Type-Based Homeomorphic Embedding and Its Applications to Online Partial Evaluation

Elvira Albert[1], John Gallagher[2], Miguel Gómez-Zamalloa[1], and Germán Puebla[3]

[1] DSIC, Complutense University of Madrid, E-28040 Madrid, Spain
[2] CBIT, Roskilde University, DK-4000 Roskilde, Denmark
[3] CLIP, Technical University of Madrid, E-28660 Boadilla del Monte, Madrid, Spain

Abstract. *Homeomorphic Embedding* (HEm) has proven to be very powerful for supervising termination of computations, provided that such computations are performed over a *finite signature*, i.e., the number of constants and function symbols involved is finite. However, there are situations, for example numeric computations, which involve an infinite (or too large) signature, in which HEm does not guarantee termination. Some extensions to HEm for the case of infinite signatures have been proposed which guarantee termination, but they 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 introduce *Type-based Homeomorphic Embedding* (TbHEm) as an extension of the standard, untyped HEm to deal with infinite signatures. In the paper, we show how TbHEm can be used to improve the accuracy of *online partial evaluation*. For this purpose, we propose an approach to constructing suitable types for partial evaluation automatically based on existing analysis tools for constraint logic programs. We also present useful properties of types which allow us to take full advantage of TbHEm in practice. Experimental results are reported which show that our work improves the state of the practice of online partial evaluation.

1 Introduction

The *homeomorphic embedding* (HEm) relation [10,11,12] has become very popular to ensure online termination of *symbolic* transformation and specialization methods and it is essential to obtain powerful optimizations, for instance, in the context of online Partial Evaluation (PE) [9]. Intuitively, HEm is a structural ordering under which an expression t_1 is greater than, i.e., it *embeds*, another expression t_2, written as $t_2 \trianglelefteq t_1$, if t_2 can be obtained from t_1 by deleting some parts, 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 can be used to guarantee termination because, provided the set of constants and functors is finite, every infinite sequence of expressions $t_1, t_2, \ldots$, contains at least a pair of elements t_i and t_j with $i < j$ s.t. $t_i \trianglelefteq t_j$. Therefore, when iteratively computing a sequence $t_1, t_2, \ldots, t_n$, finiteness of the sequence can be guaranteed by using HEm as a *whistle*. Whenever a new expression t_{n+1} is to

King, A. (Ed.): LOPSTR 2007, LNCS 4915, pp. 23–42, 2008.

be added to a finite sequence $t_1, \ldots, t_n$, we first check whether t_{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. Otherwise, t_{n+1} can be safely added to the sequence and the computation can proceed.

Two key features for the success of HEm as an approach for guaranteeing online termination are i) in the case of finite sequences, it often allows sequences to grow considerably large before the whistle blows, to the point that in a good number of cases the full sequence can be computed without the whistle blowing at all; ii) in the case of infinite sequences, it often identifies (potential) non-termination quickly, and the whistle blows without unnecessarily further expanding the sequence.

While HEm has been proved very powerful for symbolic computations, some difficulties remain in the presence of infinite signatures, such as the numbers. In the case of logic programs, infinite signatures appear as soon as certain Prolog built-ins such `is/2`, `functor/3` `name/2`, `=../2`, `atom_codes/2`, etc. are used. Some extensions to HEm over infinite signatures have been defined and used in practice (e.g. [11,2]), but they are often too ad-hoc, i.e., they only allow constants which appear explicitly in the program, regardless of which part of the program (predicate, argument position) they appear. As the approach is purely *syntactic*, it sometimes turns out to be too conservative ("whistling" too early) in practice, breaking feature i) above; while it can also be too aggressive, thus also sometimes breaking feature ii) above.

In this paper, we introduce the *type-based homeomorphic embedding* (TbHEm) relation which by taking information about the behavior of the program into account, provides more precise results in the presence of infinite signatures. In a sense, whereas [11,2] take a simple *syntactic* approach to extending the HEm relation, we propose a *semantic* approach for such extension. To achieve this, our typed relation is defined on types structured in two parts: a finite component and an infinite component. Intuitively, TbHEm allows expanding sequences as long as, whenever we compare two terms of a given type, the actual symbols which appear in such terms belong to the finite component of the type.

We illustrate the benefits of TbHEm in the context of online Partial Evaluation (PE) [9]. In particular, we use a simplified interpreter for an imperative, stack-based bytecode language written in Prolog whose specialization (if successful) allows decompiling bytecode programs to Prolog. We show how to automatically construct typings by relying on existing analysis techniques for the inference of well-typings [5]. Moreover, we present the property of a type being of *finite signature* (resp. *infinite signature*) which guarantees that all terms in the type are built out of a finite (resp. infinite) number of constant and functor symbols. We also outline how analysis of numeric bounds can be used to infer finite signature properties of types. In the case of finite signature, we can safely apply traditional HEm. We report on experimental results which compare TbHEm with previous proposals and show the benefits of our approach for the specialization of logic programs with infinite signatures.

The rest of the paper is organized as follows. Sect. 2 recalls some basic notions of PE, with special emphasis on the role of embedding. In Sect. 3, we review existing proposals in specialization of interpreters. In Sect. 4, we introduce TbHEm and prove its correctness. Sect. 5 proposes the use of well-typings as suitable types for the application of TbHEm in online PE and reports some experiments. Sect. 6 presents interesting properties of types to use TbHEm in practice, together with some experimental results. Finally, Sect. 7 discusses related work and concludes.

2 Basics on Embedding in Partial Evaluation

We assume familiarity with the basic concepts of logic programming and partial evaluation, as they are presented in e.g. [16,9]. We start by recalling the definition of HEm, which can be found for instance in Leuschel's work [14].

Definition 1 ($\trianglelefteq$). *Given two atoms $A = p(t_1, \ldots, t_n)$ and $B = p(s_1, \ldots, s_n)$, we say that B* embeds *A, written $A \trianglelefteq B$, if $t_i \trianglelefteq s_i$ for all i s.t. $1 \leq i \leq n$. The embedding relation over terms, also written $\trianglelefteq$, is defined by the following rules:*

1. *$Y \trianglelefteq X$ 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 $s_i \trianglelefteq t_i$ for all i, $1 \leq i \leq n$.*

We now explain the role that HEm plays in online PE (see e.g. [9,12,14]), which is a semantics-based program transformation technique which specializes a program w.r.t. given input data, hence, it is often called program specialization. Essentially, partial evaluators are non-standard interpreters which evaluate goals as long as termination is guaranteed and specialization is considered profitable. Given a program P and an atom S, partial evaluation produces a new program P_S which is a specialization of P for S. In logic programming, the underlying technique is to construct (possibly) *incomplete* SLD trees for the set of atoms to be specialized. In an incomplete tree, it is possible to choose *not* to further unfold a goal. Therefore, the tree may contain three kinds of leaves: failure nodes, success nodes (which contain the empty goal), and non-empty goals which are not further unfolded. The latter are required in order to guarantee termination of the partial evaluation process, since the SLD being built may be infinite. Even if the SLD trees for fully instantiated initial atoms (as regards the *input* arguments) are finite, the SLD trees produced for partially instantiated initial atoms may be infinite. This is because the SLD for partially instantiated atoms can have (infinitely many) more branches than the actual SLD tree at run-time.

HEm *in local control.* The role of local control is to determine how to construct the (incomplete) SLD trees. In particular, the *unfolding rule* decides, for each resolvent, whether to stop unfolding or to continue unfolding it and, if so, which atom to select from the resolvent. Unfolding is continued only if termination is not endangered and specialization is considered profitable. Therefore, it is

desirable to have a mechanism for guaranteeing termination which *whistles* as late as possible. State of the art local control rules based on HEm do not check for embedding against all previously selected atoms but rather only against those in its sequence of *covering ancestors* (see e.g., [18]). This increases both the efficiency of the checking and whistling later.

HEm *in global control.* Partial evaluators need to compute SLD-trees for a number of atoms in order to ensure that all atoms which appear in non-failing leaves of incomplete SLD trees are "covered" by the root of some tree (this is known as the closedness condition of partial evaluation [15]). The role of the *global control* is to ensure that we do not try to compute SLD trees for an infinite number of atoms. The usual way of achieving this is by applying an *abstraction operator* which performs "generalizations" on the atoms for which SLD trees are to be built. HEm can also be used at the global control level in order to decide when to generalize (i.e., to apply the *most specific generalization*) before proceeding to build SLD trees. Basically, for each new atom A, global control checks whether A is larger than (i.e., it embeds) any of the atoms in the set T_i (which contains the atoms in the roots of the partial trees which have already been built). If A does not embed any atom in T_i, it is added to the set; otherwise, A is generalized into $msg(A, A')$, where $A' \in T_i$ and $A' \trianglelefteq A$. At the global control level, HEm can be combined with other techniques such as *global trees*, *characteristic trees*, *trace terms*, etc. See e.g. [12] and its references.

Partial evaluation and Code Generation. As discussed above, the global control returns a set of atoms T. Finally, a partial evaluation of P w.r.t. S can then be systematically extracted from the set T. As notation, we refer to each root-to-leaf path in an SLD tree as *derivation*. The notion of *resultant* is used to generate a program rule associated with each non-failing derivation in an SLD tree. In particular, given a derivation for $P \cup \{A\}$ with $A \in T$ ending in B and θ the composition of the *mgu*s in the derivation steps, then the rule $A\theta \leftarrow B$ is called the *resultant* of the derivation. A *partial evaluation* is then defined as the union of the sets of resultants associated to the SLD trees for all atoms in T.

3 Embedding with Infinite Signatures: Motivating Example

In Fig. 1 we show a fragment of a simplified imperative bytecode interpreter implemented in Prolog. If the partial evaluator is powerful enough, given a bytecode program we can obtain a decompiled version of it in Prolog (see e.g. [1] for an object-oriented stack-based interpreter). For brevity, we omit the code of some predicates like `build_init_state/2` (whose purpose is explained below) and `localVar_update/4` which simply updates the value of a local variable. We only show the definition of `step/3` for a reduced set of instructions. The bytecode to be decompiled is represented as a set of facts `bytecode(PC,Inst)` where `PC` contains a program counter and `Inst` the corresponding bytecode instruction. A state is of the form `st(PC,OStack,LocalV)` where `OStack` represents

```
main(InArgs,Top) :-
    build_init_state(InArgs,S0),
    execute(S0,st(_,[Top|_],_)).
execute(S,S):-
    S = st(PC,_,_),
    bytecode(PC,return).
execute(S1,Sf) :-
    S1 = st(PC,_,_),
    bytecode(PC,Inst),
    step(Inst,S1,S2),
    execute(S2,Sf).
```

```
step(const(_T,Z),st(PC,S,L),S2) :-
    PCp is PC + 1,
    S2 = st(PCp,[Z|S],L).
step(istore(X),st(PC,[I|S],L),S2) :-
    PCp is PC + 1,
    localVar_update(L,X,I,Lb),
    S2 = st(PCp,S,Lb).
step(goto(O),st(PC,S,L),S2) :-
    PCp is PC+O,
    S2 = st(PCp,S,L).
....
```

Fig. 1. Fragment of simplified bytecode interpreter

the operand stack and `LocalV` the list of local variables. The predicate `main/2`, given the input method arguments `InArgs`, first builds the initial state by means of predicate `build_init_state/2` and then calls predicate `execute/2`. In turn, `execute/2` first calls predicate `step/3`, which produces `S2`, the state after executing the corresponding bytecode, and then calls predicate `execute/2` recursively with `S2` until we reach a `return` instruction.

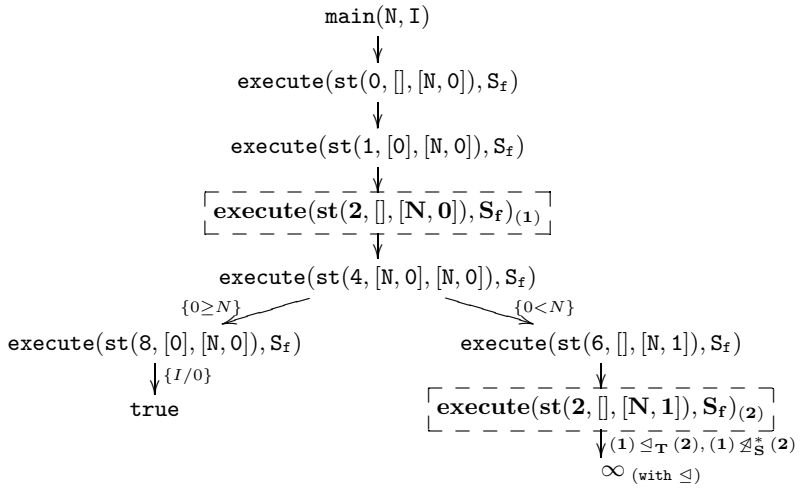
Consider the `count` method which appears in the left hand side of Fig. 2, represented as a set of facts. For clarity of the presentation, on the right hand side of Fig. 2 we show a Java source program which can be compiled into the corresponding bytecode. However, it is important to note that the decompilation is performed directly from the bytecode and that the decompiler does not have access to the source. It can be seen that `count` receives an integer and executes a loop where a counter initialized to "0" (in bytecodes `0` and `1`) is incremented by one at each iteration (bytecode `5`) until the counter reaches the value of the input parameter (checking the condition comprises bytecodes `2`, `3` and `4`). The method returns the value of the counter in bytecodes `7` and `8`. For decompiling the `count` method, we partially evaluate the interpreter w.r.t. the bytecode facts which appear to the left of the figure by specializing the atom: `main(N,I)`, where `N` is the input parameter and `I` represents the return value (i.e., the top of the stack at the end of the computation).

In Figure 3, we depict (a reduced version of) one of the SLD trees that leads to an effective decompilation of our running example and that we will refer to in the next sections. For simplicity, apart from the entry atom `main/2`, we only show atoms for `execute/2`, as it is the only recursive predicate in the program. Thus, each arrow in the tree involves the application of several unfolding steps. Note that some of the statements within the body of each `step` operation can remain residual when they involve data which is not known at specialization time. The computation rule used in the unfolding operator is able to residualize calls which are not sufficiently instantiated and select non-leftmost atoms in a safe way [3], in particular, further calls to `execute` can be selected. We represent such residual calls as labels in the arrows of the tree.

```
bytecode(0,const(int,0)).
bytecode(1,istore(1)).
bytecode(2,iload(1)).
bytecode(3,iload(0)).
bytecode(4,if_icmp(geInt,3)).
bytecode(5,iinc(1,1)).
bytecode(6,goto(-4)).
bytecode(7,iload(1)).
bytecode(8,return).
```

```
static int count(int n){
  int i = 0;
  while (i < n)
    i++;
  return i;
}
```

Fig. 2. Object program for working example




```
main(N,0) :- 0>=N.
main(N,I) :- 0<N,
             sp_execute(N,1,I).
```

```
sp_execute(N,I,I) :- I>=N.
sp_execute(N,A,I) :- A<N, A' is A+1,
                     sp_execute(N,A',I).
```

Fig. 3. Partial unfolding SLD tree and residual code of working example

3.1 Using the Original Homeomorphic Embedding

Let us first consider an online partial evaluator (which is able to accurately handle built-in predicates and to safely perform non-leftmost and) which uses HEm to control termination both at the local and global control levels. As it can be seen in the figure, the PC value "2" corresponds to the loop entry. By applying HEm, the evaluation contains a subsequence of atoms of the form: $\texttt{execute}(\texttt{st}(2,[],[\texttt{N},0]),\texttt{S}_\texttt{f})$, $\texttt{execute}(\texttt{st}(2,[],[\texttt{N},1]),\texttt{S}_\texttt{f})$, $\texttt{execute}(\texttt{st}(2,[],[\texttt{N},2]),\texttt{S}_\texttt{f}),\ldots$ marked within dashed frames in the figure, which correspond to consecutive iterations of the loop in which the control returns to the loop head (PC value 2 in the first position of the state) with a value for the loop counter

(local variable at the second position in the resulting state) increased by one. This sequence can grow infinitely, as the HEm does not flag it as potentially dangerous, which is marked by ∞ (with $\trianglelefteq$) in the figure. This is because the interpreter uses Prolog's arithmetic (i.e., the is/2 predicate), which breaks the finite signature property featured by pure logic programs.

In order to get a quality decompilation, we need to filter out the value of the counter (local variable 1) but not that of the PC. As shown in the figure, this requires stopping the derivation when we hit the atom $\texttt{execute}(\texttt{st}(2,[],[\texttt{N},1]),\texttt{S}_\texttt{f})$ (marked as $(\mathbf{1}) \trianglelefteq_{\mathbf{T}} (\mathbf{2})$) and generalize it w.r.t. the above atom within a dashed frame, resulting in $\texttt{execute}(\texttt{st}(2,[],[\texttt{N},\texttt{X}]),\texttt{S}_\texttt{f})$.

3.2 Recovering Termination: Embedding with Number Filtering

In programs which contain Prolog arithmetic but do not generate an infinite number of functors via functor/3, =../2, etc., a relatively straightforward solution in order to recover termination is to use the $\trianglelefteq_{num}$ relation, which is an adaptation of HEm which filters out numeric values, i.e., any number embeds any other number. The atom $\texttt{execute}(\texttt{st}(2,[],[\texttt{N},1]),\texttt{S}_\texttt{f})$ embeds $\texttt{execute}(\texttt{st}(2,[],[\texttt{N},0]),\texttt{S}_\texttt{f})$ under $\trianglelefteq_{num}$ and therefore we avoid non-termination. Unfortunately, this modification to HEm, is far too conservative, and leads to excessive precision loss. For instance, in the specialization of $\texttt{main}(\texttt{N},\texttt{I})$, the first two atoms for execute/2 are $\texttt{execute}(\texttt{st}(0,[],[\texttt{N},0]),\texttt{S}_\texttt{f})$ and $\texttt{execute}(\texttt{st}(1,[0],[\texttt{N},0]),\texttt{S}_\texttt{f})$. By using $\trianglelefteq_{num}$, the whistle blows at this point and unfolding has to stop. Furthermore, the latter atom is generalized at the global control level into $\texttt{execute}(\texttt{st}(\texttt{X},\texttt{Y},[\texttt{N},0]),\texttt{S}_\texttt{f})$ before proceeding with the specialization. This turns out not to be acceptable for the specialization of our interpreter, since we lose track of which the next instruction to execute is—which prevents us from eliminating the interpretation layer—and in many cases the residual program ends up containing the whole original interpreter.

3.3 Increasing Accuracy: Static Symbols in the Program

A simple syntactic way of increasing the accuracy while preserving termination, as proposed in [11], consists in considering two sets of symbols: those which appear explicitly in the program and goal, which is obviously finite, and another infinite set which contains all other symbols. In the following, this relation is denoted as $\trianglelefteq^*_S$. When comparing two terms we keep those symbols which belong to the finite set and filter out all other ones. Under this relation, the atom $\texttt{execute}(\texttt{st}(1,[0],[\texttt{N},0]),\texttt{S}_\texttt{f})$ does not embed the atom $\texttt{execute}(\texttt{st}(0,[],[\texttt{N},0]),\texttt{S}_\texttt{f})$ in the figure, as the numbers 0 and 1 are different static symbols which occur in the program. Hence, we are not forced to generalize them and we can keep the PC value.

Unfortunately, the $\trianglelefteq^*_S$ relation turns out not to be optimal in our case either since $\texttt{execute}(\texttt{st}(2,[],[\texttt{N},1]),\texttt{S}_\texttt{f})$ does not embed $\texttt{execute}(\texttt{st}(2,[],[\texttt{N},0]),\texttt{S}_\texttt{f})$. This means that unfolding proceeds with a second iteration of the loop. The process is guaranteed to terminate, we will unfold at most as many iterations of the loop as distinct numbers appear in the program. However, we are not able to

achieve the quality decompilation which appears at the bottom of Figure 3. For obtaining such good decompilation, we need to generalize the loop counter, i.e., the atom $\texttt{execute}(\texttt{st}(2,[],[\texttt{N},1]),\texttt{S}_\texttt{f})$ has to embed $\texttt{execute}(\texttt{st}(2,[],[\texttt{N},0]),\texttt{S}_\texttt{f})$. Intuitively, the reason why this relation does not behave optimally is because the fact that many symbols appear explicitly in the program for one argument (in our case the PC counter) should somehow not affect the set of symbols which we should consider as static for other arguments (the list of local variables).

Note that the use of characteristic trees [13] to control the degree of polyvariance does not lead to an optimal decompilation in this example either. The reason is that characteristic trees concern only global and not local control. Therefore, as already mentioned above, they do not stop the local derivation which may perform as many unrollings of the loop as different values for the loop counter there are in the program. Once the local control stops this unfolding process, the value of the counter will be generalized by the global control. However, the characteristic tree of this generalized term is clearly not equivalent to the one of the previous unrolling for the different values in the counter. Therefore, the decompilation of the loop body for the static values remains residual in the specialized code as well.

4 Type-Based Homeomorphic Embedding

In the presence of infinite signatures, a general method of defining homeomorphic embedding relations exists; an *extended homeomorphic embedding relation* is defined in [11] based on previous results by Kruskal [10] and by Dershowitz [6]. This solution defines a family of embedding relations, where a subsidiary ordering on function symbols plays an essential role. However, we argue that this does not really solve the practical problem of finding an effective embedding relation, since there is no automated mechanism for finding the "right" ordering relation on the function symbols in the signature.

In this section, we propose *typed-based homeomorphic embedding* (TbHEm for short), a relation which improves HEm by making use of additional information provided in the form of types. We outline how this approach can be seen as a way of generating instances of extended HEm as defined by Leuschel, including the possibility of taking into account the program semantics. The types required for guiding TbHEm can be provided manually or, interestingly, be automatically inferred by program analysis, as we will see in Section 5.

4.1 Types: Preliminaries and Notation

In the following, let P be a program and Σ_P be a (possibly infinite) signature including the functions and constants appearing in P and goals for P as well as in computations of P. We adopt the syntax of Mercury [20] for type definitions. *Type expressions (types)*, elements of $\mathcal{T}$, are constructed from an infinite set of type variables (parameters) $\mathcal{V}_\mathcal{T}$ and an alphabet of ranked type symbols $\Sigma_\mathcal{T}$; these are disjoint from the set of variables V and the alphabet of functors Σ_P of a given program P respectively.

Definition 2 (type definition). *A* type rule *for a type symbol $h/n \in \Sigma_{\mathcal{T}}$ is of the form $h(\bar{T}) \longrightarrow f_1(\bar{\tau}_1); \ldots; f_k(\bar{\tau}_k); \ldots$ $(k \geq 1)$ where $\bar{T}$ is a n-tuple of distinct type variables, $f_1, \ldots, f_k, \ldots$ are distinct function symbols from Σ_P, $\bar{\tau}_i$ $(i \geq 1)$ are tuples of corresponding arity from $\mathcal{T}$, and type variables in the right hand side, if any, are from $\bar{T}$ (a condition known as* transparency *[17,8]). 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.*

We write $t : \tau$ to mean that term t is of type τ. As in Mercury [20], a function symbol can occur in several type rules. In the definition above we allow type rules containing an infinite number of cases. 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 introduce some extra annotation into type rules. We consider the right hand side of each type rule to consist of two disjoint components, each possibly empty. More precisely, we will structure a type rule as $h(\bar{T}) \longrightarrow F; I$, 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*. Note that 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.

A *predicate signature* for an n-ary predicate p is of the form $p(\bar{\tau})$ and declares a type $\tau_i \in \mathcal{T}$ for each argument of the predicate p/n. Programs are assumed to be *well-typed* in the usual sense, namely that every atom and term in a clause can be assigned types consistent with the type declarations such that the type assigned to each head atom is a variant of the signature for its predicate, the types of the body atoms are instances of the corresponding signatures, and multiple occurrences of the same variable in the clause are assigned the same type. Furthermore, we disallow *polymorphic recursion*; body atoms for recursive predicates are assigned a type that is a variant of the signature. The relevant consequences of well-typing for our purpose are firstly that a well-typed program and goal generate only well-typed atoms in computations and secondly that only a finite number of types arise during a computation. An infinite set of different types such as $h(T), h(h(T)), h(h(h(T))), \ldots$ cannot arise in a computation, due to the absence of polymorphic recursion.

4.2 Type-Based Homeomorphic Embedding

We now define TbHEm ($\trianglelefteq_T$). It follows closely the definition of the extended HEm relation defined in [11] on untyped terms; here we define a relation on typed terms. As in the definition in [11], two subsidiary relations $\preceq_F$ and $\preceq_S$ are needed. The first, $\preceq_F$, is a relation on function symbols paired with their associated types, and it refers to the infinite component of type rules described above.

Definition 3. *Let $\preceq_F$ be the following relation on the set of pairs $\Sigma_P \times \mathcal{T}$. $(f_1, \tau_1) \preceq_F (f_2, \tau_2)$ iff (1) the rules defining τ_i are of form $h_i(\bar{V}_i) \longrightarrow F_i; I_i$, for $i = 1, 2$ and (2) either $f_1 = f_2 \wedge \tau_1 = \tau_2$ or f_2 is in the infinite component I_2 of the rule for τ_2.*

For instance, given $\tau \longrightarrow F; I$ with $F = \{1, 2\}$ and $I = \mathbb{N} \setminus \{1, 2\}$ then $(1, \tau) \not\preceq_F (2, \tau)$ and $(1, \tau) \preceq_F (5, \tau)$. The other relation, $\preceq_S$, is a relation on sequences of typed terms, and for our purposes here we can take it to be true for all pairs of sequences of typed terms. In general this relation can be defined to allow more refined treatment of associative operators, among other things; as noted in [11], whether $\wedge(a, b, c)$ is embedded in $\wedge(a, b, c, d)$ depends on the nested structure of the expressions, if $\wedge$ is taken as a binary functor. Though we do not use it here, we include the relation $\preceq_S$ in the following definition for uniformity with [11], so that our notion of typed embedding becomes an instance of the extended homeomorphic embedded defined there.

Definition 4 ($\trianglelefteq_T$). *Given two typed atoms $A = p(t_1, \ldots, t_n)$ and $B = p(s_1, \ldots, s_n)$, with predicate signature $p(\tau_1, \ldots, \tau_n)$, we say that B* embeds *A, written $A \trianglelefteq_T B$, if $t_i : \tau_i \trianglelefteq_T s_i : \tau_i$ for all i s.t. $1 \leq i \leq n$. The embedding relation over typed terms, also written $\trianglelefteq_T$, is defined by the following rules:*

1. *$Y : \tau_Y \trianglelefteq_T X : \tau_X$ for all variables X, Y.*
2. *$s : \tau \trianglelefteq_T f(t_1, \ldots, t_n) : \tau'$ if $s : \tau \trianglelefteq_T t_i : \tau'_i$ for some i, where $\tau'_1, \ldots, \tau'_n$ are the respective types of $t_1, \ldots, t_n$.*
3. *$f(s_1, \ldots, s_n) : \tau \trianglelefteq_T g(t_1, \ldots, t_m) : \tau'$ if*
   (a) $(f, \tau) \preceq_F (g, \tau')$,
   (b) $(s_1 : \tau_1, \ldots, s_n : \tau_n) \preceq_S (t_1 : \tau'_1, \ldots, t_m : \tau'_m)$, and
   (c) $\exists i_1, \ldots, i_n$ 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 $\tau_1, \ldots, \tau_n, \tau'_1, \ldots, \tau'_m$ are the respective types of $s_1, \ldots s_n, t_1, \ldots, t_m$.

Rule 3 of the definition specifies that embedding can occur between terms with different function symbols, where the function symbol of the "larger" term using the $\preceq_F$ relation is from the I component of its type. However, as long as we compare distinct terms from an infinite type and remain within the finite component F of the type, no embedding (using rule 3) occurs since the condition $(f, \tau_1) \preceq_F (g, \tau_2)$ does not hold. For instance, consider the following predicate signature and type definition, $p(\tau)$ and $\tau \longrightarrow F; I$. We have that $p(1) \trianglelefteq_T p(2)$ if $F = \emptyset$ and $I = \mathbb{N}$. However, $p(1) \not\trianglelefteq_T p(2)$ if $F = \{0, 1, 2\}$ and $I = \mathbb{N} \setminus \{0, 1, 2\}$.

Proposition 1. *Given a program P that is well-typed with respect to a type definition and set of signatures, there is no infinite sequence of well-typed atoms $A_1, A_2, \ldots$ in a computation for P such that for all i, j where $i < j$, $A_i \not\trianglelefteq_T A_j$.*

Proof. First note that, by the assumption that polymorphic recursion is disallowed, only a finite number of types (up to renaming of type variables) arises in a computation. The proposition follows from the fact that is a $\trianglelefteq_T$ *well quasi order (wqo)* on typed atoms over a finite set of types. A binary relation $\leq : D \times D$ is a

wqo if (i) it is reflexive and transitive, and (ii) for all infinite sequences $d_0, d_1, \ldots$ of elements of D, $\exists i < j$ such that $d_i \leq d_j$. By Theorem 4 from [11], this in turn follows if both $\preceq_F$ and $\preceq_s$ are wqos on their respective domains, which we now prove.

The proof that $\preceq_S$ is a wqo is trivial. For $\preceq_F$, it can easily be verified that the relation is reflexive and transitive. To prove the wqo property (ii) assume that there is an infinite sequence of pairs from $\Sigma_P \times \mathcal{T}$, $(f_0, \tau_0), (f_1, \tau_1), \ldots$. First assume there is only a finite number of function symbols occurring in the sequence; in this case, since there is also a finite number of types, there must exist i and j, $i < j$, such that $f_i = f_j \wedge \tau_i = \tau_j$ and hence $(f_i, \tau_i) \preceq_F (f_j, \tau_j)$. Secondly, assume that there is an infinite set of function symbols occurring in the sequence; since the number of types is finite there must exist some $j > 0$, such that f_j is in the infinite component of the type rule for τ_j, in which case $(f_i, \tau_i) \preceq_F (f_j, \tau_j)$ for all $i < j$. Hence, $\preceq_F$ is a wqo.

Proposition 1 ensures that partial evaluation using TbHEm terminates. The idea of using a typed homeomorphic embedding generalises an idea sketched in [11] to build an extended homeomorphic embedding based on a distinction between the finite number of symbols actually occurring in the program and goal (the *static* symbols), and the rest (the *dynamic* symbols). This could be reconstructed as a TbHEm using a single type rule $term \longrightarrow F; I$ where F contains cases of the form $f(term, ..., term)$ where f is a static symbol, and I contains the infinite number of cases where f is not static.The predicate signatures would allocate the type *term* to all arguments. As discussed in Section 3.3, that approach lacks control over the different contexts in which static symbols occur in the program. Sometimes a static symbol should block embedding but other times it should not.

5 Automatic Inference of Well-Typings

In this section, we outline and experimentally evaluate an approach which, given an untyped program and a goal or set of goals, automatically infers suitable types to be used in online partial evaluation in combination with TbHEm. The approach is based on existing analysis tools for constraint logic programs.

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 and is thus undecidable. However, the better the derived types are, the more aggressive partial evaluation can be without risking non-termination. If the derived types have finite components that are too small, then over-generalization is likely to result; if they are too large, then specialization might be over-aggressive, producing unnecessary versions.

A procedure for constructing a monomorphic well-typing of an arbitrary logic program was described by Bruynooghe *et al.* [5][1]. The procedure scales well

[1] Available on-line at `http://saft.ruc.dk/Tattoo/`

(roughly linear in program size) and is robust, in that every program has a well-typing, and the procedure works with partial programs (modules). We first apply this procedure to illustrate the use of well-typings in the context of our running example and, then, we perform an experimental evaluation to assess the gains that we achieve in the specialization of interpreters by using well-typings in combination with TbHEm.

5.1 Well-Typings for Working Example

In the original type inference procedure, an externally defined predicate such as `is/2` is treated as if defined by a clause `X is Y :- true` and is thus implicitly assumed not to generate any symbols not occurring elsewhere in the program. In deriving types for partial evaluation, we provide a type for such built-ins in the form of a dummy additional "fact" for `is/2`, namely `num is num :- true`. The constant `num` (assumed not to occur elsewhere in the program) will thus propagate during type inference into those types that unify with the types of the `is` predicate arguments. In the resulting inferred types, we interpret occurrences of the constant `num` as being an abbreviation for an infinite set of cases.

Example 1. A type is inferred for the interpreter sketched in Figure 1, together with the particular bytecode program of Fig. 2. Note that the program counter is sometimes computed in the interpreter using the predicate `is/2` as an offset from the current program counter value and hence its type is in principle any number. When the extra fact `num is num :- true` is added to the program, the inferred type τ_{PC} for the program counter argument `PC` is as follows.

τ_{PC} `--> -4; 0; 1; 2; 3; 4; 5; 6; 7; 8; num`

Type τ_{PC} can be naturally interpreted as consisting of a finite part (the named constants) and an infinite part (the numbers other than the named constants). In other words, the partition F of the rule is $\{-4, 0, 1, 2, \ldots, 8\}$ and $I = \texttt{num} \setminus F$. Using the rule structured in this way, TbHEm ensures that the program counter is never abstracted away during partial evaluation, so long as its value remains in the expected range (the named constants). The atom `execute(st(1,[0],[N,0]),Sf)` does not embed `execute(st(0,[],[N,0]),Sf)` by using the type definition above, thus, the derivation can proceed. This avoids the need for generalizing the `PC` what would prevent us from having a quality specialization (decompilation) as explained in Sect. 2. The derivation will either eventually end or the `PC` value will be repeated due to a backwards jump in the code (loops). In this case, $\trianglelefteq_T$ will flag the relevant atom as dangerous, e.g., `execute(st(2,[],[N,0]),Sf)` $\trianglelefteq_T$ `execute(st(2,[],[N,1]),Sf)`, as can be seen in Fig. 3. If, however, a different value arose, perhaps due to an addressing error, the infinite part of the type rule `num` is encountered and embedding (followed by generalization of the program counter argument) would take place.

The decompiled program that we obtain using the inferred well-typings and combined with TbHEm is shown at the bottom of Fig. 3. We can observe that the decompilation is optimal in the sense that the interpretation layer has been completely removed and there is no superfluous residual code. Note that a more

sophisticated analysis could infer that τ_{PC} becomes of finite component, i.e., $I = \emptyset$ by taking $F = \{-4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$. This can be done by computing all combinations of bytecode indeces and offsets present in the program. In fact, $F = \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$ is also a correct finite component. Though this information indicates that τ_{PC} is of *finite signature* (see Section 6 below), the quality of the decompiled program does not require this extra accuracy.

5.2 Experimental Results

We have implemented the proposed TbHEm embedding relation within the partial evaluator available in CiaoPP [19] and combined it with the results obtained from the well-typing analyzer in [5]. Table 1 shows the practical benefits that we can obtain in the context of the specialization of interpreters. Each row in the table corresponds to the specialization of a bytecode interpreter w.r.t. different bytecode programs. **Counter** corresponds to the program presented in Fig. 2. We use a set of classical iterative algorithms as additional benchmarks: **Exp**, **Gcd** and **Fib** compute respectively the exponential, greatest-common-divisor and Fibonacci, and **ExpAlt** corresponds to a different implementation of the exponential. The last two benchmarks, **LinSearch** and **BinSearch**, compute respectively the classical linear and binary searches over integer arrays. Therefore, to handle them, we use an extended version of our bytecode interpreter which handles integer array manipulation. Thus, it includes a heap in the state as well as the bytecode instructions required to manipulate arrays. We have experimented as well extending the interpreter with more advanced features such as exception handling, object orientation, etc. We believe that the results obtained are generalizable to interpreters which manipulates numbers in general, and in particular to low-level language interpreters.

For each benchmark, we study the behavior of $\trianglelefteq_T$ w.r.t. $\trianglelefteq$, $\trianglelefteq_{num}$ and $\trianglelefteq^*_S$ by measuring 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. Then, from left to right, the first two columns, **Name** and **Size**, show the name of the benchmark and the size (in KBytes) of the Prolog representation of the bytecode program. The following 9 columns show specialization times (in seconds) and residual program sizes (in KBytes) for the different strategies $\trianglelefteq$, $\trianglelefteq_{num}$, $\trianglelefteq^*_S$ and $\trianglelefteq_T$. We write "-" when the specialization does not terminate. Note that, in the group of columns corresponding to $\trianglelefteq_T$, we have an additional column T_{wt} which shows the time taken by the well-typing analysis which should be added to the specialization time in order to obtain a proper evaluation of $\trianglelefteq_T$. It should be noted also that the usage of $\trianglelefteq^*_S$ would require a preprocessing time currently not being taken into account which should be no more than the times in T_{wt}. Since we do not have an implementation of $\trianglelefteq^*_S$ the results obtained for it have been obtained using the TbHEm writing by hand the corresponding types. Finally, the last two columns show the gains (in terms of time and size) of the embedding relation $\trianglelefteq_T$ w.r.t. $\trianglelefteq_{num}$ (in column $\mathbf{T/S}_{(\trianglelefteq_{num})}$) and $\trianglelefteq^*_S$ (in column $\mathbf{T/S}_{(\trianglelefteq^*_S)}$). The gain is computed as *Old-Cost*/*New-Cost*. As we can observe in the table, $\trianglelefteq_T$

Table 1. Measuring the effects of $\trianglelefteq_T$ with the bytecode interpreter

Benchmark		$\trianglelefteq$		$\trianglelefteq_{num}$		$\trianglelefteq_S^*$		$\trianglelefteq_T$			Gains	
Name	**Size**	**Tm**	**Size**	**Tm**	**Size**	**Tm**	**Size**	$\mathbf{T}_{wt}$	**Tm**	**Size**	$\mathbf{T/S}_{(\trianglelefteq_{num})}$	$\mathbf{T/S}_{(\trianglelefteq_S^*)}$
Counter	0.27	-	-	0.12	1.79	0.60	1.26	0.03	0.09	0.28	1.4/6.3	6.7/4.4
Exp	0.39	0.14	0.50	0.24	5.51	0.14	0.50	0.03	0.14	0.50	1.7/11.0	1.0/1.0
Gcd	0.35	0.13	0.38	0.23	4.80	0.14	0.38	0.03	0.11	0.29	2.2/16.3	1.4/1.3
ExpAlt	0.44	-	-	0.26	6.13	3.75	4.50	0.03	0.13	0.34	2.0/17.8	29.0/13.1
Fib	0.52	-	-	0.49	10.72	0.99	1.41	0.03	0.15	0.51	3.2/21.2	6.6/2.8
LinSearch	0.70	-	-	0.54	13.69	3.99	9.04	0.04	0.25	1.70	2.1/8.1	15.7/5.3
BinSearch	2.00	3.14	9.26	5.05	112.50	3.20	9.26	0.04	1.59	5.51	3.2/20.4	2.0/1.7

guarantees termination and behaves significantly better than $\trianglelefteq_{num}$ and $\trianglelefteq_S^*$ both in time and size. Furthermore, $\trianglelefteq_T$ behaves as well as $\trianglelefteq$ in the examples in which $\trianglelefteq$ terminates, even after adding the additional cost taken by the well-typing analysis. An important observation as regards the gains w.r.t. $\trianglelefteq_S^*$ is that for some benchmarks such gains are large while for others they are almost insignificant. The reason for this lack of improvement is that in the corresponding atoms, the local variables within the state are not instantiated to concrete values almost from the beginning. Therefore, the over-specialization problem of $\trianglelefteq_S^*$ pointed in Sect. 3.3 is not exposed. In fact, note that these cases correspond precisely to the cases where $\trianglelefteq$ terminates (due to the same reason).

6 Type-Based Homeomorphic Embedding in Practice

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, but only sufficient information for the relations $\preceq_F$ and $\preceq_S$ proposed in Sect. 4.2 to be well defined. In particular it suffices to know whether the infinite component of type rules is (transitively) empty or not. Moreover, it would be desirable to define a condition on types specifying that a type and all the types on which it depends are defined over a finite signature. In this case, we can safely revert to the simpler HEm applied directly to terms of such types. In the following we define such a condition.

Definition 5 (finite signature). *Given a type τ defined by a type rule $\tau \longrightarrow F;\emptyset$ we say that τ is of* finite signature*, denoted* $\mathsf{f\_sig}(\tau)$*, iff $F = \{f_1(\tau_{11}, \ldots, \tau_{1k_1}), \ldots, f_n(\tau_{n1}, \ldots, \tau_{nk_n})\}$ and all types $\tau_{11}, \ldots, \tau_{nk_n}$ are of finite signature.*

Hence, if a type τ is of *finite signature* the (possibly infinite) set of terms of type τ contains only a finite set of functors. As the following Proposition implies, we can then use $\trianglelefteq$ instead of $\trianglelefteq_T$ when comparing terms in the context of finite signatures.

Proposition 2. *Given two typed terms $t_1 : \tau_1$ and $t_2 : \tau_2$, if* $\mathsf{f\_sig}(\tau_2)$ *holds then $t_1 : \tau_1 \trianglelefteq_T t_2 : \tau_2 \Leftrightarrow t_1 \trianglelefteq t_2$.*

In the following, for every type τ for which $\mathsf{f\_sig}(\tau)$ holds, we simply write $\mathsf{f\_sig}$ instead of the particular type. We now propose an extension to the definition of $\trianglelefteq_T$ to consider $\mathsf{f\_sig}$ types. This is done simply by adding the following rule to Def. 4: *4.* $s{:}\tau_1 \trianglelefteq_T t{:}\mathsf{f\_sig}$ if $s \trianglelefteq t$.

In order to put these ideas into practice it is convenient to also have the type $\mathsf{i\_sig}$ which is assigned to an argument when we cannot guarantee it is of finite signature and we do not have further information available about its type. Note that we are assuming a scenario where infinite signatures can include functors as well as numbers.

Definition 6 ($\mathsf{i\_sig}$). *The type* $\mathsf{i\_sig}$ *is defined by the following type rule:* $\mathsf{i\_sig} \longrightarrow \emptyset; I$ *where* $I = \{f_1(\tau_{11}, \ldots, \tau_{1k_1}), \ldots, f_n(\tau_{n1}, \ldots, \tau_{nk_n}), \ldots\}$ *and* f_i *are all possible functors and all types* $\tau_{11}, \ldots, \tau_{nk_n}$ *are* $\mathsf{i\_sig}$.

Note that since every case of the type rule belongs to the infinite component then $s{:}\tau \trianglelefteq_T t{:}\mathsf{i\_sig}$ will always hold (as $\preceq_F$ holds for every s, τ and t). Hence, termination is trivially guaranteed for terms of type $\mathsf{i\_sig}$. In practice, in programs with infinite signatures, unless the user (or an automatic analysis) explicitly writes more concrete type declarations, a default *typing* will be assumed such that all predicates p/n of a program have the *predicate signature* $p(\tau_1, \ldots, \tau_n)$ with $\tau_i = \mathsf{i\_sig}$, $(0 \leq i \leq n)$. Then, more concrete declarations are allowed both by declaring particular types and signatures (always preserving the well-typing assumption, see Sect. 4) or by using the special type $\mathsf{f\_sig}$.

Example 2. Consider again the interpreter in our motivating example. Though it is natural to use integer numbers to represent program counters, the set of instructions is finite in any bytecode program. Therefore the `PC` can be safely declared as $\mathsf{f\_sig}$. Thus we may write the following predicate signature and type definition:

$$\texttt{execute}(\tau_{st}, \tau_{st}).$$
$$\tau_{st} \longrightarrow \{\texttt{st}(\mathsf{f\_sig}, \mathsf{i\_sig}, \mathsf{i\_sig})\}; \emptyset.$$

With this type declaration we are able to obtain the same results as in Sect 5.1 in a more efficient way, as we can get rid of the overhead produced by the comparisons checking that the current `PC` belongs to the finite part of the corresponding type. In addition, the type declaration holds for all input programs, whereas before a separate type inference was needed for each input object program.

Another interesting observation is that the relation $\trianglelefteq^*_S$ may be defined as a particular case of TbHEm by simply declaring the following particular type and assuming that every argument of every predicate is of this type: $\mathsf{s\_symb} \longrightarrow F; I$ where $F = \{f_1(\tau_{11}, \ldots, \tau_{1k_1}), \ldots, f_n(\tau_{n1}, \ldots, \tau_{nk_n})\}$ with $f_1, \ldots, f_k$ being all the functor symbols which explicitly occur in the program text plus initial goal(s) and the types $\tau_{11}, \ldots, \tau_{nk_n}, \ldots$ are $\mathsf{s\_symb}$. I contains the infinite set of all other possible functors, with auxiliary types $\mathsf{i\_sig}$ in all cases.

6.1 Automatic Inference of Finite Signature

If, in a program with builtins, we can use some static analysis which allows us to determine that the type of an argument has a finite signature, we can provide

this information to the partial evaluator as an f_sig declaration, without having to specify the exact type. E.g., given a logic program processing numeric values, analyses exist that make over-approximations of the set of values that the program arguments can have. Polyhedral analyses are perhaps the most widely known of these and they have successfully been applied to constraint logic programs [4]. Let us assume for the sake of this discussion that a polyhedral analysis can return, for a given program and goal, an approximation to the set of calls to each n-ary predicate p, in the form: $p(X_1, \ldots, X_n) \leftarrow c(X_1, \ldots, X_n)$, where the expression $c(X_1, \ldots, X_n)$ is a set of linear constraints (describing a possibly not closed polyhedron). From this information it can be determined whether each argument X_i is bounded or not by projecting $c(X_1, \ldots, X_n)$ onto X_i. If it is bounded (from above and below), and it is known that the ith argument takes on integral values, then it can take only a finite set of values and thus can be declared as f_sig.

Example 3. Consider the following clauses defining a procedure for computing an exponential.

```
exp(Base,Exp,Res)      :- exp_(Base,Exp,1,Res).
exp_(_,0,Ac,Ac).
exp_(Base,Exp,Ac,Res) :- Exp > 0, Exp' is Exp-1, Ac' is Ac*Base,
                         exp_(Base,Exp',Ac',Res)
```

Type inference yields the following signature for the predicate `exp_/4`: `exp_(t24,t24, t24,t24)` with the type `t24 --> 0; 1; num`. A polyhedral analysis of the same program with respect to the goal `exp(Base,10,Res)` yields the following approximation to the queries to `exp_/4`: `exp_(Base,Exp,Ac,Res) :- Exp > -1, Exp =< 10`. Combining this with the inferred type, and assuming that the second argument can take only integer values. the second argument (`Exp`) can be declared as f_sig, and hence we can revert to HEm and do not abstract away the value of the second argument of `exp_/4`. This allows maximum specialization to be achieved.

6.2 Experimental Results

We have incorporated the proposed predefined types f_sig and i_sig within our partial evaluator and instrumented TbHEm to properly handle them as proposed above. Table 2 shows the practical benefits that we obtain on a set of numeric programs which we make extensive use of the arithmetic builtin `is/2`. **exp** and **fib** correspond to the iterative implementations (using accumulators) of the exponential and Fibonacci functions respectively. **vnr** computes a combinatorial function, in this case without accumulators. **list_exp** takes a list of numbers and an exponent and computes a list in which every element is powered to the corresponding exponent (using the predicate `exp/3` defined in **exp**) and also computes the length of the list by using an accumulator. Finally, **dfs** performs a depth-first search avoiding state repetitions in a two dimensional space. Predicate `path/4` computes the path and its cost (using an accumulator) given the initial and final states.

Table 2. Measuring the effects of $\trianglelefteq_T$ with numeric programs

Bench	**Entry**	$\mathbf{T}_{orig}$	$\mathbf{T}_{res\trianglelefteq}$	$\mathbf{T}_{res\trianglelefteq_{num}}$	**PE-type**	$\mathbf{T}_{res\trianglelefteq_T}$
exp	exp(11,1000,_)	19.60	14.60	19.20	exp_(i_sig,f_sig,i_sig,i_sig)	14.20
	exp(11,_,_)	19.20	-	19.20		19.00
fib	fib(1000,_)	17.20	14.20	16.00	fib_(f_sig,i_sig,i_sig,i_sig)	14.00
	fib(_,_)	16.80	-	16.00		15.60
vnr	vnr(10000,1000,_)	31.80	14.20	32.40	vnr(i_sig,f_sig,i_sig)	14.00
	vnr(10000,_,_)	30.00	-	30.00		32.20
dfs	path((1,1),(4,4),_,_)	49.79	15.60	43.39	path_(f_sig,f_sig,i_sig,i_sig,...)	15.80
	path(_,_,_,_)	43.39	-	39.79		42.19
list_exp	lel([1,...,40|_],200,_,_)	32.40	-	32.40	lel_(i_sig,i_sig,i_sig,i_sig)	14.40
	lel(_,200,_,_)	31.80	-	31.60		26.80

In this case, in order to measure the quality of the specialization we compare the execution times of the specialized programs ($\mathbf{T}_{res}$) with the execution times of the original programs ($\mathbf{T}_{orig}$) for sufficiently large inputs. From left to right, the first two columns, **Bench** and **Entry**, show respectively the name of the benchmark and the entry for which the program will be specialized. Then, for each pair benchmark-entry, we show the execution times (in seconds) of the original programs in $\mathbf{T}_{orig}$ and of the corresponding residual programs, by using the three relations $\mathbf{T}_{res\trianglelefteq}$, $\mathbf{T}_{res\trianglelefteq_{num}}$ and $\mathbf{T}_{res\trianglelefteq_T}$. We also show the particular type definition which has been used to guide $\trianglelefteq_T$. Note that in this case we do not consider $\trianglelefteq^*_S$ since it does not produce any significant improvement w.r.t. $\trianglelefteq_{num}$ (constants do not play any role in the involved terms). All times have been computed as the arithmetic means of five runs. For each run, in order to accurately compare the involved programs we run five consecutive times the call `findall(_,Goal,_)`. The particular goals used for measuring the execution times have been chosen to match the entries proposed for each benchmark. As it can be seen, $\trianglelefteq_T$ guarantees termination and outperforms significantly $\trianglelefteq_{num}$. As expected, $\trianglelefteq$ exposes termination problems for some entries as showed in column $\mathbf{T}_{res\trianglelefteq}$. In the examples in which $\trianglelefteq$ terminates, $\trianglelefteq_T$ behaves as well as $\trianglelefteq$. In some examples, no improvements are obtained in the residual programs. This is explained by the fact that the corresponding entries do not provide static information to be used in the specialization. In these examples, it is usual to observe the (unnecessary) over-aggressive nature of $\trianglelefteq$ (even endangering termination in presence of infinite signatures) while, we can see, that the particular type declarations can prevent such undesired behavior in $\trianglelefteq_T$. An interesting observation is that, although many of the examples in this table may be handled in offline PE (by providing the corresponding annotations), there are cases, as **dfs**, where it is not possible to obtain a ranking function for the key arguments. Luckily, we may infer boundedness which is a sufficient condition to effectively use our TbHEm.

7 Discussion and Related Work

Guaranteeing termination is essential in a number of tasks which have to deal with possibly infinite computations. These tasks include PE, abstract model checking, rewriting, etc. Broadly speaking, guaranteeing termination can be tackled in an *offline* or an *online* fashion. The main difference between these two perspectives 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. Traditionally, 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 aim at dynamically guaranteeing termination by supervising the computation in such a way that it is not allowed to proceed 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 important performance gains. However, the online approach is potentially more precise, since we have the concrete values at hand. In offline PE, the problem of termination of local unfolding has been tackled by annotating arguments as "bounded static". The work of Glenstrup and Jones [7] is the main reference, though the idea of bounded static variation goes back a long way. To detect bounded static arguments it is necessary to prove some decrease in well-founded ordering (e.g. using size-change techniques). Quasi-termination is weaker than standard termination but still quite hard to prove. Recent work on this has been done by Vidal [21] and by Glenstrup and Jones [7]. On the other hand, ensuring termination in online PE is easier because we can use "dynamic" termination detection based on supervisors of the computations such as for example embeddings. This means that we do not need any well-founded orderings but only well-quasi-orderings. In effect, in our technique it is only necessary to show boundedness of an argument's values instead of decrease.

In the context of online PE, we have compared TbHEm with the extension of the embedding relation to deal with infinite signatures explained in [11], known as *extended embedding* with static symbols in Sect. 3.3, which is based on a distinction between the different static symbols which occur in the program. As we have shown in the paper, the main advantage of TbHEm is that it achieves a more refined treatment, as it allows treating different arguments in a different way depending on their particular types, which can be automatically inferred by semantic-based analysis, while previous proposals are purely syntactic. Additionally, we have shown that TbHEm can be applied to the specialization of numeric programs, by means of finite signature annotations, in which static constants do not play any role.

Acknowledgments. The authors would like to thank the anonymous referees for their useful comments. This work was funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-15905 *MOBIUS* project, by the Danish Natural

Science Research Council under the FNU-272-06-0574 *SAFT* project, by the Spanish Ministry of Education under the TIN-2005-09207 *MERIT* project, and by the Madrid Regional Government under the S-0505/TIC/0407 *PROMESAS* project.

References

1. Albert, E., Gómez-Zamalloa, M., Hubert, L., Puebla, G.: Verification of Java Bytecode Using Analysis and Transformation of Logic Programs. In: Hanus, M. (ed.) PADL 2007. LNCS, vol. 4354, pp. 124–139. Springer, Heidelberg (2007)
2. Albert, E., Hanus, M., Vidal, G.: A practical partial evaluation scheme for multi-paradigm declarative languages. Journal of Functional and Logic Programming 2002(1) (2002)
3. Albert, E., Puebla, G., Gallagher, J.: Non-leftmost Unfolding in Partial Evaluation of Logic Programs with Impure Predicates. In: Hill, P.M. (ed.) LOPSTR 2005. LNCS, vol. 3901, pp. 115–132. Springer, Heidelberg (2006)
4. Benoy, F., King, A.: Inferring argument size relationships with CLP(R). In: Gallagher, J.P. (ed.) LOPSTR 1996. LNCS, vol. 1207, pp. 204–223. Springer, Heidelberg (1996)
5. Bruynooghe, M., Gallagher, J.P., Van Humbeeck, W.: Inference of Well-Typings for Logic Programs with Application to Termination Analysis. In: Hankin, C., Siveroni, I. (eds.) SAS 2005. LNCS, vol. 3672, pp. 35–51. Springer, Heidelberg (2005)
6. Dershowitz, N., Jouannaud, J.-P.: Rewrite systems. In: van Leeuwen, J. (ed.) Handbook of Theoretical Computer Science, Vol. B, pp. 243–320. Elsevier, Amsterdam (1990)
7. Glenstrup, A.J., Jones, N.D.: Termination analysis and specialization-point insertion in offline partial evaluation. ACM Trans. Program. Lang. Syst. 27(6), 1147–1215 (2005)
8. Hill, P.M., Topor, R.W.: A semantics for typed logic programs. In: Pfenning, F. (ed.) Types in Logic Programming, pp. 1–62. MIT Press, Cambridge (1992)
9. Jones, N.D., Gomard, C.K., Sestoft, P.: Partial Evaluation and Automatic Program Generation. Prentice Hall, New York (1993)
10. Kruskal, J.B.: Well-quasi-ordering, the tree theorem, and Vazsonyi's conjecture. Transactions of the American Mathematical Society 95, 210–225 (1960)
11. Leuschel, M.A.: Homeomorphic Embedding for Online Termination of Symbolic Methods. In: Mogensen, T.Æ., Schmidt, D.A., Sudborough, I.H. (eds.) The Essence of Computation. LNCS, vol. 2566, pp. 379–403. Springer, Heidelberg (2002)
12. Leuschel, M., Bruynooghe, M.: Logic program specialisation through partial deduction: Control issues. Theory and Practice of Logic Programming 2(4&5), 461–515 (2002)
13. Leuschel, M., Martens, B., De Schreye, D.: Controlling Generalisation and Polyvariance in Partial Deduction of Normal Logic Programs. ACM Transactions on Programming Languages and Systems 20(1), 208–258 (1998)
14. Leuschel, M.: On the power of homeomorphic embedding for online termination. In: Levi, G. (ed.) SAS 1998. LNCS, vol. 1503, pp. 230–245. Springer, Heidelberg (1998)
15. Lloyd, J.W., Shepherdson, J.C.: Partial evaluation in logic programming. The Journal of Logic Programming 11, 217–242 (1991)

16. Lloyd, J.W.: Foundations of Logic Programming. Springer, Heidelberg (1987) (second, extended edition)
17. Mycroft, A., O'Keefe, R.A.: A polymorphic type system for Prolog. Artif. Intell. 23(3), 295–307 (1984)
18. Puebla, G., Albert, E., Hermenegildo, M.: Efficient Local Unfolding with Ancestor Stacks for Full Prolog. In: Etalle, S. (ed.) LOPSTR 2004. LNCS, vol. 3573, pp. 149–165. Springer, Heidelberg (2005)
19. Puebla, G., Albert, E., Hermenegildo, M.: Abstract Interpretation with Specialized Definitions. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 107–126. Springer, Heidelberg (2006)
20. Somogyi, Z., Henderson, F., Conway, T.: The Execution Algorithm of Mercury: an Efficient Purely Declarative Logic Programming Language. JLP 3 (October 1996)
21. Vidal, G.: Quasi-Terminating Logic Programs for Ensuring the Termination of Partial Evaluation. In: Proc. of the ACM SIGPLAN 2007 Workshop on Partial Evaluation and Program Manipulation (PEPM 2007), pp. 51–60. ACM Press, New York (2007)