Typed-based Homeomorphic Embedding for Online Termination

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] Computer Science, Roskilde University, DK-4000 Roskilde, Denmark
[3] CLIP, Technical University of Madrid, E-28660 Boadilla del Monte, Madrid, Spain

Abstract. We introduce the *type-based homeomorphic embedding* relation as an extension of the standard, untyped homeomorphic embedding which allows us to obtain more precise results in the presence of infinite signatures (e.g., the integers). In particular, we show how our type-based relation can be used to improve the accuracy of *online partial evaluation*. For this purpose, we outline an approach to constructing suitable types for partial evaluation automatically, given an untyped program and a goal or set of goals. Our approach is based on existing analysis tools for constraint logic programs: (i) inference of a well-typing of a program and goal, and (ii) bounds analysis for numerical values. We argue that our work improves the state of the practice of online termination and it is very relevant for instance in the context of the specialization of interpreters.

1 Introduction

The *homeomorphic embedding* (HEm) relation [10–12] 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 *embeds* expression t_2, written as $t_2 \trianglelefteq t_1$, if t_2 can be obtained from t_1 by deleting some operators, e.g., $\underline{\mathtt{s}}(\mathtt{s}(\underline{\mathtt{U}}+\mathtt{W})\underline{\times}(\underline{\mathtt{U}}+\mathtt{s}(\underline{\mathtt{V}})))$ embeds $\mathtt{s}(\mathtt{U}\times(\mathtt{U}+\mathtt{V}))$.

The HEm relation can be used to guarantee termination because, assuming that 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 be added to the sequence, we first check whether $t_i \not\trianglelefteq t_{n+1}$ for all i s.t. $1 \leq i \leq n$. If that is the case, finiteness is guaranteed and computation can proceed. Otherwise, HEm is not capable of guaranteeing finiteness and the computation has to be stopped. The intuition is that computation can proceed as long as the new expression is not larger than any of the previously computed ones since that is a sign of potential non-termination. The success of HEm is due to the fact that sequences can usually grow considerably large before the whistle blows, when compared to other online approaches to guaranteeing termination.

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` and `name/2` are used. HEm relations over infinite signatures have been defined (e.g. [11, 2]), but they tend to be too conservative in practice ("whistling" too early).

A starting point of our work is the observation that, even if an expression is defined over an infinite signature, it might then only take a finite set of values over such domain for each computation. In this paper, we introduce the *type-based homeomorphic embedding* (TbHEm) relation on typed atoms and typed terms, which by taking context information 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 partition. Intuitively, TbHEm allows expanding sequences as long as, whenever we compare sub-terms from an infinite type, the concrete values which appear in the expression remain within the finite part of the type.

The benefits of TbHEm are illustrated in the context of online Partial Evaluation (PE) [9]. In particular, we use a simplified bytecode interpreter in Prolog whose specialization (if successful) allows decompiling simple bytecode programs to Prolog. For the interpreter, we show how to automatically construct typings which are appropriate to be combined with TbHEm. They are inferred by relying on existing analysis techniques, namely on the inference of well-typings [5]. Moreover, we outline how analysis of numeric bounds can also be used to infer useful information for TbHEm. Such analysis makes over-approximations of the set of values that the program arguments can have. Intuitively, when we can prove that such set of values is bounded, then we know that the infinite partition of the type is empty and, hence, we can safely apply traditional HEm (and improve the effectiveness of PE). Although further experimentation is required, we believe that the examples we present already show the benefits of our approach for the specialization of logic programs with infinite signatures.

2 Embedding in Partial Evaluation with Infinite Signatures

This section intends to illustrate the challenges that infinite signatures pose to online termination based on HEm. For the sake of concreteness, we present our ideas in the context of online PE, but they can be also applied to other online transformation and specialization methods (see [11]). We start by recalling the definition of HEm, which can be found for instance in Leuschel's work [13].

Definition 1 ($\trianglelefteq$). *Given two atoms $A = p(t_1, \ldots, t_n)$ and $B = p(s_1, \ldots, s_n)$, we say that A is embedded by B, 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, $1 \leq i \leq n$.*

Online PE [9] is a semantics-based program transformation technique which specializes a program w.r.t. a given input data, hence, it is often called program specialization. Essentially, partial evaluators are non-standard interpreters which evaluate expressions while termination is guaranteed and specialization is considered profitable. In PE of logic programs, such evaluation basically consists in building a partial SLD tree for a given atom. How to construct the evaluations and when to stop them is determined by the *local control* (also referred to as *unfolding rule*). In state-of-the-art partial evaluators, HEm is used to guarantee termination by ensuring that the sequence of covering ancestors of the atom selected for further unfolding

```
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) :-
    next(PC,PCp),
    S2 = st(PCp,[Z|S],L).
```

```
step(istore(X),st(PC,[I|S],L),S2) :-
    next(PC,PCp),
    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).
....
....
next(PC,PCp) :-
    bytecode(PC,_,N),
    PCp is PC + N.
```

Fig. 1. Fragment of simplified bytecode interpreter

remains finite (see, e.g., [15]). When the embedding whistle blows, evaluation is terminated and the selected atom is passed to the *global control*, whose role is to ensure that we do not try to specialize an infinite number of atoms. Here again, HEm can be applied to guarantee finiteness of the set of atoms which are specialized. Now, if the whistle blows, the atom is generalized so that it no longer embeds any of the previous atoms.

As an 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 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. Furthermore, we have removed the frame stack and therefore only intra-procedural executions are considered. The bytecode to be decompiled is represented as a set of facts `bytecode(PC,Inst,NumBytes)` where `PC` contains the program counter position, `Inst` the particular bytecode instruction, and `NumBytes` the number of bytes the instruction takes up. A state is of the form `st(PC,Stack,Local)` where `Stack` represents the operand stack and `Local` 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/3` recursively with `S2` until we reach a `return` instruction.

Now, we want to decompile a method which receives an integer and executes a loop where a counter (initialized to "0") is incremented by one at each iteration until the counter reaches the value of the input parameter. For this, we partially evaluate the interpreter w.r.t. the bytecode of this method by specializing the atom: `main([N],I)`, where `N` is the input parameter and `I` represents the returned value (i.e. the top of the stack at the end of the computation).

Let us first consider an online partial evaluator[4] which uses HEm to control termination both at the local and global control levels. We do not show the SLD trees built by the partial evaluator nor the decompilation due to space limitations. However, it suffices to know that in the bytecode program, the PC value "2" corresponds to the loop entry. By applying HEm, the evaluation contains a subsequence of atoms of the form: $\texttt{execute}(\texttt{st}(2,[],[\texttt{N},0]),\texttt{S}_\texttt{f})$, $\texttt{execute}(\texttt{st}(2,[],[\texttt{N},1]),\texttt{S}_\texttt{f})$, $\texttt{execute}(\texttt{st}(2,[],[\texttt{N},2]),\texttt{S}_\texttt{f})\ldots$, which correspond to consecutive iterations of the loop in which the control returns to the loop head 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. In order to get a quality decompilation we need to filter the value of the counter (local variable) but not that of the PC. This would result in stopping the derivation when we hit the atom $\texttt{execute}(\texttt{st}(2,[],[\texttt{N},1]),\texttt{S}_\texttt{f})$ and its generalization into $\texttt{execute}(\texttt{st}(2,[],[\texttt{N},\texttt{X}]),\texttt{S}_\texttt{f})$.

A possible relatively straightforward solution in this case is to use the relation $\trianglelefteq_{num}$ which is a slight adaptation of HEm which filters numeric values, i.e., any number embeds any other number. Under this relation, the atom $\texttt{execute}(\texttt{st}(2,[],[\texttt{N},1]),\texttt{S}_\texttt{f})$ embeds $\texttt{execute}(\texttt{st}(2,[],[\texttt{N},0]),\texttt{S}_\texttt{f})$ and therefore we avoid non-termination. Unfortunately, this modification to HEm, though is too conservative and leads to excessive precision loss. For instance, at the beginning of the specialization process we have the atom $\texttt{execute}(\texttt{st}(0,[],[\texttt{N},0]),\texttt{S}_\texttt{f})$ and, after one unfolding step, we obtain the atom $\texttt{execute}(\texttt{st}(1,[0],[\texttt{N},0]),\texttt{S}_\texttt{f})$. By using $\trianglelefteq_{num}$, the whistle blows at this point and unfolding has to stop. Furthermore, the latter atom is generalized 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 specialization of our interpreter, since we lose track of what is the next instruction to execute, which avoids eliminating the interpretation layer and in many cases the residual program ends up containing the original interpreter.

Another solution is to use an extension of the embedding relation, as explained in [11], which is based on a distinction between the finite number of symbols actually occurring in the program and goal. Under this relation, the atom $\texttt{execute}(\texttt{st}(1,[0],[\texttt{N},0]),\texttt{S}_\texttt{f})$ does not embed $\texttt{execute}(\texttt{st}(0,[],[\texttt{N},0]),\texttt{S}_\texttt{f})$, as the numbers 0 and 1 are different static symbols which occur in the program. Hence, we are not forced to generalize them and lose the PC value. However, this extended embedding turns out not to be optimal either since we have that $\texttt{execute}(\texttt{st}(2,[],[\texttt{N},1]),\texttt{S}_\texttt{f})$ does not embed $\texttt{execute}(\texttt{st}(2,[],[\texttt{N},0]),\texttt{S}_\texttt{f})$. This means that we will not stop the unfolding process after evaluating one iteration of the loop, i.e., we proceed with a second iteration of the loop and so on. Although the process terminates once we have unfolded as many iterations of the loop as distinct numbers appear in the program, we are not able to achieve a quality decompilation. For obtaining a 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 $\texttt{execute}(\texttt{st}(2,[],[\texttt{N},0]),\texttt{S}_\texttt{f})$.

This suggests that embeddings that take context into account are needed: an appropriate embedding handling PC values has to be different from one handling numeric values in program variables such as the loop counter.

[4] We assume that we have a partial evaluator which is able to accurately handle built-in predicates and to safely perform non-leftmost unfolding [3].

3 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 functions 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 program-specific instances of extended HEm as defined by Leuschel. Such additional information is program-dependent and might also be goal-dependent; it could be provided manually or be automatically inferred by program analysis, as we will see in Section 4.

3.1 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 [16] 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 $\Sigma_P^\mathcal{N}$ 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 $h(\bar{T}) \longrightarrow f_1(\bar{\tau}_1); \ldots; f_k(\bar{\tau}_k); \ldots \quad (k \geq 1)$ 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 $\bar{T}$.*[5] *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.*

As in Mercury [16], 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 partitions, 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. A type $\tau \in \mathcal{T}$ is labelled (when necessary) with ∞ denoting *infinite* if I is non-empty in the rule defining τ. If a type τ is written with no label then it could be either finite or infinite. Note that there could be different partitions of the same type in different type definitions; 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.

[5] The last condition is known as *transparency* and is necessary to ensure that well-typed programs cannot go wrong [14, 8].

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. The standard concept of a *well-typed program* is assumed, restricted to be *monomorphic* in the sense that the atoms in a clause, and their sub-terms, can be assigned types such that the type assigned to each head and body atom is a variant of the signature for its predicate, and multiple occurrences of the same variable in the clause are assigned the same type. A more general well-typing allows the types of the body atoms to be instances of the signatures rather than variants. It suffices for our purpose to state that, given a well-typed program and a well-typed atomic goal, then each atom arising in computations of the goal (that is, in an SLD tree for the program and goal) has a type that is a variant of its respective signature. In short, a well-typed program and goal generate only well-typed atoms in computations. Furthermore, the monomorphic assumption implies that only a finite number of types arises during computation.

3.2 Type-based Homeomorphic Embedding

We now define TbHEm ($\trianglelefteq_T$). We first define a subsidiary relation on function symbols paired with their associated types.

Definition 3. *Let $\preceq_F$ be the following relation on the set of pairs $\Sigma_P \times \mathcal{T}$; we assume that $\mathcal{T}$ is finite and there is a set of type rules defining the types. Σ_P is possibly infinite, but we assume that the arity of the function symbols is bounded. $(f_1, \tau_1) \preceq_F (f_2, \tau_2)$ iff either $f_1 = f_2 \wedge \tau_1 = \tau_2$ or f_1 and f_2 have the same arity, the rule defining τ_2 is of form $h(\bar{V}) \longrightarrow F; I$, and f_2 is in the infinite partition I.*

Definition 4 ($\trianglelefteq_T$). *We write $t{:}\tau$ to mean that term t is of type τ. 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 A is embedded by B, 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. *$f(s_1, \ldots, s_n) : \tau_1 \trianglelefteq_T g(t_1, \ldots, t_m) : \tau_2$ if $(f, \tau_1) \preceq_F (g, \tau_2)$, and $s_i : \tau_i \trianglelefteq_T t_i : \tau_i'$ for all i, $1 \leq i \leq n$, where $\tau_1, \ldots, \tau_n, \tau_1', \ldots, \tau_n'$ are the respective types of $s_1, \ldots s_n, t_1, \ldots, t_n$.*

Referring to Definition 3, rule 3 specifies that embedding can occur between terms with different function symbols, where the function symbol of the "larger" term is from the I partition of its type. However, as long as we compare distinct terms from an infinite type and remain within the finite part 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 type definition and set of signatures, there is no infinite sequence of well-typed atoms $A_1, A_2, \ldots$ such that for all i, j where $i < j$, $A_i \not\trianglelefteq_T A_j$.*

Proof. (Outline). The ordering defined above can be seen as a special case of the "extended homeomorphic embedding" $\trianglelefteq^*$ [11], which is defined for terms over infinite signatures. The detailed proof shows that the relation $\preceq_F$ is a *well binary relation* on the set $\Sigma_P \times \mathcal{T}$.

We remark that this could be seen as a refinement of the 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). However, the types allow a more fine-grained control over the embedding than is possible with that approach. Also, in Definition 3 functions have to have the same arity in order for the $\preceq_F$ to hold. This restriction could be relaxed, using an ordering on sequences as in the definition of extended homeomorphic embedding [11].

Note that, if we assume an embedding relation based on a given set of types and signatures that is a well-typing for a program, we are assured that the embedding relation is well-defined for all pairs of atoms arising in computations of that program.

4 Automatic Inference of Well-Typings

In this section we outline an approach to constructing in an automatic way suitable types to be used in online partial evaluation in combination with TbHEm, given an untyped program and a goal or set of goals. 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, the 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][6]. The procedure scales well (roughly linear in program size) and is robust, in that every program has a well-typing, and the procedure works with partial programs (modules).

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 bytecode interpreter sketched in Figure 1, together with a particular bytecode program. 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 for the program counter argument `PC` is as follows.

```
t51 --> -8; 0; 1; 2; 3; 4; 5; 6; 7; 8; 9; 10; 11; 12; 13; 14; num
```

This type 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

[6] available on-line at `http://wagner.ruc.dk/Tattoo/`

other words, the partition F of the rule is $\{-8, 0, 1, 2, \ldots, 14\}$ and $I = \texttt{num} \setminus F$. Using the rule structured in this way, the typed-homeomorphic embedding 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). In particular, the atom `execute(st(1,[0],[N,0]))` does not embed `execute(st(0,[],[N,0]))` 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]))` $\trianglelefteq_{\mathtt{T}}$ `execute(st(2,[],[N,1]))`. 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.

5 Analysis of Numeric Bounds

It is important to note that TbHEm allows us to distinguish a finite set of functors (the F component of the type rules) even in the case of infinite signatures. A non-empty I component in type rules often arises during inference of well-typings. We now consider performing additional dataflow analysis in order to infer that the I component in type rules is empty. Indeed, we would like to infer whether a type τ is a *bounded interval*, i.e., if the type rule for τ is of the form $\tau \longrightarrow F; \emptyset$ and F is a finite set of values.

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]. When we can prove that the set of values that all program arguments can have is bounded, then we know that its infinite partition is empty and, hence, we can safely apply traditional HEm (and improve the effectiveness of PE).

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 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.

Example 2. 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.
```

The second argument is thus bounded. Combining this with the inferred type, we obtain the signature `exp_(t24,s,t24,t24)` with the types `t24 --> 0; 1; num` and `s --> 0..10` (we use the interval notation `0..10` as a shortcut to `0; 1; .. ; 10`). Here we assume that the second argument can take on only integer values. The finite type `0..10` implies that the typed-homeomorphic embedding will not abstract away the value of the second argument of `exp_/4` and this allow maximum specialization to be achieved.

6 Discussion and Related Work

Guaranteeing termination is essential in a number of tasks which have to deal with possibly infinite computations. These tasks include partial evaluation, 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 such values. 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. On the other hand, the online approach is potentially more precise, since we have the concrete values at hand, but also more expensive, because of the overhead introduced by the termination supervision.

In the context of partial evaluation, our problem in the online setting is similar to offline termination in that we have to find conditions for ensuring local and global termination. 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 a bit weaker than standard termination but still quite hard to prove. There is Vidal's recent work on this [17] as well as Glenstrup-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.

References

1. E. Albert, M. Gómez-Zamalloa, L. Hubert, and G. Puebla. Verification of Java Bytecode using Analysis and Transformation of Logic Programs. In *Proc. PADL*, number 4354 in LNCS, pages 124–139. Springer-Verlag, 2007.
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. E. Albert, G. Puebla, and J. Gallagher. Non-Leftmost Unfolding in Partial Evaluation of Logic Programs with Impure Predicates. In *Proc. of LOPSTR'05*. Springer LNCS 3901, pages 115–132, April 2006.
4. 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 *Springer-Verlag LNCS*, pages 204–223, August 1996.
5. Maurice Bruynooghe, John Gallagher, and Wouter Van Humbeeck. Inference of well-typings for logic programs with application to termination analysis. LNCS 3672, pages 35–51, 2005.
6. 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.
7. A. J. Glenstrup and N. D. Jones. Termination analysis and specialization-point insertion in offline partial evaluation. *ACM Trans. Program. Lang. Syst.*, 27(6):1147–1215, 2005.
8. Patricia M. Hill and Rodney W. Topor. A semantics for typed logic programs. In Frank Pfenning, editor, *Types in Logic Programming*, pages 1–62. MIT Press, 1992.
9. N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, New York, 1993.
10. J.B. Kruskal. Well-quasi-ordering, the tree theorem, and Vazsonyi's conjecture. *Transactions of the American Mathematical Society*, 95:210–225, 1960.
11. M. Leuschel. Homeomorphic embedding for online termination of symbolic methods. In *The Essence of Computation*, volume 2566 of *LNCS*, pages 379–403. Springer, 2002.
12. 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.
13. Michael Leuschel. On the power of homeomorphic embedding for online termination. In Giorgio Levi, editor, Static Analysis. *Proceedings of SAS'98*, LNCS 1503, pages 230–245, Pisa, Italy, September 1998. Springer-Verlag.
14. Alan Mycroft and Richard A. O'Keefe. A polymorphic type system for Prolog. *Artif. Intell.*, 23(3):295–307, 1984.
15. G. Puebla, E. Albert, and M. Hermenegildo. Efficient Local Unfolding with Ancestor Stacks for Full Prolog. In *Proc. of LOPSTR'04*, pages 149–165. Springer LNCS 3573, 2005.
16. Z. Somogyi, F. Henderson, and T. Conway. The Execution Algorithm of Mercury: an Efficient Purely Declarative Logic Programming Language. *JLP*, 29(1–3), October 1996.
17. G. Vidal. Quasi-Terminating Logic Programs for Ensuring the Termination of Partial Evaluation. In *ACM PEPM'07*, pages 51–60, 2007.