to another expression e_1, written as $e_1 \trianglelefteq e_2$, if e_1 can be obtained from e_2 by deleting some parts of e_2. Under these circumstances we say that e_2 *embeds* e_1. E.g., $\underline{\mathtt{s}}(\mathtt{s}(\underline{\mathtt{U}} + \mathtt{W}) \underline{\times} (\underline{\mathtt{U}} \underline{+} \mathtt{s}(\underline{\mathtt{V}})))$ embeds $\mathtt{s}(\mathtt{U} \times (\mathtt{U} + \mathtt{V}))$.

The HEm relation was first defined over strings by Higman [9] and later extended by Kruskal [11] to ordered trees (and thus symbolic expressions). Since then, HEm has been used for many applications. Arguably, the heaviest use of HEm within computer science was made in the context of TRS [7], to automatically derive well-founded orders for static termination analysis. The usefulness of HEm in the context of online partial evaluation was first discovered and advocated by Marlet [17]. It was later, independently, rediscovered and adapted for supercompilation by Sørensen and Glück [23]. Later on, Leuschel and Martens [15,16] demonstrated that HEm provides a mathematically simpler and still more powerful way of ensuring termination of partial deduction than existing wfos and wqos. The latter was then witnessed by Leuschel [12]. A survey on the theory and practice of HEm can be found in [13].

The HEm relation can be used to guarantee termination when computing a sequence $e_1, e_2, \ldots$, by using HEm as a *whistle*. Whenever a new expression e_{n+1} is to be added to a finite sequence $e_1, \ldots, e_n$, we first check whether e_{n+1} embeds any of the expressions already in the sequence. If that is the case, we say that HEm whistles, i.e., it has detected (potential) non-termination and the computation has to be stopped. If HEm does not whistle e_{n+1} can be safely added to the sequence and the computation can proceed without endangering termination.

The reason for the success of HEm as an approach for guaranteeing online termination is twofold. i) It often allows sequences to grow quite large before the whistle blows, to the point that in a good number of finite sequences the full sequence can be computed without the whistle blowing at all. This is essential for instance, in program specialization, as allowing a larger sequence implies further propagation of information and hence, as we will see in the paper, a better specialization can often be obtained. ii) It often identifies redundant computations quickly, and the whistle blows without unnecessarily further expanding the sequence, thus avoiding irrelevant computations. This is also essential in program specialization both for efficiency of the specialization process and for quality of the resulting program.

While HEm has proven to be very useful for symbolic computations (as required by program specialization and analysis techniques, see [12]), some difficulties remain in the presence of infinite signatures, such as the numbers. For instance, the `is/2` Prolog built-in is used to evaluate arithmetic expressions; given two numbers, it can produce as output a number which does not appear in the program text. If this can be infinitely repeated, we need to handle an infinite signature. As further examples, in the case of logic programs, infinite signatures appear as soon as certain built-ins such as `functor/3`, `name/2`, `=../2`, `atom_codes/2`, etc. are used, since they allow creating fresh constants and function symbols. Some extensions to HEm over infinite signatures have been defined and used in practice (e.g. [2,13]), but they are often too *ad hoc*; for instance, they only handle constants which appear explicitly in the program, regardless of which part of the program (function, argument position) they appear. As such approaches are purely *syntactic*, in practice they sometimes turn out to be too conservative ("whistling" too early) or else too aggressive, and thus do not have either of the features i) or ii) above.

In essence, while other works [2,13] take a simple *syntactic* approach to extending the HEm relation, we propose a *semantic* approach for such extension. In particular, we introduce the *type-based homeomorphic embedding* (TbHEm) relation which, by taking information about the behavior of the computation into account, provides more precise results in the presence of infinite signatures. For this, our

typed relation is defined on types structured into a (possibly empty) *finite part* and a (possibly empty) *infinite part*. TbHEm allows expanding sequences as long as the concrete values which appear in the expression remain within the finite part of the type. Note that in computations with a finite signature it is always possible to obtain types whose infinite part is empty, which produces the same effect as the traditional HEm. Intuitively, this allows us to achieve i) and ii) simultaneously as: i) finite sequences are expected to have an empty infinite partition, or non-empty but with a large finite part, and hence they can grow considerably before the whistle blows; ii) infinite sequences will have a non-empty partition which forces the whistle to blow.

HEm has been extensively used for supervising partial evaluation of logic programs. However, it is important to stress that both the HEm and TbHEm relations are of interest for supervising any computation which manipulates acyclic data structures, such as lists (or equivalently, acyclic linked lists), trees, etc. which can grow indefinitely large. This is so regardless of the programming language in which such computation is implemented.

The rest of the article is organized as follows. Section 2 recalls some basic notions and introduces notation. In Section 3, we introduce TbHEm, as a novel extension to untyped HEm, and prove its soundness. Section 5 shows how TbHEm generalizes existing relations used in current systems. In Section 6 we present some experimental results. Finally, Section 7 discusses the practicality of TbHEm in the context of online termination approaches and concludes.

2. Preliminaries and Notation

We recall some preliminary concepts, in particular on the HEm relation, and introduce some notation.

2.1. *Symbolic Expressions*

For the sake of generality we consider the language of *symbolic expressions* (first-order terms). Its *alphabet* consists of the following classes of symbols: 1) *variables* ($\mathcal{V}$) and 2) *function symbols* (Σ). Function symbols have an associated *arity*. *Constants* are function symbols with arity 0. We refer to the set of functions in an alphabet as its *signature*.

Definition 1 (Symbolic Expressions) *The set of symbolic expressions $\mathcal{E}$ over some given alphabet, $\Sigma \cup \mathcal{V}$, is inductively defined as follows:*

(i) a variable $v \in \mathcal{V}$ is an expression,
(ii) a function symbol $f \in \Sigma$ of arity $n \geq 0$ applied to a sequence $e_1, \ldots, e_n$ of expressions, denoted $f(e_1, \ldots, e_n)$, is also an expression.

We will adhere to the following syntactical conventions: Variables are denoted by upper-case letters like $X, Y, Z, \ldots$, constants by lower-case letters like $a, b, c, \ldots$, and non-constant function symbols by lower-case letters like $f, g, h, \ldots$

2.2. *Homeomorphic Embedding*

We now introduce some auxiliary definitions on orders which are required to define the HEm relation.

Definition 2 (Quasi-order) *A* quasi-order *is a reflexive and transitive binary relation on $\mathcal{E}$.*

A *well-quasi order* is a *well-binary relation* which is also a quasi-order, as stated below.

Definition 3 (wbr, wqo) *Let $\leq$ be a binary relation on $\mathcal{E}$. We say that $\leq$ is a* well-binary *relation* (wbr) *iff for any infinite sequence $e_1, e_2, \ldots$ of expressions, $\exists i, j : i < j \ \wedge \ e_i \leq e_j$. If $\leq$ is also a quasi-order then $\leq$ is called a* well-quasi order (wqo).

The next definition recalls the HEm relation on expressions, as presented by Leuschel [12].

Definition 4 (HEm,$\trianglelefteq$) *The homeomorphic embedding relation over expressions, written $\trianglelefteq$, is defined by the following rules:*

(i) $Y \trianglelefteq X$ for all variables X, Y.
(ii) $s \trianglelefteq f(t_1, \ldots, t_n)$ if $s \trianglelefteq t_i$ for some i.
(iii) $f(s_1, \ldots, s_n) \trianglelefteq f(t_1, \ldots, t_n)$ if $s_i \trianglelefteq t_i$ for all i, $1 \leq i \leq n$.

As already discussed, $e_1 \trianglelefteq e_2$ iff e_1 can be obtained from e_2 by removing some symbols. Hence, the structure of e_1, split in parts, reappears within e_2. For finite signatures, HEm is a wqo (see, e.g., [12]).

2.3. *Types*

We adopt the syntax of Mercury [22] for type definitions. The set of *type expressions (types)*, denoted $\mathcal{T}$, is constructed from an infinite set $\mathcal{V}_\mathcal{T}$ of type variables (parameters) and a set $\Sigma_\mathcal{T}$ of type symbols with their associated arities; these are disjoint from the set of variables $\mathcal{V}$ and the signature Σ. Types and symbolic expressions are related by means of *type definitions*.

Definition 5 (type definition) *A* type rule *for a type symbol h with arity n in $\Sigma_\mathcal{T}$ is of one of these two forms:*

$h(\bar{T}) \longrightarrow f_1(\bar{\tau}_1); \ldots; f_k(\bar{\tau}_k)$ $(k \geq 1)$, *or*
$h(\bar{T}) \longrightarrow f_1(\bar{\tau}_1); \ldots$ (*infinite sequence*)

where the following conditions hold:
(i) $\bar{T}$ *is an* n*-tuple of distinct type variables,*
(ii) $f_1, \ldots, f_k, \ldots$ *are distinct function symbols from* Σ*,*
(iii) each $\bar{\tau}_i$ $(i \geq 1)$ *is an* m*-tuple from* $\mathcal{T}$*, where* m *is the arity of the corresponding* f_i*,*
(iv) type variables in the right-hand side, if any, are from $\bar{T}$*.*

We say that $f_1(\bar{\tau}_1); \ldots; f_k(\bar{\tau}_k)$ *or* $f_1(\bar{\tau}_1); \ldots$ *are the, possibly infinite, set of cases of the type.*

A type definition *is a finite set of type rules where no two rules contain the same type symbol on the left hand side, and there is a rule for each type symbol occurring in the type rules.*

Example 6 *The following type natlist characterizes lists of natural numbers:*

$natlist \longrightarrow nil; cons(nat, natlist)$
$nat \longrightarrow 0; 1; 2; \ldots$

A *variable typing* is a mapping $\sigma : \mathcal{V} \rightarrow \mathcal{T}$. An expression $t \in \mathcal{E}$ is of type $\tau \in \mathcal{T}$ with respect to a given type definition and a variable typing, written $t : \tau$, if (i) $t \in \mathcal{V}$ and $\sigma(t) = \tau$, or (ii) $t = f(t_1, \ldots, t_n)$, and there is an instance of a type rule, $\tau \longrightarrow \ldots; f(\tau_1, \ldots, \tau_n); \ldots$, and $t_i : \tau_i$, $1 \leq i \leq n$.

Definition 5 permits *overloading* – a function symbol can occur in several type rules. Thus, a given expression may be of more than one type. We also allow type rules containing an infinite number of distinct function symbols on the right-hand side. Thus, standard infinite types such as *integer* are permitted, defined by a rule with an infinite number of cases containing the numeric constants.

In order to define TbHEm we need to handle types with an infinite number of cases. This can be done simply by using some sort of intensional notation for them (e.g. $\mathbb{N}$ for natural numbers). However, at the same time we need to distinguish some finite number of elements of the type. Hence, we introduce the following extra annotation into type rules. The right-hand side of each type rule consists of two disjoint components, each possibly empty. More precisely, type rules are of the form $h(\bar{T}) \longrightarrow F; I$, where the union $F \cup I$ are the cases in the type rule, $F \cup I$ is non-empty, F is either empty or finite and I is either empty or infinite. We say that a type $\tau \in \mathcal{T}$ is of *infinite component* if I is non-empty in the rule defining τ. Otherwise it is said to be of *finite component*. Thus, for types of infinite component there are infinitely many ways of splitting them into type rules; for example $nat \longrightarrow F; I$ where $F = \emptyset$ and $I = \mathbb{N}$, or $F = \{0, 1, 2\}$ and $I = \mathbb{N} \setminus \{0, 1, 2\}$, etc. The way in which infinite components are split affects the behavior of TbHEm.

3. Type-based Homeomorphic Embedding

HEm turns out to be unsatisfactory, due to the restriction to finite signatures. Most real-life programs involve infinite signatures. These, for example, appear quite easily if the program performs arithmetic operations. Indeed, the fully general definition of HEm which dates back to the 1960s [11,7], referred to as the *extended homeomorphic embedding* ($\trianglelefteq^*$), allows infinite signatures. It is based on two generic relations, $\preceq_\Sigma$ and $\preceq_S$ on function symbols and sequences of expressions respectively. It can be shown that if these relations are wbrs (resp. wqos) then $\trianglelefteq^*$ is a wbr (resp. wqo). The next definition is adapted from Leuschel [13], but we use the symbol $\preceq_\Sigma$ instead of $\preceq_F$.

Definition 7 (extended HEm,$\trianglelefteq^*$) *Given a* wqo $\preceq_\Sigma$ *on the function symbols and a* wqo $\preceq_S$ *on sequences of expressions, the* extended homeomorphic embedding *on expressions is defined by the following rules:*
(i) $X \trianglelefteq^* Y$ *if* X *and* Y *are variables*
(ii) $s \trianglelefteq^* f(t_1, \ldots, t_n)$ *if* $s \trianglelefteq^* t_i$ *for some* i
(iii) $f(s_1, \ldots, s_n) \trianglelefteq^* g(t_1, \ldots, t_m)$ *if*
(a) $f \preceq_\Sigma g$,
(b) $\langle s_1, \ldots, s_n \rangle \preceq_S \langle t_1, \ldots, t_m \rangle$, *and*
(c) $\exists i_1, \ldots, i_n$ *such that* $1 \leq i_1 < \ldots < i_n \leq m$ *and* $\forall j \in \{1, \ldots, n\} : s_j \trianglelefteq^* t_{i_j}$.

The most important point in the above definition is that, in contrast to Definition 4, the left- and right-hand expressions in rule *(iii)* do not have to have the same function symbol. The two function symbols are instead compared by using the relation $\preceq_\Sigma$. Furthermore, the expressions do not have to be of the same arity; the left-hand side expression can have fewer arguments than the right-hand expression. In this case, $m - n$ arguments from the right-hand expression are ignored.

We do not use the full generality of this definition here; in particular the relation $\preceq_S$ will be taken as the relation that is always true, in which case condition *(iii.b)* is trivially satisfied. As noted by Leuschel [13], the relation $\preceq_S$ could be used to give a more refined treatment of variables as well as a more refined treatment of associative operators. For

example, it would be natural to have $\wedge(a, b, c)$ embedded in $\wedge(a, b, c, d)$, but if $\wedge$ is taken as a binary functor then the embedding relation depends on the nested structure of the expressions.

The above definition establishes a family of embedding relations but leaves open the practical problem of finding an effective embedding relation, since there is no automated mechanism for finding a "good" ordering relation $\preceq_\Sigma$ on the function symbols in the signature to ensure effective termination control. This is the problem we address in the next section, where we propose using types in order to define the $\preceq_\Sigma$ relation.

3.1. *The Type-based Relation*

We now introduce *type-based homeomorphic embedding* (TbHEm). We outline how TbHEm can provide a way of generating instances of extended HEm based on given type definitions. Note that this allows taking into account program-specific factors by using the type definition associated to a given program. The types required for guiding TbHEm can be automatically inferred by program analysis, as discussed in Section 7, or be provided manually.

Intuitively, termination control using TbHEm is based on the following idea: embedding occurs between typed expressions with different function symbols, if the function symbol of the "larger" expression is from the infinite component of its type. However, as long as we compare distinct expressions from an infinite type whose function symbols are from the finite component of the type, we can safely use essentially the standard embedding relation. This motivates the definition of the relation $\preceq_{\Sigma,D}$, which plays the role of $\preceq_\Sigma$ in Definition 7.

Definition 8 ($\preceq_{\Sigma,D}$) *Given a type definition D, let $\preceq_{\Sigma,D}$ be the following relation on the set of pairs $\Sigma \times \mathcal{T}$. $(f_1, \tau_1) \preceq_{\Sigma,D} (f_2, \tau_2)$ iff*

- *$f_1 = f_2 \wedge \tau_1 \trianglelefteq \tau_2$, or*
- *f_2 appears in the infinite component of some type rule in D.*

Here the relation $\trianglelefteq$ is the embedding relation (Definition 4) applied to types. For example, $list(A) \trianglelefteq list(list(A))$ where $list/1$ is a type symbol. As another example, given D containing $\tau \longrightarrow F; I$ with $F = \{1, 2\}$ and $I = \mathbb{N} \setminus \{1, 2\}$ then $(1, \tau) \not\preceq_{\Sigma,D} (2, \tau)$ and $(1, \tau) \preceq_{\Sigma,D} (5, \tau)$.

Lemma 1 *Let D be a type definition, Σ a set of function symbols and $\Sigma_\mathcal{T}$ a finite set of type symbols. Assume that every function symbol in Σ appears in some type rule in D, and that every type symbol in $\Sigma_\mathcal{T}$ appears on the left of some rule in D. Then $\preceq_{\Sigma,D}$ is a* wqo *on the set $\Sigma \times \mathcal{T}$.*

PROOF. It can easily be verified that $\preceq_{\Sigma,D}$ is reflexive and transitive, as required by Definition 2. Now, we prove the wbr property (Definition 3). The proof is by contradiction. Assume that there is an infinite sequence of pairs from $\Sigma \times \mathcal{T}$ of the form $(f_0, \tau_0), (f_1, \tau_1), \ldots$, and for all i, j, $i < j \rightarrow (f_i, \tau_i) \not\preceq_{\Sigma,D} (f_j, \tau_j)$. We distinguish two cases:

(i) First, assume that there is a finite number of function symbols from Σ occurring in the sequence. Then there must exist some f occurring infinitely often in the sequence, say $(f, \tau_{k_1}), (f, \tau_{k_2}), \ldots$. The relation $\trianglelefteq$ is a wqo on $\mathcal{T}$ since $\Sigma_\mathcal{T}$ is finite. Hence there must exist i, j such that $i < j$ and $\tau_{k_i} \trianglelefteq \tau_{k_j}$. Hence $(f, \tau_{k_i}) \preceq_{\Sigma,D} (f, \tau_{k_j})$ which contradicts the assumption.

(ii) Second, assume that there is an infinite set of function symbols from Σ occurring in the sequence. Then there must exist some $j > 0$, such that f_j is in the infinite component of some type rule in D, in which case $(f_i, \tau_i) \preceq_\Sigma (f_j, \tau_j)$ for all $i < j$ which contradicts the assumption.

Hence, there are no such infinite sequences and together with reflexivity and transitivity this establishes that $\preceq_{\Sigma,D}$ is a wqo.

The next definition presents our notion of type-based homeomorphic embedding, $\trianglelefteq_T$, based on the above relation $\preceq_{\Sigma,D}$. Since we assume that the relation $\preceq_S$ is true for all arguments, we omit it together with its associated condition (which is trivially true) in the definition below.

Definition 9 (TbHEm, $\trianglelefteq_T$) *Given a type definition D and the relation $\preceq_{\Sigma,D}$, the embedding relation over typed expressions, written $\trianglelefteq_T$, is defined by the following rules:*

(i) $Y{:}\tau_Y \trianglelefteq_T X{:}\tau_X$ *for all variables X, Y;*

(ii) $s : \tau \trianglelefteq_T f(t_1, \ldots, t_n) : \tau'$ *if* $s : \tau \trianglelefteq_T t_i : \tau'_i$ *for some i, where* $\tau' \longrightarrow \ldots; f(\tau'_1, \ldots, \tau'_n); \ldots$ *is an instance of a type rule in D;*

(iii) $f(s_1, \ldots, s_n){:}\tau \trianglelefteq_T g(t_1, \ldots, t_m){:}\tau'$ *if*

- *(a)* $(f, \tau) \preceq_{\Sigma,D} (g, \tau')$ *and*
- *(b)* $\exists i_1, \ldots, i_n$ *such that* $1 \leq i_1 < \cdots < i_n \leq m$ *and* $\forall j \in \{1, \ldots, n\}$, $s_j : \tau_j \trianglelefteq_T t_{i_j} : \tau'_{i_j}$, *where* $\tau_1, \ldots, \tau_n$ *(resp.* $\tau'_{i_1}, \ldots, \tau'_{i_n}$*) are the types of* $s_1, \ldots s_n$ *(resp.* $t_{i_1}, \ldots, t_{i_n}$*).*

The following theorem states the *soundness* of TbHEm. Informally, it states that infinite sequences cannot be built when the TbHEm relation is used to blow the whistle. This guarantees that TbHEm can be safely used in online tools (see Section 1).

Theorem 10 (soundness) *For all infinite sequences $e_1 : \tau_1, e_2 : \tau_2, \ldots$ of typed expressions with respect to a type definition D there exists $i < j$ such that $e_i : \tau_i \trianglelefteq_T e_j : \tau_j$.*

PROOF. The proof amounts to demonstrating that $\trianglelefteq_T$ is a wqo (see Definition 3) on typed expressions. By Theorem 4 from [13], this in turn follows if $\preceq_{\Sigma,D}$ is a wqo. As the sequence consists of typed terms with respect to some type definition D, all the types occurring in the sequence are constructed from the finite set of type symbols occurring in D and hence $\preceq_{\Sigma,D}$ is a wqo by Lemma 1. Hence the main result follows.

4. A Case-study in Online Partial Evaluation

Partial evaluation (PE) [10] is a semantics-based program transformation technique whose purpose is to specialize a program w.r.t. the part of its input data which is known at specialization time. Essentially, a partial evaluator dynamically expands program states to propagate the known input data, giving rise to a (possibly infinite) sequence of expressions which represent states. HEm has proven to be very effective in practice to control PE: not only does it ensure termination, it often allows sequences to grow sufficiently large to obtain accurate results while at the same time stopping the computation as soon as potential redundancy is detected.

This section presents as case-study a classical and non-trivial application of online PE: the specialization of interpreters. In particular, we consider an interpreter (implemented in Prolog) for a simple, imperative, bytecode language. In theory [8], the specialization of such an interpreter w.r.t. a particular bytecode program allows transforming the bytecode program into a semantically equivalent version written in Prolog. In practice, the quality and usefulness of the transformation depends on the particular techniques used to control the process. We have implemented the proposed TbHEm relation within a partial evaluator of logic programs [19], together with the procedure for constructing a monomorphic well-typing devised by Bruynooghe *et al.* [5] to automatically infer the types.

In PE of interpreters, termination problems occur as soon as the bytecode program w.r.t. which we are specializing the interpreter has a loop or a recursion whose termination condition is undecidable at specialization time. Let us consider the bytecode program fragment below, which corresponds to the simple loop "*for(i = 0; i < n; i++){}*":

```
0:push(0); 1:store(i); 2:load(i); 3:load(n);
4:ifge(7); 5:inc(i); 6:goto(2); 7:...
```

The two instructions at program counters 0 and 1 initialize `i` to 0. Note that the bytecode language is stack-based, e.g., to perform the operation $i < n$ variables `i` and `n` are first pushed on the stack $(2, 3)$ so that the conditional branching `ifge` (i.e. if greater or equal) uses them. To understand the problem, it is enough to know that the interpreter manipulates an environment of the form $s(PC, LV)$ where PC is the program counter and LV is the list of local variables, in this case $[N, I]$. In the following we ignore variable N and the list constructor for simplicity, thus we write $s(PC, I)$. Given a program, the PC can only take a finite number of values, while the local variables can, in general, change infinitely. The well-typing analysis of [5] allocates the type τ to every $s(PC, I)$ expression such that:

$\tau \longrightarrow s(\tau_1, \tau_2)$
$\tau_1 \longrightarrow F; I$ with $F = \{0, 1, \ldots, 7\}$ and $I = \mathbb{N} \setminus F$
$\tau_2 \longrightarrow \emptyset; \mathbb{N}$

During the specialization of the interpreter w.r.t. this particular program, the partial evaluator expands the interpreter states to propagate the information known from the bytecode program. The following (infinite) sequence of expressions arises $\ldots, s(2,0), s(3,0), s(4,0), s(5,0), s(6,0), \underline{s(2,1)}, s(3,1), \ldots, s(6,1), s(2,2), s(3,2), \ldots$. The program counter loops in the interval [2..6], while variable I is infinitely incremented by one after each loop iteration. An optimal strategy should only expand the above sequence until the underlined expression $s(2,1)$ appears, which actually corresponds to a loop in the program. This allows transforming the loop into the Prolog code:

```
p(I,N) :- I >= N.
p(I,N) :- I < N, I1 is I+1, p(I1,N).
```

Such an optimal behavior is achievable by using TbHEm in combination with the above (automatically inferred) types. Note that $s(2,0) : \tau \not\trianglelefteq_T s(3,0) : \tau$ as $2 : \tau_1 \not\trianglelefteq_T 3 : \tau_1$ while $s(2,0) : \tau \trianglelefteq_T s(2,1) : \tau$ as $0 : \tau_2 \trianglelefteq_T 1 : \tau_2$.

However, stopping the derivation later causes unnecessary unrollings of the loop, thus producing an over-specialized program. E.g., this program is obtained when the sequence is stopped at $s(2,2)$ (two loop unrollings):

```
p(I,N) :- I >= N.
p(I,N) :- I < N, I1 is I+1, I1 >= N.
p(I,N) :- I < N, I1 is I+1, I1 < N,
          I2 is I1+1, p(I2,N).
```

This often highly degrades both the efficiency of the specialization process and the quality of the specialized program. Experimental evidence for this is shown in Section 6 below.

On the other hand, stopping the derivation earlier, e.g. at $s(3,0)$, as other techniques would do, results in a very poor specialization. Note that, in the limit, we could perform a single unfolding step per atom. This basically results in obtaining exactly the same interpreter we started from. Therefore, no gains have been achieved at all and PE does not remove the interpretation layer. Also, if we stop too early, many atoms will be filtered out in order to guarantee termination. This may result in an important information loss. Due to space limitations, we do not present a full algorithm for PE of logic programs here (see, e.g., [14] for more details). Note that as soon as we filter away the value of the PC, execution could proceed by any of the instructions in the bytecode program, which results in large specialization times and in residual programs with plenty of useless code.

5. Instances of Type-based Embedding

This section shows that existing relations based on embedding, which are currently used in state-of-the-art specialization tools (e.g. [2,14,13]) can be reconstructed as instances of TbHEm just by providing a particular type. Let us make a distinction between the *static* symbols occurring in the program and the goal, and the remaining ones, called the *dynamic* symbols. We use S_τ to denote the set of all $f(\tau,....,\tau)$ where f is a static symbol.

5.1. *Embedding with Number Filtering.*

In programs which contain arithmetic as the only way of generating an infinite number of symbols, a relatively straightforward solution in order to recover termination is to use the $\trianglelefteq_{num}$ relation. It is an adaptation of HEm which filters out numeric values, i.e., any number embeds any other number. The $\trianglelefteq_{num}$ relation could be reconstructed as a TbHEm assuming that every argument in every predicate is of type τ_{num} which is defined as:

$$\tau_{num} \longrightarrow S_{\tau_{num}} \setminus num; num$$

where num is the infinite set of all numbers.

Example 11 *Let us re-consider the example in Section 4. The behavior of $\trianglelefteq_{num}$ is obtained using $\trianglelefteq_T$ by allocating the type τ to every expression of the form $s(PC,I)$, where:*

$$\tau \longrightarrow s(\tau_{num}, \tau_{num})$$
$$\tau_{num} \longrightarrow \emptyset; \mathbb{N}$$

Unfortunately, this modification to HEm is far too conservative and leads to excessive precision loss.

Example 12 *The sequence in Section 4 is stopped too early by $\trianglelefteq_{num}$ as $s(2,0) \trianglelefteq_{num} s(3,0)$, thus breaking feature i) in Section 1.*

5.2. *Static vs. Dynamic Symbols.*

TbHEm generalizes an idea sketched by Leuschel [13] to build an extended homeomorphic embedding based on a distinction between the static and the dynamic symbols. This relation is denoted in the following as $\trianglelefteq^*_S$. We introduce the following extra notation. We use Dyn to denote the infinite number of cases of the form $f(\tau_d,....,\tau_d)$ where f is a dynamic symbol and $\tau_d \longrightarrow \emptyset; Dyn$.

Then, $\trianglelefteq^*_S$ can be reconstructed as a TbHEm assuming that every argument in every predicate is of type τ_S which is defined as:

$$\tau_S \longrightarrow S_{\tau_S}; Dyn$$

Example 13 *For our working example, every expression $s(PC,I)$ would be allocated the type τ:*

$$\tau \longrightarrow s(\tau_S, \tau_S)$$
$$\tau_S \longrightarrow F; I \text{ with } F = \{0,1,\ldots,7\} \text{ and } I = \mathbb{N} \setminus F$$

As discussed in Section 1, this relation lacks control over the different contexts in which static symbols occur in the program. For example, this relation makes no distinction between the set of values which an argument can take w.r.t. the set of values for other arguments: all static symbols in the program are put in the same set, regardless of where they come from or where they are used. Furthermore, the result of specializing a piece of code depends on whether in the same compilation unit there

is a lot of (dead) code or not, since all static symbols, even if they appear in the context of completely unrelated subprograms will be considered as part of the finite component.

Example 14 *Let us re-consider the situation in Section 4. Unlike $\trianglelefteq_{num}$, in this case we have that $s(2,0) \not\trianglelefteq^*_S s(3,0)$. However, as both argument positions are given the same type, as opposed to the type given in Section 4, the loop will not be detected until an atom containing a number not occurring in the program arises in the sequence, namely $s(2,8)$, since $s(2,0) \trianglelefteq^*_S s(2,8)$. As explained in Section 4, this over expansion can highly degrade the efficiency of the specialization and the quality of the specialized programs.*

6. Experimental Evaluation

In order to measure the performance of TbHEm, we experimentally evaluated our case-study using TbHEm and compared the results against those obtained using $\trianglelefteq$, $\trianglelefteq_{num}$ and $\trianglelefteq^*_S$. We measured two aspects which are crucial in the specialization of interpreters, the specialization time and the residual program size. Both aspects are directly related to the quality of the decompilation.

From the experiments we conclude that $\trianglelefteq_T$ always guarantees termination (unlike $\trianglelefteq$) and behaves significantly better than $\trianglelefteq_{num}$ and $\trianglelefteq^*_S$. We compute the gain as *Old-Cost/New-Cost* and obtain an average gain of 2.3 in time and 14.4 in size w.r.t. $\trianglelefteq_{num}$, and 8.9 in time and 4.23 in size w.r.t. $\trianglelefteq^*_S$. Furthermore, $\trianglelefteq_T$ behaves at least as well as $\trianglelefteq$ in the examples in which $\trianglelefteq$ terminates, even after adding the additional cost taken by the well-typing analysis. We have observed that the largest gains are obtained when the sets of numbers in the different contexts do not intersect. In these cases, our method benefits from the context-sensitivity of TbHEm which directly contributes to obtaining smaller decompiled programs and times. As an example, for a *linear search* algorithm, we produce a 1.7 KB Prolog program in about 300 ms using $\trianglelefteq_T$, while we obtain a 9 KB Prolog program in 4 secs using $\trianglelefteq^*_S$. For this one, $\trianglelefteq_{num}$ gets a 13.7 KB Prolog program in about 540 ms while $\trianglelefteq$ does not terminate.

7. Discussion

This note presents a novel, type-based, homeomorphic embedding relation (TbHEm) and proves its soundness. We show that existing approaches which extend the untyped embedding relation to handle infinite signatures can be reconstructed as instances of our TbHEm relation.

The practicality of our approach heavily depends on automatically being able to infer suitable types to be used in combination with TbHEm. We note first that the problem does not allow a precise, computable solution. Determining the exact set of symbols that can appear at run-time at a specific program point, and in particular determining whether the set is finite, is closely related to termination detection. Let us briefly describe existing methods developed in the context of logic programming to infer the required types. As pointed out in Section 6, a well-typing analysis for logic programs was described by Bruynooghe *et al.* [5]. The procedure scales well (roughly linear in program size) and is robust, in that every program has a well-typing. We have seen in Section 6 that one can first apply this analysis to infer well-typings and then achieve good specializations by using the well-typings in combination with TbHEm.

An important observation is that, in order to take full advantage of TbHEm in practice, it is not always necessary to know the actual type definitions. In particular it suffices to know whether the infinite component of type rules is (transitively) empty or not. As another way to infer the types, in a program with built-ins, we can use existing static analyses which allow determining that the type of an argument has a finite signature. We can provide this information without having to specify the exact type. A similar idea has been recently outlined by Ruggieri and Mesnard [20], where a type system for linear constraints and its use in mode analysis of CLP programs is presented.

Analyses exist that make over-approximations of the set of values that a program's numeric arguments can have. Polyhedral analyses and interval analyses are perhaps the most widely known of these and they have successfully been applied to constraint logic programs [3]. Such analyses can determine upper and lower bounds for arguments. If an argument is bounded from above and below, and it is known that such argument takes on integral values, then it can only take a finite set of values. In general, the better the derived types are, the further the sequences can be extended without risking non-termination. If the derived types have finite components that are too small, then it is more likely that sequences will be stopped too early; if they are too large, then se-

quences could be expanded too far, producing an unnecessary expansion.

Though we have outlined procedures to infer types in the context of logic programming, our type-based relation is not tied to any programming paradigm. Moreover, it can be used for a wide range of applications (as those mentioned in Section 1).

Acknowledgments.

We gratefully thank the anonymous referees for many useful comments and suggestions. Preliminary material of this note appeared in the Proc. of LOPSTR 2007 [1]. This work was funded in part by the IST program of the European Commission, Future and Emerging Technologies under the IST-231620 *HATS* project, by the Spanish Ministry of Education (MEC) under the TIN-2005-09207 *MERIT* and TIN-2008-05624 *DOVES* projects, the Madrid Regional Government under the S-0505/TIC/0407 *PROMESAS* project and the Danish Natural Science Research Council under grant FNU 272-06-0574 (SAFT).

References

[1] E. Albert, J. Gallagher, M. Gómez-Zamalloa, and G. Puebla. Type-based Homeomorphic Embedding and its Applications to Online Partial Evaluation. In *17th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'07)*, volume 4915 of *LNCS*, pages 23–42. Springer-Verlag, February 2008.

[2] E. Albert, M. Hanus, and G. Vidal. A Practical Partial Evaluation Scheme for Multi-Paradigm Declarative Languages. *Journal of Functional and Logic Programming*, 2002(1), 2002.

[3] F. Benoy and A. King. Inferring Argument Size Relationships with CLP(R). In John P. Gallagher, editor, *Logic-Based Program Synthesis and Transformation (LOPSTR'96)*, volume 1207 of *LNCS*, pages 204–223, Springer-Verlag, August 1996.

[4] R. Bol. Loop Checking in Partial Deduction. *Journal of Logic Programming*, 16(1&2):25–46, 1993.

[5] M. Bruynooghe, J. Gallagher, and W. Van Humbeeck. Inference of Well-typings for Logic Programs with Application to Termination Analysis. In *12th International Static Analysis Symposium (SAS'05)*, volume 3672 of *LNCS*, pages 35–51. Springer-Verlag, 2005.

[6] N. Dershowitz. Orderings for Term-Rewriting Systems. *Theoretical Computer Science*, 17:279–301, 1982.

[7] N. Dershowitz and J. P. Jouannaud. Rewrite Systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B*, pages 243–320, Elsevier, 1990.

[8] Y. Futamura. Partial Evaluation of Computation Process - An Approach to a Compiler-Compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.

[9] G. Higman. Ordering by Divisibility in Abstract Algebras. *Proc. London Math. Soc.*, 2:326–336, 1952.

[10] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall, New York, 1993.

[11] J.B. Kruskal. Well-quasi-ordering, the Tree Theorem, and Vazsonyi's Conjecture. *Transactions of the American Mathematical Society*, 95:210–225, 1960.

[12] M. Leuschel. On the Power of Homeomorphic Embedding for Online Termination. In Giorgio Levi, editor, *Proceedings of SAS'98*, volume 1503 of *LNCS*, pages 230–245, Springer-Verlag, September 1998.

[13] M. Leuschel. Homeomorphic Embedding for Online Termination of Symbolic Methods. In *The Essence of Computation*, volume 2566 of *LNCS*, pages 379–403, Springer, 2002.

[14] 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.

[15] M. Leuschel and B. Martens. Global Control for Partial Deduction through Characteristic Atoms and Global Trees. In *1996 Dagstuhl Seminar on Partial Evaluation*, volume 1110 of *LNCS*, pages 263–283, Schloß Dagstuhl, 1996.

[16] 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.

[17] R. Marlet. *Vers une formalisation de l'èvaluation partielle.* PhD thesis, Universitè de Nice - Sophia Antipolis, December 1994.

[18] B. Martens and D. De Schreye. Automatic Finite Unfolding Using Well-Founded Measures. *Journal of Logic Programming*, 28(2):89–146, 1996.

[19] G. Puebla, E. Albert, and M. Hermenegildo. Efficient Local Unfolding with Ancestor Stacks for Full Prolog. In *Proc. of LOPSTR'04*, volume 3573 of *LNCS*, pages 149–165, Springer, 2005.

[20] S. Ruggieri and F. Mesnard. Typing Linear Constraints for Moding CLP(R) Programs. In *14th International Symposium on Static Analysis*, volume 5079 of *LNCS*, pages 128–143, Springer, 2008.

[21] D. Sahlin. Mixtus: An Automatic Partial Evaluator for Full Prolog. *New Generation Computing*, 12(1):7–51, 1993.

[22] Z. Somogyi, F. Henderson, and T. Conway. The Execution Algorithm of Mercury: an Efficient Purely Declarative Logic Programming Language. *Journal of Logic Programming*, 29(1–3):17–64, October 1996.

[23] M.H. Sørensen and R. Glück. An Algorithm of Generalization in Positive Supercompilation. In *Proc. of ILPS'95*, pages 465–479, The MIT Press, 1995.